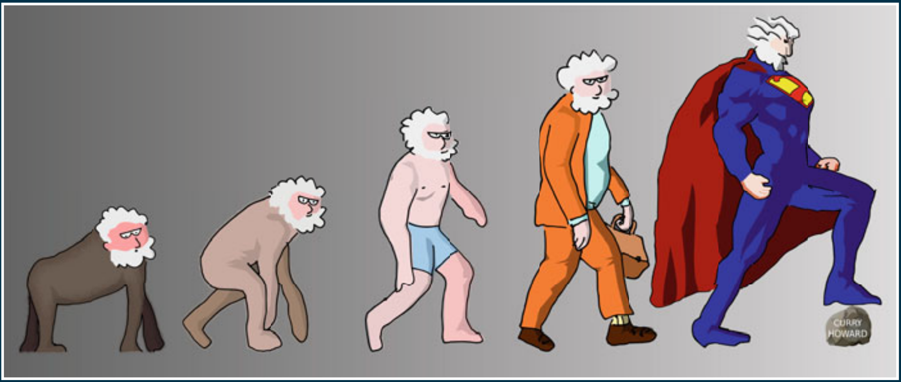
Festschrift

LNCS 9600

Sam Lindley · Conor McBride
Phil Trinder · Don Sannella (Eds.)

A List of Successes That Can Change the World

Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday



Springer

Sam Lindley · Conor McBride
Phil Trinder · Don Sannella (Eds.)

A List of Successes That Can Change the World

Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday

Springer

Editors
Sam Lindley
Informatics Forum
University of Edinburgh
Edinburgh
UK

Conor McBride
University of Strathclyde
Glasgow
UK

Phil Trinder
University of Glasgow
Glasgow
UK

Don Sannella
Informatics Forum
University of Edinburgh
Edinburgh
UK

Cover illustration: The illustration appearing on the cover of this book is the work of Janne Irgens. Used with permission.

Photograph on p. V: The photograph of the honoree was taken by Catherine Lyons. Used with permission.

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-30935-4 ISBN 978-3-319-30936-1 (eBook)
DOI 10.1007/978-3-319-30936-1

Library of Congress Control Number: 2016932871

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Philip Wadler

Preface

This volume is dedicated to Professor Phil Wadler, and the collection of papers form a Festschrift for Phil. The contributions are made by some of the many who know Phil and have been influenced by him. The research papers included here represent some of the areas in which Phil has been active, and the editors thank their colleagues for agreeing to contribute to this Festschrift. Each paper received at least two reviews, and we thank the many external reviewers who helped to make this feasible. These proceedings were prepared with the aid of EasyChair.

In this preface, we attempt to summarize Phil Wadler's scientific achievements. In addition, we describe the personal style and enthusiasm that Phil has brought to the subject.

Introduction

Throughout his career Phil has made fundamental contributions to programming language design and formal systems for capturing the essence of programming language features. In his research he fruitfully combines theory with practice. Many of his contributions have opened up entire fields of investigation, for example, the "Theorems for Free!" paper [82] inspired much research on parametricity. At other times Phil has worked to bring a technique into the mainstream, for example, popularizing the use of monads for describing effectful computations in functional languages [92].

He has contributed to widely used programming languages and important type systems. His contributions to popular languages include many aspects of the design and implementation of Haskell [31], a formal account of the XQuery declarative query language [103], and generic types for Java [6]. His contributions to important type systems include his work on effects [118], linearity [88], and blame [113].

He has also developed experimental programming languages. Orwell was an early experimental lazy functional programming language, and a precursor to Haskell, and Links [14] is a current experimental functional programming language for the Web.

Phil stimulates fellow researchers. He asks hard and unexpected questions, and often provides valuable critical insights. He takes a personal interest in the work of his research students, providing timely and creative feedback.

Phil sets high standards. He will not accept a technical explanation until it is crisp, clear, and concise. As a result, interacting with Phil can sometimes be painful, particularly if your idea has not yet fully crystallized. However, if you persist and carefully address each of Phil's concerns then you often end up with a much deeper understanding of the original problem.

Scientific Contribution

Rather than giving a detailed review of Phil's research, we provide a few highlights. Much of Phil's work is collaborative with other research workers and with his research students and post-docs, often with long-standing collaborations. Our abridged summary does not mention all of the many collaborators in his work, but the list of co-authors in the bibliography below documents these collaborations.

In addition to specific technical contributions outlined below, Phil has provided leadership in the community, for example, serving as chair of the ACM Special Interest Group on Programming Languages (SIGPLAN) between 2009 and 2012.

Programming Languages Phil has made major contributions to programming language design, in particular contributing to functional programming and the theory underpinning functional languages.

Phil's PhD thesis, entitled "Listlessness Is Better Than Laziness," made an early contribution to the field of deforestation, and the title, like many of his research paper titles, contains an apposite wordplay. (Phil's paper titles even appear as the subject of a "spotlight on style" in a book by Helen Sword, *Stylish Academic Writing*.)

In 1984 he developed the Orwell lazy functional language, a precursor of Haskell. In the late 1980s he added ad hoc polymorphism to Haskell in the form of type classes [115]. Type classes became influential, for example, laying the foundation for concepts in C++ and inspiring the design of implicits and traits in Scala.

Phil proposed views [79] as a way of reconciling pattern matching with abstract data types. His work on views was influential in the design of pattern matching for dependently typed programming languages such as Agda and Idris. Abstractions similar to views are now part of F# and Haskell.

Another important piece of work was to develop the theory of, and evangelize the use of, monads for structuring functional programs. Monads were originally proposed by Eugenio Moggi as a basis for reasoning about program equivalence, by making a distinction between values and computations, and allowing the notion of computation to vary. Phil showed how they could be used for a variety of purposes, for instance, expressing queries over collection classes and stateful computations in a functional language. Monads were initially adopted in Haskell, and subsequently in other languages, such as OCaml, F#, and Scala. Phil proposed monad comprehensions [86], which have recently made a comeback in Microsoft's LINQ, F#, and Haskell.

His work on type erasure for Java [6] directly inspired the current design and implementation of generics in Java [53].

His work on the XQuery typed functional language for querying XML [103] is another example of Phil applying theory to improve a programming language technology. He encouraged the development of a formal semantics for XQuery by showing that the natural language semantics was flawed.

In 2005 Phil began working on the Links functional web programming language [14]. From a single source file, Links generates target code for all three tiers of a typical

Web application (client, server, and database). Ideas developed in Links have had significant impact in industry. For instance, formlets [15], a canonical abstraction for constructing Web forms, have become an integral part of IntelliFactory's WebSharper framework for Web programming in F#. Similarly, the Links approach to language-integrated query has influenced the design of Microsoft's LINQ.

Links remains an active research vehicle. Phil currently heads a research project entitled "From Data Types to Session Types—A Basis for Concurrency and Distribution" (or simply "ABCD"). An important part of the ABCD project involves adding session types to Links. The design is partly inspired by Phil's theoretical work connecting classical linear logic and session types [111].

Type Systems Phil has made major contributions to type systems for programming languages and logic.

Phil's work on free theorems [82] helped popularize relational parametricity, a fundamental notion of uniformity enjoyed by second-order logic and hence the polymorphic lambda calculus [106]. Relational parametricity underlies the design of generics in Java and C#. Free theorems and relational parametricity have been applied in areas ranging from dimension types to bidirectional programming to physics.

In the 1990s Phil wrote a series of papers on linear logic [3, 48, 49, 61, 68, 85, 88, 89], promoting its application to programming languages via linear type systems. Nowadays a number of research languages, including F*, Clean, Cyclone, Idris, and Links, make use of some form of linear typing. Mozilla's language Rust, intended as a safe alternative to C, also incorporates a form of linearity. Furthermore, linear types are fundamental to session typing.

Having successfully popularized the idea of using monads to structure effectful programs, Phil later observed a deep correspondence between monads and effect type systems [118]. Recently, his work has inspired the investigation of an important refinement of monads known as graded monads.

Phil has long been interested in the Curry-Howard correspondence between propositions in logic and types in programming languages. His CACM article [114] and regular invited talks at industry conferences are helping to make the Curry-Howard correspondence much more widely appreciated. As well as the Curry-Howard correspondence between second-order intuitionistic logic and polymorphic lambda calculus [106], a notable correspondence Phil has worked on recently is that between classic linear logic and session types [111].

Within the last ten years Phil has developed an interest in gradual typing—systems that combine statically typed programs with untyped programs in a controlled fashion. He introduced the blame calculus [116] (along with the slogan "well-typed programs cannot be blamed"), a formal language for tracking which untyped part of a program causes a run-time error in gradually typed programming languages. He continues to develop the theory of blame, for instance, extending it to account for new features such as polymorphism [1].

Biographical Details

Philip Lee Wadler was born in 1956. He received a Bachelor of Science degree in Mathematics from Stanford University in 1977, and a Master of Science degree in Computer Science from Carnegie Mellon University in 1979. He completed his Doctor of Philosophy in Computer Science at Carnegie Mellon University in 1984. His thesis, "Listlessness Is Better Than Laziness," was supervised by Nico Habermann.

Between 1983 and 1987 Phil worked as a Research Fellow at St. Cross College in Oxford, working in the Programming Research Group with Mary Sheeran, Richard Bird, John Hughes, and others. (Richard and Phil co-authored the influential book *Introduction to Functional Programming* [4]). In 1987 Phil moved to the University of Glasgow to join Mary Sheeran and John Hughes, and shortly thereafter they attracted Simon Peyton Jones to join them. At Glasgow Phil was appointed Reader in 1990, and Professor in 1993. In 1996 he joined Bell Labs, Lucent Technologies, as a researcher. Subsequently, his group was spun out as part of Avaya Labs. In 2003 Phil took up his current position as Professor of Theoretical Computer Science in the Laboratory for Foundations of Computer Science, in the School of Informatics at the University of Edinburgh. In 2006, Phil wrote the O'Reilly book *Java Generics and Collections* [53], co-authored with his friend Maurice Naftalin.

Phil maintains an impressive group of post-docs, PhD students, and visitors. He has held visiting positions in Chalmers University of Technology and the Universities of Sydney and Copenhagen. He is regularly invited to speak at conferences across the world, both academic and industrial. Phil also regularly performs computer science-oriented stand-up comedy.

The quality of Phil's research has been recognized by a plethora of awards, and some of the notable ones are as follows. He was elected to the Royal Society of Edinburgh in 2005, and as an ACM Fellow in 2007. He held a Wolfson-Royal Society Research Merit award between 2004 and 2009. Together with Simon Peyton Jones he was awarded the Most Influential POPL Paper Award in 2003 (for 1993) for their paper "Imperative Functional Programming" [39].

Throughout his career Phil has been passionate about teaching. At Edinburgh he successfully established the first-year Haskell course—Haskell is now the first programming language undergraduate students are exposed to. He has often taught more courses than required, and inspires students with his passion for the subject. This enthusiasm has been recognized, for example, in the Edinburgh University Student Association (EUSA) Teaching Awards 2009.

At the end of 2014, Phil fell ill with what turned out to be a serious illness. Initially, his symptoms were mysterious to doctors, so they performed a full range of tests. He had a serious, but treatable, heart and liver infection. Additionally, but unrelated, a small tumor on one of his kidneys was detected. Phil was fortunate to have the infection when he did, because otherwise the kidney tumor would probably have gone undetected until much later. After the infection had cleared, Phil had the offending kidney removed, and is now fully recovered.

Phil was supported by his wife Catherine from the time they met in Oxford in 1985 until their amicable separation in 2015. Phil and Catherine are devoted parents to their twin son and daughter, Adam and Leora. Phil is actively engaged in wider society, for example, maintaining a liberal Jewish faith, being active in the University and College Union, and the Scottish Green Party.

January 2016

Sam Lindley
Conor McBride
Don Sannella
Phil Trinder

References

1. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011, pp. 201–214. ACM (2011). http://doi.acm.org/10.1145/1926385.1926409
2. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995, pp. 233–246. ACM Press (1995). http://doi.acm.org/10.1145/199448.199507
3. Benton, P.N., Wadler, P.: Linear logic, monads and the lambda calculus. In: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27–30, 1996, pp. 420–431. IEEE Computer Society (1996). http://dx.doi.org/10.1109/LICS.1996.561458
4. Bird, R.S., Wadler, P.: Introduction to Functional Programming. Series in Computer Science. Prentice Hall International, Prentice Hall (1988)
5. Bird, R.S., Wadler, P.: Einführung in die funktionale Programmierung. Hanser Studienbücher der Informatik, Hanser (1992)
6. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: adding genericity to the Java programming language. In: Freeman-Benson, B.N., Chambers, C. (eds.) Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18–22, 1998, pp. 183–200. ACM (1998). http://doi.acm.org/10.1145/286936.286957
7. Brown, A., Fuchs, M., Robie, J., Wadler, P.: MSL - a model for W3C XML schema. In: Shen, V.Y., Saito, N., Lyu, M.R., Zurko, M.E. (eds.) Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1–5, 2001, pp. 191–200. ACM (2001). http://doi.acm.org/10.1145/371920.371982
8. Brown, A., Fuchs, M., Robie, J., Wadler, P.: MSL: a model for W3C XML schema. Comput. Netw. **39**(5), 507–521 (2002). http://dx.doi.org/10.1016/S1389-1286(02)00225-6
9. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 523–549. Springer, Berlin (1998). http://dx.doi.org/10.1007/BFb0054106
10. Cesarini, F., Wadler, P. (eds.): Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang, Snowbird, Utah, USA, September 22, 2004. ACM (2004)
11. Cheney, J., Lindley, S., Radanne, G., Wadler, P.: Effective quotation: relating approaches to language-integrated query. In: Chin, W., Hage, J. (eds.) Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, PEPM 2014, San Diego, California, USA, January 20–21, 2014, pp. 15–26. ACM (2014). http://doi.acm.org/10.1145/2543728.2543738
12. Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA, September 25–27, 2013, pp. 403–416. ACM (2013). http://doi.acm.org/10.1145/2500365.2500586
13. Cheney, J., Lindley, S., Wadler, P.: Query shredding: efficient relational evaluation of queries over nested multisets. In: Dyreson, C.E., Li, F., Özsu, M.T. (eds.) International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014, pp. 1027–1038. ACM (2014). http://doi.acm.org/10.1145/2588555.2612186

14. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Berlin (2006). http://dx.doi.org/10.1007/978-3-540-74792-5_12
15. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The essence of form abstraction. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 205–220. Springer, Berlin (2008). http://dx.doi.org/10.1007/978-3-540-89330-1_15
16. Cooper, E., Wadler, P.: The RPC calculus. In: Porto, A., López-Fraguas, F.J. (eds.) Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Coimbra, Portugal, September 7–9, 2009, pp. 231–242. ACM (2009). http://doi.acm.org/10.1145/1599410.1599439
17. Dahl, V., Wadler, P. (eds.): PADL 2003. LNCS, vol. 2562. Springer, Berlin (2003)
18. Danvy, O., O'Hearn, P.W., Wadler, P.: Preface. Theor. Comput. Sci. **375**(1–3), 1–2 (2007). doi:10.1016/j.tcs.2006.12.024
19. Davis, K., Wadler, P.: Backwards strictness analysis: proved and improved. In: Davis, K., Hughes, J. (eds.) Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21–23 August 1989, Fraserburgh, Scotland, UK. Workshops in Computing, pp. 12–30. Springer (1989)
20. Dreyer, D., Field, J., Giacobazzi, R., Hicks, M., Jagannathan, S., Sagiv, M., Sewell, P., Wadler, P.: Principles of POPL. SIGPLAN Not. **48**(4S), 12–16 (2013). doi:10.1145/2502508.2502517
21. Fankhauser, P., Fernández, M.F., Malhotra, A., Rys, M., Siméon, J., Wadler, P.: The XML query algebra. http://www.w3.org/TR/2000/WD-query-algebra-20001204/, http://www.w3.org/TR/2000/WD-query-algebra-20001204/
22. Fasel, J.H., Hudak, P., Jones, S.L.P., Wadler, P.: SIGPLAN notices special issue on the functional programming language Haskell. SIGPLAN Not. **27**(5), 1 (1992)
23. Fernández, M.F., Siméon, J., Wadler, P.: An algebra for XML query. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 11–45. Springer, Berlin (2000). http://dx.doi.org/10.1007/3-540-44450-5_2
24. Fernández, M.F., Siméon, J., Wadler, P.: A semi-monad for semi-structured data. In: den Bussche, J.V., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 263–300. Springer, Heidelberg (2001). http://dx.doi.org/10.1007/3-540-44503-X_18
25. Hagiya, M., Wadler, P. (eds.): FLOPS 2006. LNCS, vol. 3945. Springer, Berlin (2006)
26. Hall, C.V., Hammond, K., Jones, S.L.P., Wadler, P.: Type classes in Haskell. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, pp. 241–256. Springer, Berlin (1994). http://dx.doi.org/10.1007/3-540-57880-3_16
27. Hall, C.V., Hammond, K., Jones, S.L.P., Wadler, P.: Type classes in Haskell. ACM Trans. Program. Lang. Syst. **18**(2), 109–138 (1996). doi:10.1145/227699.227700
28. Hall, C.V., Hammond, K., Partain, W., Jones, S.L.P., Wadler, P.: The glasgow Haskell compiler: a retrospective. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Proceedings of the 1992 Glasgow Workshop on Functional Programming. Glasgow 1992, Ayr, Scotland, July 6–8, 1992. Workshops in Computing, pp. 62–71. Springer (1992)
29. He, J., Wadler, P., Trinder, P.W.: Typecasting actors: from akka to takka. In: Haller, P., Miller, H. (eds.) Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28–29, 2014, pp. 23–33. ACM (2014). http://doi.acm.org/10.1145/2637647.2637651
30. Heldal, R., Holst, C.K., Wadler, P. (eds.): Functional Programming, Glasgow 1991, Proceedings of the 1991 Glasgow Workshop on Functional Programming, Portree, Isle of Skye, August 12–14, 1991. Workshops in Computing, Springer (1992)

31. Hudak, P., Hughes, J., Jones, S.L.P., Wadler, P.: A history of Haskell: being lazy with class. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference, HOPL-III, San Diego, California, USA, June 9–10, 2007, pp. 1–55. ACM (2007). http://doi.acm.org/10.1145/1238844.1238856
32. Hudak, P., Jones, S.L.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J.H., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R.B., Nikhil, R.S., Partain, W., Peterson, J.: Report on the programming language Haskell, a non-strict, purely functional language. SIGPLAN Not. **27**(5), 1 (1992). http://doi.acm.org/10.1145/130697.130699
33. Hull, R., Thiemann, P., Wadler, P.: 07051 abstracts collection – programming paradigms for the web: web programming and web services. In: Hull, R., Thiemann, P., Wadler, P. (eds.) Programming Paradigms for the Web: Web Programming and Web Services, January 28–February 2, 2007. Dagstuhl Seminar Proceedings, vol. 07051. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007). http://drops.dagstuhl.de/opus/volltexte/2007/1128
34. Hull, R., Thiemann, P., Wadler, P.: 07051 executive summary – programming paradigms for the web: web programming and web services. In: Hull, R., Thiemann, P., Wadler, P. (eds.) Programming Paradigms for the Web: Web Programming and Web Services, January 28–February 2, 2007. Dagstuhl Seminar Proceedings, vol. 07051. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007). http://drops.dagstuhl.de/opus/volltexte/2007/1125
35. Hull, R., Thiemann, P., Wadler, P.: 07051 working group outcomes – programming paradigms for the web: web programming and web services. In: Hull, R., Thiemann, P., Wadler, P. (eds.) Programming Paradigms for the Web: Web Programming and Web Services, January 28–February 2, 2007. Dagstuhl Seminar Proceedings, vol. 07051. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007). http://drops.dagstuhl.de/opus/volltexte/2007/1127
36. Hull, R., Thiemann, P., Wadler, P. (eds.): Programming Paradigms for the Web: Web Programming and Web Services, Dagstuhl Seminar Proceedings, January 28–February 2 2007, vol. 07051. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007). http://drops.dagstuhl.de/portals/07051/
37. Igarashi, A., Pierce, B.C., Wadler, P.: Featherwieght Java: a minimal core calculus for Java and GJ. In: Hailpern, B., Northrop, L.M., Berman, A.M. (eds.) Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1999, Denver, Colorado, USA, November 1–5, 1999, pp. 132–146. ACM (1999). http://doi.acm.org/10.1145/320384.320395
38. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001). http://doi.acm.org/10.1145/503502.503505
39. Jones, S.L.P., Wadler, P.: Imperative functional programming. In: Deusen, M.S.V., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, pp. 71–84. ACM Press (1993). http://doi.acm.org/10.1145/158511.158524
40. Jones, S.L.P., Wadler, P.: The educational pearls column. J. Funct. Program. **13**(5), 833–834 (2003). http://dx.doi.org/10.1017/S0956796803004787
41. Jones, S.L.P., Wadler, P.: Comprehensive comprehensions. In: Keller, G. (ed.) Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007, pp. 61–72. ACM (2007). http://doi.acm.org/10.1145/1291201.1291209

42. Morris, Jr., J.H., Schmidt, E., Wadler, P.: Experience with an applicative string processing language. In: Abrahams, P.W., Lipton, R.J., Bourne, S.R. (eds.) Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980, pp. 32–46. ACM Press (1980). http://doi.acm.org/10.1145/567446.567450
43. King, D.J., Wadler, P.: Combining monads. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, July 6–8, 1992, pp. 134–143. Workshops in Computing, Springer (1992)
44. Launchbury, J., Gill, A., Hughes, J., Marlow, S., Jones, S.L.P., Wadler, P.: Avoiding unnecessary updates. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, July 6–8, 1992, pp. 144–153. Workshops in Computing, Springer (1992)
45. Lindley, S., Wadler, P.: The audacity of hope: thoughts on reclaiming the database dream. In: Gordon, A.D. (ed.) ESOP 2010, ETAPS 2010. LNCS, vol. 6012, p. 1. Springer, Berlin (2010). http://dx.doi.org/10.1007/978-3-642-11957-6_1
46. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. J. Funct. Program. **20**(1), 51–69 (2010). http://dx.doi.org/10.1017/S095679680999027X
47. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electr. Notes Theor. Comput. Sci. **229**(5), 97–117 (2011). http://dx.doi.org/10.1016/j.entcs.2011.02.018
48. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. Electr. Notes Theor. Comput. Sci. **1**, 370–392 (1995). http://dx.doi.org/10.1016/S1571-0661(04)00022-2
49. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. Theor. Comput. Sci. **228**(1–2), 175–210 (1999). http://dx.doi.org/10.1016/S0304-3975(98)00358-2
50. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. **8**(3), 275–317 (1998). http://journals.cambridge.org/action/displayAbstract?aid=44169
51. Marlow, S., Wadler, P.: Deforestation for higher-order functions. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, July 6–8, 1992, pp. 154–165. Workshops in Computing. Springer (1992)
52. Marlow, S., Wadler, P.: A practical subtyping system for Erlang. In: Jones, S.L.P., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, ICFP 1997, Amsterdam, The Netherlands, June 9–11, 1997, pp. 136–149. ACM (1997). http://doi.acm.org/10.1145/258948.258962
53. Naftalin, M., Wadler, P.: Java generics and collections. O'Reilly (2006). http://www.oreilly.de/catalog/javagenerics/index.html
54. Najd, S., Lindley, S., Svenningsson, J., Wadler, P.: Everything old is new again: quoted domain-specific languages. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 25–36. ACM (2016). http://doi.acm.org/10.1145/2847538.2847541
55. Necula, G.C., Wadler, P. (eds.): Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008. ACM (2008). http://dl.acm.org/citation.cfm?id=1328438

56. Odersky, M., Runne, E., Wadler, P.: Two ways to bake your pizza - translating parameterised types into Java. In: Jazayeri, M., Loos, R., Musser, D.R. (eds.) Generic Programming, International Seminar on Generic Programming. LNCS, vol. 1766, pp. 114–132. Springer, Berlin (1998). http://dx.doi.org/10.1007/3-540-39953-4_10
57. Odersky, M., Wadler, P.: Pizza into Java: translating theory into practice. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, January 15–17, 1997, pp. 146–159. ACM Press (1997). http://doi.acm.org/10.1145/263699.263715
58. Odersky, M., Wadler, P. (eds.): Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, Montreal, Canada, September 18–21, 2000. ACM (2000)
59. Odersky, M., Wadler, P., Wehr, M.: A second look at overloading. In: Williams, J. (ed.) Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, La Jolla, California, USA, June 25–28, 1995, pp. 135–146. ACM (1995). http://doi.acm.org/10.1145/224164.224195
60. Sabry, A., Wadler, P.: A reflection on call-by-value. In: Harper, R., Wexelblat, R.L. (eds.) Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, May 24–26, 1996, pp. 13–24. ACM (1996). http://doi.acm.org/10.1145/232627.232631
61. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Program. Lang. Syst. **19** (6), 916–941 (1997). doi:10.1145/267959.269968
62. Schrijvers, T., Stuckey, P.J., Wadler, P.: Monadic constraint programming. J. Funct. Program. **19**(6), 663–697 (2009). doi:10.1017/S0956796809990086
63. Siek, J.G., Thiemann, P., Wadler, P.: Blame and coercion: together again for the first time. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 425–435. ACM (2015). http://doi.acm.org/10.1145/2737924.2737968
64. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17–23, 2010, pp. 365–376. ACM (2010). http://doi.acm.org/10.1145/1706299.1706342
65. Siméon, J., Wadler, P.: The essence of XML (preliminary version). In: Hu, Z., Rodrguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 21–46. Springer, Berlin (2002). http://dx.doi.org/10.1007/3-540-45788-7_2
66. Siméon, J., Wadler, P.: The essence of XML. In: Aiken, A., Morrisett, G. (eds.) Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15–17, 2003, pp. 1–13. ACM (2003). http://doi.acm.org/10.1145/640128.604132
67. Thompson, S.J., Wadler, P.: Functional programming in education - introduction. J. Funct. Program. **3**(1), 3–4 (1993). doi:10.1017/S0956796800000563
68. Turner, D.N., Wadler, P.: Operational interpretations of linear logic. Theor. Comput. Sci. **227**(1–2), 231–248 (1999). doi:10.1016/S0304-3975(99)00054-7
69. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: Williams, J. (ed.) Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, La Jolla, California, USA, June 25–28, 1995, pp. 1–11. ACM (1995). http://doi.acm.org/10.1145/224164.224168
70. Wadler, P.: Analysis of an algorithm for real time garbage collection. Commun. ACM **19** (9), 491–500 (1976). http://doi.acm.org/10.1145/360336.360338

71. Wadler, P.: Applicative style programming, program transformation, and list operators. In: Arvind, Dennis, J.B. (eds.) Proceedings of the 1981 conference on Functional programming languages and computer architecture, FPCA 1981, Wentworth, New Hampshire, USA, October 1981, pp. 25–32. ACM (1981). http://doi.acm.org/10.1145/800223.806759
72. Wadler, P.: Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In: LISP and Functional Programming, pp. 45–52 (1984)
73. Wadler, P.: How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. In: Jouannaud, J. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985). http://dx.doi.org/10.1007/3-540-15975-4_33
74. Wadler, P.: Listlessness is better than laziness II: composing listless functions. In: Ganzinger, H., Jones, N.D. (eds.) Proceedings of a Workshop Programs as Data Objects. LNCS, vol. 217, pp. 282–305. Springer, Berlin (1985). http://dx.doi.org/10.1007/3-540-16446-4_16
75. Wadler, P.: A simple language is also a functional language. Softw. Pract. Exper. **15**(2), 219 (1985). http://dx.doi.org/10.1002/spe.4380150207
76. Wadler, P.: A new array operation. In: Fasel, J.H., Keller, R.M. (eds.) Proceedings of a Workshop on Graph Reduction. LNCS, vol. 279, pp. 328–335. Springer, Berlin (1986). http://dx.doi.org/10.1007/3-540-18420-1_64
77. Wadler, P.: A critique of Abelson and Sussman or why calculating is better than scheming. SIGPLAN Not. **22**(3), 83–94 (1987). http://doi.acm.org/10.1145/24697.24706
78. Wadler, P.: Fixing some space leaks with a garbage collector. Softw. Pract. Exper. **17**(9), 595–608 (1987). http://dx.doi.org/10.1002/spe.4380170904
79. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21–23, 1987, pp. 307–313. ACM Press (1987). http://doi.acm.org/10.1145/41625.41653
80. Wadler, P.: Deforestation: transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 344–358. Springer, New York (1988). http://dx.doi.org/10.1007/3-540-19027-9_23
81. Wadler, P.: Strictness analysis aids time analysis. In: Ferrante, J., Mager, P. (eds.) Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10–13, 1988, pp. 119–132. ACM Press (1988). http://doi.acm.org/10.1145/73560.73571
82. Wadler, P.: Theorems for free! In: Stoy, J.E. (ed.) Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK, September 11–13, 1989, pp. 347–359. ACM (1989). http://doi.acm.org/10.1145/99370.99404
83. Wadler, P.: Comprehending monads. In: LISP and Functional Programming, pp. 61–78 (1990). http://doi.acm.org/10.1145/91556.91592
84. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73** (2), 231–248 (1990). http://dx.doi.org/10.1016/0304-3975(90)90147-A
85. Wadler, P.: Is there a use for linear logic? In: Consel, C., Danvy, O. (eds.) Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 1991, Yale University, New Haven, Connecticut, USA, June 17–19, 1991, pp. 255–273. ACM (1991). http://doi.acm.org/10.1145/115865.115894
86. Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**(4), 461–493 (1992). http://dx.doi.org/10.1017/S0960129500001560

87. Wadler, P.: The essence of functional programming. In: Sethi, R. (ed.) Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19–22, 1992, pp. 1–14. ACM Press (1992). http://doi.acm.org/10.1145/143165.143169
88. Wadler, P.: A syntax for linear logic. In: Brookes, S.D., Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A. (eds.) 9th International Conference on Mathematical Foundations of Programming Semantics. LNCS, vol. 802, pp. 513–529. Springer, Heildelberg (1993). http://dx.doi.org/10.1007/3-540-58027-1_24
89. Wadler, P.: A taste of linear logic. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 185–210. Springer, Berlin (1993). http://dx.doi.org/10.1007/3-540-57182-5_12
90. Wadler, P.: Monads and composable continuations. Lisp Symbolic Comput. **7**(1), 39–56 (1994)
91. Wadler, P.: How to declare an imperative. In: Lloyd, J.W. (ed.), Proceedings of the 1995 International Symposium on Logic Programming, Portland, Oregon, USA, December 4–7, 1995, pp. 18–32. MIT Press (1995)
92. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming. First International Spring School on Advanced Functional Programming Techniques. LNCS, vol. 925, pp. 24–52. Springer, Berlin (1995). http://dx.doi.org/10.1007/3-540-59451-5_2
93. Wadler, P.: Static analysis refuses to stay still: prospects of static analysis for dynamic allocation (abstract). In: Baker, H.G. (ed.) IWMM 1995. LNCS, vol. 986, p. 117. Springer, Berlin (1995). http://dx.doi.org/10.1007/3-540-60368-9_20
94. Wadler, P.: Lazy versus strict. ACM Comput. Surv. **28**(2), 318–320 (1996). http://doi.acm.org/10.1145/234528.234738
95. Wadler, P.: Functional programming: an angry half-dozen. In: Cluet, S., Hull, R. (eds.) 6th International Workshop on Database Programming Languages. DBPL-6. LNCS, vol. 1369, pp. 25–34. Springer, Berlin (1997). http://dx.doi.org/10.1007/3-540-64823-2_2
96. Wadler, P.: How to declare an imperative. ACM Comput. Surv. **29**(3), 240–263 (1997). doi:10.1145/262009.262011
97. Wadler, P.: An angry half-dozen. SIGPLAN Not. **33**(2), 25–30 (1998). doi:10.1145/274930.274933
98. Wadler, P.: The marriage of effects and monads. In: Felleisen, M., Hudak, P., Queinnec, C. (eds.) Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, ICFP 1998, Baltimore, Maryland, USA, September 27–29, 1998, pp. 63–74. ACM (1998). http://doi.acm.org/10.1145/289423.289429
99. Wadler, P.: Why no one uses functional languages. SIGPLAN Not. **33**(8), 23–27 (1998). doi:10.1145/286385.286387
100. Wadler, P.: A formal semantics of patterns in XSLT and XPath. Markup Lang. **2**(2), 183–202 (2000). http://cm.bell-labs.com/cm/cs/who/wadler/papers/xsl-semantics/xsl-semantics.pdf
101. Wadler, P.: Et tu, XML? the downfall of the relational empire (abstract). In: Apers, P.M.G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R.T. (eds.) Proceedings of 27th International Conference on Very Large Data Bases, VLDB 2001, Roma, Italy, September 11–14, 2001, p. 15. Morgan Kaufmann (2001). http://www.vldb.org/conf/2001/P015.pdf
102. Wadler, P.: The Girard-Reynolds isomorphism. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 468–491. Springer, New York (2001). http://dx.doi.org/10.1007/3-540-45500-0_24

103. Wadler, P.: XQuery: a typed functional language for querying XML. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 188–212. Springer, Berlin (2002). http://dx.doi.org/10.1007/978-3-540-44833-4_7
104. Wadler, P.: Call-by-value is dual to call-by-name. SIGPLAN Not. **38**(9), 189–201 (2003). doi:10.1145/944746.944723
105. Wadler, P.: Call-by-value is dual to call-by-name. In: Runciman, C., Shivers, O. (eds.) Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25–29, 2003, pp. 189–201. ACM (2003). http://doi.acm.org/10.1145/944705.944723
106. Wadler, P.: The Girard-Reynolds isomorphism. Inf. Comput. **186**(2), 260–284 (2003). doi:10.1016/S0890-5401(03)00141-X
107. Wadler, P.: Call-by-value is dual to call-by-name - reloaded. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 185–203. Springer, Berlin (2005). http://dx.doi.org/10.1007/978-3-540-32033-3_15
108. Wadler, P.: Faith, evolution, and programming languages: from Haskell to Java to links. In: Tarr, P.L., Cook, W.R. (eds.) Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, Oregon, USA, October 22–26, 2006, p. 508. ACM (2006). http://doi.acm.org/10.1145/1176617.1176623
109. Wadler, P.: The Girard-Reynolds isomorphism (second edition). Theor. Comput. Sci. **375** (1–3), 201–226 (2007). doi:10.1016/j.tcs.2006.12.042
110. Wadler, P.: Propositions as sessions. In: Thiemann, P., Findler, R.B. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9–15, 2012, pp. 273–286. ACM (2012). http://doi.acm.org/10.1145/2364527.2364568
111. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2–3), 384–418 (2014). http://dx.doi.org/10.1017/S095679681400001X
112. Wadler, P.: You and your research and the elements of style. In: Neykova, R., Ng, N. (eds.) 2014 Imperial College Computing Student Workshop, ICCSW 2014, London, UK, September 25–26, 2014. OASICS, vol. 43, p. 2. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014). http://dx.doi.org/10.4230/OASIcs.ICCSW.2014.2
113. Wadler, P.: A complement to blame. In: Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages, SNAPL 2015, Asilomar, California, USA, May 3–6, 2015. LIPIcs, vol. 32, pp. 309–320. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.309
114. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). doi:10.1145/2699407
115. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11–13, 1989, pp. 60–76. ACM Press (1989). http://doi.acm.org/10.1145/75277.75283
116. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009, ETAPS 2009. LNCS, vol. 5502, pp. 1–16. Springer, Berlin (2009). http://dx.doi.org/10.1007/978-3-642-00590-9_1
117. Wadler, P., Hughes, R.J.M.: Projections for strictness analysis. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 385–407. Springer, New York (1987). http://dx.doi.org/10.1007/3-540-18317-5_21
118. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Log. **4**(1), 1–32 (2003). doi:10.1145/601775.601776

Additional Reviewers

Archibald, Blair
Atkey, Robert
Benton, Nick
Brady, Edwin
Campbell, Brian
Chapman, James
Chechina, Natalia
Fehrenbach, Stefan
Gay, Simon
Gill, Andrew
Grust, Torsten
Heunen, Chris
Hofmann, Martin
Hughes, John
Kammar, Ohad
Kaposi, Ambrus
Kiselyov, Oleg
Loidl, Hans-Wolfgang
Macqueen, David
Maier, Patrick
Meusburger, Catherine
Michaelson, Greg
Midtgaard, Jan
Morris, J. Garrett
Morton, John Magnus
Myers, Andrew
Najd, Shayan
Orchard, Dominic
Padovani, Luca
Perera, Roly
Pretnar, Matija
Russo, Claudio
Steuwer, Michel
Stewart, Robert
Uustalu, Tarmo
Vanderbauwhede, Wim
Williams, Jack
Zalewski, Jakub

Contents

Reflections on Monadic Lenses

Faris Abou-Saleh[1], James Cheney[2](✉), Jeremy Gibbons[1], James McKinna[2], and Perdita Stevens[2]

[1] University of Oxford, Oxford, UK
{faris.abou-saleh,jeremy.gibbons}@cs.ox.ac.uk
[2] University of Edinburgh, Edinburgh, UK
{james.cheney,james.mcKinna,perdita.stevens}@ed.ac.uk

Abstract. Bidirectional transformations (bx) have primarily been modeled as pure functions, and do not account for the possibility of the side-effects that are available in most programming languages. Recently several formulations of bx that use monads to account for effects have been proposed, both among practitioners and in academic research. The combination of bx with effects turns out to be surprisingly subtle, leading to problems with some of these proposals and increasing the complexity of others. This paper reviews the proposals for monadic lenses to date, and offers some improved definitions, paying particular attention to the obstacles to naively adding monadic effects to existing definitions of pure bx such as lenses and symmetric lenses, and the subtleties of equivalence of symmetric bidirectional transformations in the presence of effects.

1 Introduction

Programming with multiple concrete representations of the same conceptual information is a commonplace, and challenging, problem. It is commonplace because data is everywhere, and not all of it is relevant or appropriate for every task: for example, one may want to work with only a subset of one's full email account on a mobile phone or other low-bandwidth device. It is challenging because the most direct approach to mapping data across sources A and B is to write separate functions, one mapping to B and one to A, following some (not always explicit) specification of what it means for an A value and a B value to be *consistent*. Keeping these transformations coherent with each other, and with the specification, is a considerable maintenance burden, yet it remains the main approach found in practice.

Over the past decade, a number of promising proposals to ease programming such *bidirectional transformations* have emerged, including *lenses* (Foster et al. 2007), bx based on consistency relations (Stevens 2010), *symmetric lenses* (Hofmann et al. 2011), and a number of variants and extensions (e.g. (Pacheco et al. 2014; Johnson and Rosebrugh 2014)). Most of these proposals consist of an interface with pure functions and some equational laws that characterise good behaviour; the interaction of bidirectionality with other effects has received comparatively little attention.

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 1–31, 2016.
DOI: 10.1007/978-3-319-30936-1_1

Some programmers and researchers have already proposed ways to combine lenses and monadic effects (Diviánszky 2013; Pacheco et al. 2014). Recently, we have proposed symmetric notions of bidirectional computation based on *entangled state monads* (Cheney et al. 2014; Abou-Saleh et al. 2015a) and *coalgebras* (Abou-Saleh et al. 2015b). As a result, there are now several alternative proposals for bidirectional transformations with effects. While this diversity is natural and healthy, reflecting an active research area, the different proposals tend to employ somewhat different terminology, and the relationships among them are not well understood. Small differences in definitions can have disproportionate impact.

In this paper we summarise and compare the existing proposals, offer some new alternatives, and attempt to provide general and useful definitions of "monadic lenses" and "symmetric monadic lenses". Perhaps surprisingly, it appears challenging even to define the composition of lenses in the presence of effects, especially in the symmetric case. We first review the definition of pure asymmetric lenses and two prior proposals for extending them with monadic effects. These definitions have some limitations, and we propose a new definition of monadic lens that overcomes them.

Next we consider the symmetric case. The effectful bx and coalgebraic bx in our previous work are symmetric, but their definitions rely on relatively heavyweight machinery (monad transformers and morphisms, coalgebra). It seems natural to ask whether just adding monadic effects to symmetric lenses in the style of (Hofmann et al. 2011) would also work. We show that, as for asymmetric lenses, adding monadic effects to symmetric lenses is challenging, and give examples illustrating the problems with the most obvious generalisation. We then briefly discuss our recent work on symmetric forms of bx with monadic effects (Cheney et al. 2014; Abou-Saleh et al. 2015a, b). Defining composition for these approaches also turns out to be tricky, and our definition of monadic lenses arose out of exploring this space. The essence of composition of symmetric monadic bx, we now believe, can be presented most easily in terms of monadic lenses, by considering *spans*, an approach also advocated (in the pure case) by Johnson and Rosebrugh (2014).

Symmetric pure bx need to be equipped with a notion of equivalence, to abstract away inessential differences of representation of their "state" or "complement" spaces. As noted by Hofmann et al. (2011) and Johnson and Rosebrugh (2014), isomorphism of state spaces is unsatisfactory, and there are competing proposals for equivalence of symmetric lenses and spans. In the case of spans of monadic lenses, the right notion of equivalence seem even less obvious. We compare three, increasingly coarse, equivalences of spans based on isomorphism (following Abou-Saleh et al. (2015a)), span equivalence (following Johnson and Rosebrugh (2014)), and bisimulation (following Hofmann et al. (2011) and Abou-Saleh et al. (2015b)). In addition, we show a (we think surprising) result: in the pure case, span equivalence and bisimulation equivalence coincide.

In this paper we employ Haskell-like notation to describe and compare formalisms, with a few conventions: we write function composition $f \cdot g$ with a centred dot, and use a lowered dot for field lookup $x.f$, in contrast to Haskell's notation $f\ x$. Throughout the paper, we introduce a number of different representations of lenses, and rather than pedantically disambiguating them all, we freely redefine identifiers as we go. We assume familiarity with common uses of monads in Haskell to encapsulate effects (following Wadler (1995)), and with the **do**-notation (following Wadler's monad comprehensions Wadler (1992)). Although some of these ideas are present or implicit in recent papers (Hofmann et al. 2011; Johnson and Rosebrugh 2014; Cheney et al. 2014; Abou-Saleh et al. 2015a, b), this paper reflects our desire to clarify these ideas and expose them in their clearest form — a desire that is strongly influenced by Wadler's work on a wide variety of related topics (Wadler 1992; King and Wadler 1992; Wadler 1995), and by our interactions with him as a colleague.

2 Asymmetric Monadic Lenses

Recall that a *lens* (Foster et al. 2007, 2012) is a pair of functions, usually called *get* and *put*:

$$\mathbf{data}\ \alpha \rightsquigarrow \beta = \mathit{Lens}\ \{\mathit{get} :: \alpha \to \beta, \mathit{put} :: \alpha \to \beta \to \alpha\}$$

satisfying (at least) the following *well-behavedness* laws:

$$\begin{array}{ll} (\mathsf{GetPut}) & \mathit{put}\ a\ (\mathit{get}\ a) = a \\ (\mathsf{PutGet}) & \mathit{get}\ (\mathit{put}\ a\ b) = b \end{array}$$

The idea is that a lens of type $A \rightsquigarrow B$ maintains a source of type A, providing a view of type B onto it; the well-behavedness laws capture the intuition that the view faithfully reflects the source: if we "get" a b from a source a and then "put" the same b value back into a, this leaves a unchanged; and if we "put" a b into a source a and then "get" from the result, we get b itself. Lenses are often equipped with a *create* function

$$\mathbf{data}\ \alpha \rightsquigarrow \beta = \mathit{Lens}\ \{\mathit{get} :: \alpha \to \beta, \mathit{put} :: \alpha \to \beta \to \alpha, \mathit{create} :: \beta \to \alpha\}$$

satisfying an additional law:

$$(\mathsf{CreateGet}) \quad \mathit{get}\ (\mathit{create}\ b) = b$$

When the distinction is important, we use the term *full* for well-behaved lenses equipped with a *create* operation. It is easy to show that the source and view types of a full lens must either both be empty or both non-empty, and that the *get* operation of a full lens is surjective.

Lenses have been investigated extensively; see for example Foster et al. (2012) for a recent tutorial overview. For the purposes of this paper, we just recall the definition of *composition* of lenses:

$$\begin{array}{l} (;) :: (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \gamma) \to (\alpha \rightsquigarrow \gamma) \\ l_1\ ;\ l_2 = \mathit{Lens}\ (l_2.\mathit{get} \cdot l_1.\mathit{get}) \\ \qquad\qquad\qquad (\lambda a\ c \to l_1.\mathit{put}\ a\ (l_2.\mathit{put}\ (l_1.\mathit{get}\ a)\ c)) \\ \qquad\qquad\qquad (l_1.\mathit{create} \cdot l_1.\mathit{create}) \end{array}$$

which preserves well-behavedness.

2.1 A Naive Approach

As a first attempt, consider simply adding a monadic effect μ to the result types of both *get* and *put*.

$$\textbf{data}\ [\alpha \rightsquigarrow_0 \beta]_\mu = \mathit{MLens}_0\ \{\mathit{mget} :: \alpha \to \mu\ \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha\}$$

Such an approach has been considered and discussed in some recent Haskell libraries and online discussions (Diviánszky 2013). A natural question arises immediately: what laws should a lens $l :: [A \rightsquigarrow_0 B]_M$ satisfy? The following generalisations of the laws appear natural:

$$\begin{array}{lll} (\mathsf{MGetPut_0}) & \textbf{do}\ \{b \leftarrow \mathit{mget}\ a; \mathit{mput}\ a\ b\} & = \mathit{return}\ a \\ (\mathsf{MPutGet_0}) & \textbf{do}\ \{a' \leftarrow \mathit{mput}\ a\ b; \mathit{mget}\ a'\} & = \textbf{do}\ \{a' \leftarrow \mathit{mput}\ a\ b; \mathit{return}\ b\} \end{array}$$

that is, if we "get" b from a and then "put" the same b value back into a, this has the same effect as just returning a (and doing nothing else), and if we "put" a value b and then "get" the result, this has the same effect as just returning b after doing the "put". The obvious generalisation of composition from the pure case for these operations is:

$$\begin{array}{l} (;) :: [\alpha \rightsquigarrow_0 \beta]_\mu \to [\beta \rightsquigarrow_0 \gamma]_\mu \to [\alpha \rightsquigarrow_0 \gamma]_\mu \\ l_1\ ;\ l_2 = \mathit{MLens}_0\ (\lambda a \to \textbf{do}\ \{b \leftarrow l_1.\mathit{mget}\ a; l_2.\mathit{mget}\ b\}) \\ \qquad\qquad\qquad\qquad (\lambda a\ c \to \textbf{do}\ \{b \leftarrow l_1.\mathit{mget}\ a; b' \leftarrow l_2.\mathit{mput}\ b\ c; l_1.\mathit{mput}\ a\ b'\}) \end{array}$$

This proposal has at least two apparent problems. First, the ($\mathsf{MGetPut_0}$) law appears to sharply constrain *mget*: indeed, if *mget a* has an irreversible side-effect then ($\mathsf{MGetPut_0}$) cannot hold. This suggests that *mget* must either be pure, or have side-effects that are reversible by *mput*, ruling out behaviours such as performing I/O during *mget*. Second, it appears difficult to compose these structures in a way that preserves the laws, unless we again make fairly draconian assumptions about μ. In order to show ($\mathsf{MGetPut_0}$) for the composition $l_1\ ;\ l_2$, it seems necessary to be able to commute $l_2.\mathit{mget}$ with $l_1.\mathit{mget}$ and we also need to know that doing $l_1.\mathit{mget}$ twice is the same as doing it just once. Likewise, to show ($\mathsf{MPutGet_0}$) we need to commute $l_2.\mathit{mget}$ with $l_1.\mathit{mput}$.

2.2 Monadic Put-Lenses

Pacheco et al. (2014) proposed a variant of lenses called *monadic putback-oriented lenses*. For the purposes of this paper, the putback-orientation of their approach is irrelevant: we focus on their use of monads, and we provide a slightly simplified version of their definition:

$$\textbf{data}\ [\alpha \rightsquigarrow_1 \beta]_\mu = \mathit{MLens}_1\ \{\mathit{mget} :: \alpha \to \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha\}$$

The main difference from their version is that we remove the *Maybe* type constructors from the return type of *mget* and the first argument of *mput*. Pacheco et al. state laws for these monadic lenses. First, they assume that the monad μ has a *monad membership* operation

$$(\in) :: \alpha \to \mu\ \alpha \to \mathit{Bool}$$

satisfying the following two laws:

$$\begin{array}{ll} (\in\text{-ID}) & x \in \mathit{return}\ x \ \Leftrightarrow \mathit{True} \\ (\in\text{-}\ggg\!=) & y \in (m \ggg\!= f) \Leftrightarrow \exists x\ .\ x \in m \wedge y \in (f\ x) \end{array}$$

Then the laws for MLens_1 (adapted from Pacheco et al. (2014 Proposition 3, p. 49)) are as follows:

$$\begin{array}{lll} (\mathsf{MGetPut}_1) & v = \mathit{mget}\ s & \Longrightarrow \mathit{mput}\ s\ v = \mathit{return}\ s \\ (\mathsf{MPutGet}_1) & s' \in \mathit{mput}\ s\ v' & \Longrightarrow v' = \mathit{mget}\ s' \end{array}$$

In the first law we correct an apparent typo in the original paper, as well as removing the *Just* constructors from both laws. By making *mget* pure, this definition avoids the immediate problems with composition discussed above, and Pacheco et al. outline a proof that their laws are preserved by composition. However, it is not obvious how to generalise their approach beyond monads that admit a sensible $\in$ operation.

Many interesting monads do have a sensible $\in$ operation (e.g. *Maybe*, $[\,]$). Pacheco et al. suggest that $\in$ can be defined for any monad as $x \in m \equiv (\exists h : h\, m = x)$, where h is what they call a "(polymorphic) algebra for the monad at hand, essentially, a function of type $m\ a \to a$ for any type a." However, this definition doesn't appear satisfactory for monads such as *IO*, for which there is no such (pure) function: the ($\in$-ID) law can never hold in this case. It is not clear that we can define a useful $\in$ operation directly for *IO* either: given that $m :: \mathit{IO}\ a$ could ultimately return any a-value, it seems safe, if perhaps overly conservative, to define $x \in m = \mathit{True}$ for any x and m. This satisfies the $\in$ laws, at least, if we make a simplifying assumption that all types are inhabited, and indeed, it seems to be the only thing we could write in Haskell that would satisfy the laws, since we have no way of looking inside the monadic computation $m :: \mathit{IO}\ a$ to find out what its eventual return value is. But then the precondition of the ($\mathsf{MPutGet}_1$) law is always true, which forces the view space to be trivial. These complications suggest, at least, that it would be advantageous to find a definition of monadic lenses that makes sense, and is preserved under composition, for any monad.

2.3 Monadic Lenses

We propose the following definition of monadic lenses for any monad M:

Definition 2.1 (Monadic Lens). A *monadic lens* from source type A to view type B in which the put operation may have effects from monad M (or "M-lens from A to B"), is represented by the type $[A \leadsto B]_M$, where

$$\textbf{data}\ [\alpha \leadsto \beta]_\mu = \mathit{MLens}\ \{\mathit{mget} :: \alpha \to \beta, \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha\}$$

(dropping the μ from the return type of *mget*, compared to the definition in Sect. 2.1). We say that M-lens l is *well-behaved* if it satisfies

$$\begin{array}{ll}(\mathsf{MGetPut}) & \mathbf{do}\,\{\, l.mput\ a\ (l.mget\ a)\,\} = return\ a \\ (\mathsf{MPutGet}) & \mathbf{do}\,\{\, a' \leftarrow l.mput\ a\ b; k\ a'\ (l.mget\ a')\,\} \\ & \quad = \mathbf{do}\,\{\, a' \leftarrow l.mput\ a\ b; k\ a'\ b\,\}\end{array} \qquad \diamondsuit$$

Note that in (MPutGet), we use a continuation $k :: \alpha \rightarrow \beta \rightarrow \mu\ \gamma$ to quantify over all possible subsequent computations in which a' and $l.mget\ a'$ might appear. In fact, using the laws of monads and simply-typed lambda calculus we can prove this law from just the special case $k = \lambda a\ b \rightarrow return\ (a, b)$, so in the sequel when we prove (MPutGet) we may just prove this case while using the strong form freely in the proof.

The ordinary asymmetric lenses are exactly the monadic lenses over $\mu = Id$; the laws then specialise to the standard equational laws. Monadic lenses where $\mu = Id$ are called *pure*, and we may refer to ordinary lenses as pure lenses also.

Definition 2.2. We can also define an operation that lifts a pure lens to a monadic lens:

$$\begin{array}{l} lens2mlens :: Monad\ \mu \Rightarrow \alpha \rightsquigarrow \beta \rightarrow [\alpha \rightsquigarrow \beta]_\mu \\ lens2mlens\ l = MLens\ (l.get)\ (\lambda a\ b \rightarrow return\ (l.put\ a\ b)) \end{array} \qquad \diamondsuit$$

Lemma 2.3. If $l :: Lens\ \alpha\ \beta$ is well-behaved, then so is $lens2mlens\ l$. $\diamondsuit$

Example 2.4. To illustrate, some simple pure lenses include:

$$\begin{array}{l} id_l :: \alpha \rightsquigarrow \alpha \\ id_l = Lens\ (\lambda a \rightarrow a)\ (\lambda\_\ a \rightarrow a) \\ fst_l :: (\alpha, \beta) \rightsquigarrow \alpha \\ fst_l = MLens\ fst\ (\lambda (s_1, s_2)\ s_1' \rightarrow (s_1', s_2)) \end{array}$$

Many more examples of pure lenses are to be found in the literature (Foster et al. 2007, 2012), all of which lift to well-behaved monadic lenses. $\diamondsuit$

As more interesting examples, we present asymmetric versions of the partial and logging lenses presented by Abou-Saleh et al. (2015a). Pure lenses are usually defined using total functions, which means that *get* must be surjective whenever A is nonempty, and *put* must be defined for all source and view pairs. One way to accommodate partiality is to adjust the return type of *get* to *Maybe b* or give *put* the return type *Maybe a* to allow for failure if we attempt to put a b-value that is not in the range of *get*. In either case, the laws need to be adjusted somehow. Monadic lenses allow for partiality without requiring such an ad hoc change. A trivial example is

$$\begin{array}{l} constMLens :: \beta \rightarrow [\alpha \rightsquigarrow \beta]_{Maybe} \\ constMLens\ b = MLens\ (const\ b) \\ \qquad\qquad (\lambda a\ b' \rightarrow \mathbf{if}\ b == b'\ \mathbf{then}\ Just\ a\ \mathbf{else}\ Nothing) \end{array}$$

which is well-behaved because both sides of (MPutGet) fail if the view is changed to a value different from b. Of course, this example also illustrates that the *mget* function of a monadic lens need not be surjective.

As a more interesting example, consider:

$$\begin{array}{l} \mathit{absLens} :: [\mathit{Int} \leadsto \mathit{Int}]_{\mathit{Maybe}} \\ \mathit{absLens} = \mathit{MLens}\ \mathit{abs} \\ \qquad\qquad (\lambda a\ b \to \mathbf{if}\ b < 0 \\ \qquad\qquad\qquad \mathbf{then}\ \mathit{Nothing} \\ \qquad\qquad\qquad \mathbf{else}\ \mathit{Just}\ (\mathbf{if}\ a < 0\ \mathbf{then} - b\ \mathbf{else}\ b)) \end{array}$$

In the *mget* direction, this lens maps a source number to its absolute value; in the reverse direction, it fails if the view b is negative, and otherwise uses the sign of the previous source a to determine the sign of the updated source.

The following *logging lens* takes a pure lens l and, whenever the source value a changes, records the previous a value.

$$\begin{array}{l} \mathit{logLens} :: \mathit{Eq}\ \alpha \Rightarrow \alpha \leadsto \beta \to [\alpha \leadsto \beta]_{\mathit{Writer}\ \alpha} \\ \mathit{logLens}\ l = \mathit{MLens}\ (l.\mathit{get})\ (\lambda a\ b \to \\ \quad \mathbf{let}\ a' = l.\mathit{put}\ a\ b\ \mathbf{in}\ \mathbf{do}\ \{\mathbf{if}\ a \not\equiv a'\ \mathbf{then}\ \mathit{tell}\ a\ \mathbf{else}\ \mathit{return}\ (); \mathit{return}\ a'\}) \end{array}$$

We presented a number of more involved examples of effectful symmetric bx in (Abou-Saleh et al. 2015a). They show how monadic lenses can employ user interaction, state, or nondeterminism to restore consistency. Most of these examples are equivalently definable as *spans* of monadic lenses, which we will discuss in the next section.

In practical use, it is usually also necessary to equip lenses with an *initialisation* mechanism. Indeed, as already mentioned, Pacheco et al.'s monadic put-lenses make the α argument optional (using *Maybe*), to allow for initialisation when only a β is available; we chose to exclude this from our version of monadic lenses above.

We propose the following alternative:

$$\begin{array}{l} \mathbf{data}\ [\alpha \leadsto \beta]_\mu = \mathit{MLens}\ \{\mathit{mget} :: \alpha \to \beta, \\ \qquad\qquad\qquad\qquad \mathit{mput} :: \alpha \to \beta \to \mu\ \alpha, \\ \qquad\qquad\qquad\qquad \mathit{mcreate} :: \beta \to \mu\ \alpha\} \end{array}$$

and we consider such initialisable monadic lenses to be well-behaved when they satisfy the following additional law:

$$(\mathsf{MCreateGet}) \quad \mathbf{do}\ \{a \leftarrow \mathit{mcreate}\ b; k\ a\ (\mathit{mget}\ a)\} = \mathbf{do}\ \{a \leftarrow \mathit{mcreate}\ b; k\ a\ b\}$$

As with (MPutGet), this property follows from the special case $k = \lambda x\ y \to \mathit{return}\ (x, y)$, and we will use this fact freely.

This approach, in our view, helps keep the (GetPut) and (PutGet) laws simple and clear, and avoids the need to wrap *mput*'s first argument in *Just* whenever it is called.

Next, we consider composition of monadic lenses.

$$\begin{array}{l} (;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \leadsto \beta]_\mu \to [\beta \leadsto \gamma]_\mu \to [\alpha \leadsto \gamma]_\mu \\ l_1 ; l_2 = \mathit{MLens}\ (l_2.\mathit{mget} \cdot l_1.\mathit{mget})\ \mathit{mput}\ \mathit{mcreate}\ \mathbf{where} \end{array}$$

$$\begin{array}{l} \mathit{mput}\ a\ c = \mathbf{do}\ \{\, b \leftarrow l_2.\mathit{mput}\ (l_1.\mathit{mget}\ a)\ c; l_1.\mathit{mput}\ a\ b \,\} \\ \mathit{mcreate}\ c = \mathbf{do}\ \{\, b \leftarrow l_2.\mathit{mcreate}; l_1.\mathit{mcreate} \,\} \end{array}$$

Note that we consider only the simple case in which the lenses share a common monad μ. Composing lenses with effects in different monads would require determining how to compose the monads themselves, which is nontrivial (King and Wadler 1992; Jones and Duponcheel 1993).

Theorem 2.5. If $l_1 :: [A \rightsquigarrow B]_M, l_2 :: [B \rightsquigarrow C]_M$ are well-behaved, then so is $l_1\ ;\ l_2$. ♢

3 Symmetric Monadic Lenses and Spans

Hofmann et al. (2011) proposed *symmetric lenses* that use a *complement* to store (at least) the information that is not present in both views.

$$\begin{array}{lll} \mathbf{data}\ \alpha \overset{\gamma}{\longleftrightarrow} \beta = \mathit{SLens}\ \{ & \mathit{put}_{\mathrm{R}} & :: (\alpha,\gamma) \to (\beta,\gamma), \\ & \mathit{put}_{\mathrm{L}} & :: (\beta,\gamma) \to (\alpha,\gamma), \\ & \mathit{missing} & :: \gamma \} \end{array}$$

Informally, $\mathit{put}_{\mathrm{R}}$ turns an α into a β, modifying a complement γ as it goes, and symmetrically for $\mathit{put}_{\mathrm{L}}$; and *missing* is an initial complement, to get the ball rolling. Well-behavedness for symmetric lenses amounts to the following equational laws:

$$\begin{array}{ll} (\mathsf{PutRL}) & \mathbf{let}\ (b,c') = \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a,c)\ \mathbf{in}\ \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b,c') \\ & \quad = \mathbf{let}\ (b,c') = \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a,c)\ \mathbf{in}\ (a,c') \\ (\mathsf{PutLR}) & \mathbf{let}\ (a,c') = \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b,c)\ \mathbf{in}\ \mathit{sl}.\mathit{put}_{\mathrm{R}}\ (a,c') \\ & \quad = \mathbf{let}\ (a,c') = \mathit{sl}.\mathit{put}_{\mathrm{L}}\ (b,c)\ \mathbf{in}\ (b,c') \end{array}$$

Furthermore, the composition of two symmetric lenses preserves well-behavedness, and can be defined as follows:

$$\begin{array}{l} (;) :: (\alpha \overset{\sigma_1}{\longleftrightarrow} \beta) \to (\beta \overset{\sigma_2}{\longleftrightarrow} \gamma) \to (\alpha \overset{(\sigma_1,\sigma_2)}{\longleftrightarrow} \gamma) \\ l_1\ ;\ l_2 = \mathit{SLens}\ \mathit{put}_{\mathrm{R}}\ \mathit{put}_{\mathrm{L}}\ (l_1.\mathit{missing}, l_2.\mathit{missing})\ \mathbf{where} \\ \quad \mathit{put}_{\mathrm{R}}\ (a,(s_1,s_2)) = \mathbf{let}\ (b,s_1') = \mathit{put}_{\mathrm{R}}\ (a,s_1) \\ \qquad\qquad\qquad\qquad\qquad\quad (c,s_2') = \mathit{put}_{\mathrm{R}}\ (b,s_2) \\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (c,(s_1',s_2')) \\ \quad \mathit{put}_{\mathrm{L}}\ (c,(s_1,s_2)) = \mathbf{let}\ (b,s_2') = \mathit{put}_{\mathrm{L}}\ (c,s_2) \\ \qquad\qquad\qquad\qquad\qquad\quad (a,s_1') = \mathit{put}_{\mathrm{L}}\ (b,s_1) \\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (a,(s_1',s_2')) \end{array}$$

We can define an *identity* symmetric lens as follows:

$$\begin{array}{l} \mathit{id}_{\mathrm{sl}} :: \alpha \overset{()}{\longleftrightarrow} \alpha \\ \mathit{id}_{\mathrm{sl}} = \mathit{SLens}\ \mathit{id}\ \mathit{id}\ () \end{array}$$

It is natural to wonder whether symmetric lens composition satisfies identity and associativity laws making symmetric lenses into a category. This is complicated by the fact that the complement types of the composition $id_{\mathrm{sl}}; sl$ and of sl differ, so it is not even type-correct to ask whether $id_{\mathrm{sl}}; sl$ and sl are equal. To make it possible to relate the behaviour of symmetric lenses with different complement types, Hofmann *et al.* defined equivalence of symmetric lenses as follows:

Definition 3.1. Suppose $R \subseteq C_1 \times C_2$. Then $f \sim_R g$ means that for all c_1, c_2, x, if $(c_1, c_2) \in R$ and $(y, c_1') = f\ (x, c_1)$ and $(y', c_2') = g\ (y, c_2)$, then $y = y'$ and $(c_1', c_2') \in R$. ♢

Definition 3.2 (Symmetric Lens Equivalence). Two symmetric lenses $sl_1 :: X \xleftrightarrow{C_1} Y$ and $sl_2 :: X \xleftrightarrow{C_2} Y$ are considered *equivalent* ($sl_1 \equiv_{\mathrm{sl}} sl_2$) if there is a relation $R \subseteq C_1 \times C_2$ such that

1. $(sl_1.missing, sl_2.missing) \in R$,
2. $sl_1.put_{\mathrm{R}} \sim_R sl_2.put_{\mathrm{R}}$, and
3. $sl_1.put_{\mathrm{L}} \sim_R sl_2.put_{\mathrm{L}}$. ♢

Hofmann *et al.* show that $\equiv_{\mathrm{sl}}$ is an equivalence relation; moreover it is sufficiently strong to validate identity, associativity and congruence laws:

Theorem 3.3 (Hofmann et al. 2011). If $sl_1 :: X \xleftrightarrow{C_1} Y$ and $sl_2 :: Y \xleftrightarrow{C_2} Z$ are well-behaved, then so is $sl_1 ; sl_2$. In addition, composition satisfies the laws:

$$\begin{array}{ll} (\mathsf{Identity}) & sl \,;\, id_{\mathrm{sl}} \equiv_{\mathrm{sl}} sl \equiv_{\mathrm{sl}} id_{\mathrm{sl}} \,;\, sl \\ (\mathsf{Assoc}) & sl_1 \,;\, (sl_2 \,;\, sl_3) \equiv_{\mathrm{sl}} (sl_1 \,;\, sl_2) \,;\, sl_3 \\ (\mathsf{Cong}) & sl_1 \equiv_{\mathrm{sl}} sl_1' \wedge sl_2 \equiv_{\mathrm{sl}} sl_2' \Longrightarrow sl_1 \,;\, sl_2 \equiv_{\mathrm{sl}} sl_1' \,;\, sl_2' \end{array}$$ ♢

3.1 Naive Monadic Symmetric Lenses

We now consider an obvious monadic generalisation of symmetric lenses, in which the put_{L} and put_{R} functions are allowed to have effects in some monad M:

Definition 3.4. A *monadic symmetric lens* from A to B with complement type C and effects M consists of two functions converting A to B and vice versa, each also operating on C and possibly having effects in M, and a complement value *missing* used for initialisation:

$$\begin{array}{l} \mathbf{data}\ [\alpha \xleftrightarrow{\gamma} \beta]_{\mu} = SMLens\ \{ mput_{\mathrm{R}} :: (\alpha, \gamma) \to \mu\ (\beta, \gamma), \\ \qquad\qquad\qquad\qquad\qquad\quad mput_{\mathrm{L}} :: (\beta, \gamma) \to \mu\ (\alpha, \gamma), \\ \qquad\qquad\qquad\qquad\qquad\quad missing :: \gamma \} \end{array}$$

Such a lens sl is called *well-behaved* if:

$$\begin{array}{ll} (\mathsf{PutRLM}) & \mathbf{do}\ \{(b, c') \leftarrow sl.mput_{\mathrm{R}}\ (a, c); sl.mput_{\mathrm{L}}\ (b, c')\} \\ & \quad = \mathbf{do}\ \{(b, c') \leftarrow sl.mput_{\mathrm{R}}\ (a, c); return\ (a, c')\} \\ (\mathsf{PutLRM}) & \mathbf{do}\ \{(a, c') \leftarrow sl.mput_{\mathrm{L}}\ (b, c); sl.mput_{\mathrm{R}}\ (a, c')\} \\ & \quad = \mathbf{do}\ \{(a, c') \leftarrow sl.mput_{\mathrm{L}}\ (b, c); return\ (b, c')\} \end{array}$$ ♢

The above monadic generalisation of symmetric lenses appears natural, but it turns out to have some idiosyncrasies, similar to those of the naive version of monadic lenses we considered in Sect. 2.1.

Composition and Well-Behavedness. Consider the following candidate definition of composition for monadic symmetric lenses:

$$\begin{array}{l}
(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \xleftrightarrow{\sigma_1} \beta]_\mu \rightarrow [\beta \xleftrightarrow{\sigma_2} \gamma]_\mu \rightarrow [\alpha \overset{(\sigma_1,\sigma_2)}{\longleftrightarrow} \gamma]_\mu \\
\mathit{sl}_1 \mathbin{;} \mathit{sl}_2 = \mathit{SMLens}\ \mathit{put}_{\mathrm{R}}\ \mathit{put}_{\mathrm{L}}\ \mathit{missing}\ \textbf{where} \\
\quad \mathit{put}_{\mathrm{R}}\ (a,(s_1,s_2)) = \textbf{do}\ \{ (b,s_1') \leftarrow \mathit{sl}_1.\mathit{mput}_{\mathrm{R}}\ (a,s_1); \\
\qquad\qquad\qquad\qquad\quad (c,s_2') \leftarrow \mathit{sl}_2.\mathit{mput}_{\mathrm{R}}\ (b,s_2); \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (c,(s_1',s_2'))\} \\
\quad \mathit{put}_{\mathrm{L}}\ (c,(s_1,s_2)) = \textbf{do}\ \{ (b,s_2') \leftarrow \mathit{sl}_2.\mathit{mput}_{\mathrm{L}}\ (c,s_2); \\
\qquad\qquad\qquad\qquad\quad (a,s_1') \leftarrow \mathit{sl}_1.\mathit{mput}_{\mathrm{L}}\ (b,s_1); \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (a,(s_1',s_2'))\} \\
\quad \mathit{missing} \qquad\quad = (\mathit{sl}_1.\mathit{missing}, \mathit{sl}_2.\mathit{missing})
\end{array}$$

which seems to be the obvious generalisation of pure symmetric lens composition to the monadic case. However, it does not always preserve well-behavedness.

Example 3.5. Consider the following construction:

$$\begin{array}{l}
\mathit{setBool} \quad :: \mathit{Bool} \rightarrow [() \xleftrightarrow{()} ()]_{\mathit{State\ Bool}} \\
\mathit{setBool}\ b = \mathit{SMLens}\ m\ m\ ()\ \textbf{where}\ m\ \_ = \textbf{do}\ \{\mathit{set}\ b; \mathit{return}\ ((),())\}
\end{array}$$

The lens *setBool True* has no effect on the complement or values, but sets the state to *True*. Both *setBool True* and *setBool False* are well-behaved, but their composition (in either direction) is not: (PutRLM) fails for *setBool True*; *setBool False* because *setBool True* and *setBool False* share a single *Bool* state value. ♢

Proposition 3.6. *setBool b* is well-behaved for $b \in \{\mathit{True}, \mathit{False}\}$, but *setBool True* ; *setBool False* is not well-behaved. ♢

Composition does preserve well-behavedness for commutative monads, i.e. those for which

$$\textbf{do}\ \{a \leftarrow x; b \leftarrow y; \mathit{return}\ (a,b)\} = \textbf{do}\ \{b \leftarrow y; a \leftarrow x; \mathit{return}\ (a,b)\}$$

but this rules out many interesting monads, such as *State* and *IO*.

3.2 Entangled State Monads

The types of the $\mathit{mput}_{\mathrm{R}}$ and $\mathit{mput}_{\mathrm{L}}$ operations of symmetric lenses can be seen (modulo mild reordering) as stateful operations in the *state monad* $\mathit{State}\ \gamma\ \alpha = \gamma \rightarrow (\alpha, \gamma)$, where the state $\gamma = C$. This observation was also anticipated by Hofmann et al. In a sequence of papers, we considered generalising these operations and their laws to an arbitrary monad (Cheney et al. 2014; Abou-Saleh et al. 2015a, b). In our initial workshop paper, we proposed the following definition:

$$\mathbf{data}\ [\alpha \mathrel{\rightleftharpoons\!\!\!\!/} \beta]_\mu = \mathit{SetBX}\ \{\, \mathit{get}_L :: \mu\ \alpha, \mathit{set}_L :: \alpha \to \mu\ (), \\ \mathit{get}_R :: \mu\ \beta, \mathit{set}_R :: \beta \to \mu\ ()\,\}$$

subject to a subset of the *State* monad laws (Plotkin and Power 2002), such as:

$$(\mathsf{Get_L Set_L})\quad \mathbf{do}\ \{\, a \leftarrow \mathit{get}_L; \mathit{set}_L\ a \,\} = \mathit{return}\ ()$$
$$(\mathsf{Set_L Get_L})\quad \mathbf{do}\ \{\, \mathit{set}_L\ a; \mathit{get}_L \,\} = \mathbf{do}\ \{\, \mathit{set}_L\ a; \mathit{return}\ a \,\}$$

This presentation makes clear that bidirectionality can be viewed as a state effect in which two "views" of some common state are *entangled*. That is, rather than storing a pair of views, each independently variable, they are entangled, in the sense that a change to either may also change the other. Accordingly, the entangled state monad operations do *not* satisfy all of the usual laws of state: for example, the set_L and set_R operations do not commute.

However, one difficulty with the entangled state monad formalism is that, as discussed in Sect. 2.1, effectful *mget* operations cause problems for composition. It turned out to be nontrivial to define a satisfactory notion of composition, even for the well-behaved special case where $\mu = \mathit{StateT}\ \sigma\ \nu$ for some ν, where *StateT* is the *state monad transformer* (Liang et al. 1995), i.e. $\mathit{StateT}\ \sigma\ \nu\ \alpha = \sigma \to \nu\ (\alpha, \sigma)$. We formulated the definition of *monadic lenses* given earlier in this paper in the process of exploring this design space.

3.3 Spans of Monadic Lenses

Hofmann et al. (2011) showed that a symmetric lens is equivalent to a *span* of two ordinary lenses, and later work by Johnson and Rosebrugh (2014) investigated such *spans of lenses* in greater depth. Accordingly, we propose the following definition:

Definition 3.7 (Monadic Lens Spans). A span of monadic lenses ("*M*-lens span") is a pair of *M*-lenses having the same source:

$$\mathbf{type}\ [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu = \mathit{Span}\ \{\, \mathit{left} :: [\sigma \rightsquigarrow \alpha]_\mu, \mathit{right} :: [\sigma \rightsquigarrow \beta]_\mu \,\}$$

We say that an *M*-lens span is well-behaved if both of its components are. ◇

We first note that we can extend either leg of a span with a monadic lens (preserving well-behavedness if the arguments are well-behaved):

$$(\lhd)\quad :: \mathit{Monad}\ \mu \Rightarrow [\alpha_1 \rightsquigarrow \alpha_2]_\mu \to [\alpha_1 \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu \to [\alpha_2 \leftsquigarrow \sigma \rightsquigarrow \beta]_\mu$$
$$\mathit{ml} \lhd \mathit{sp} = \mathit{Span}\ (\mathit{sp.left} \mathbin{;} \mathit{ml})\ (\mathit{sp.right})$$
$$(\rhd)\quad :: \mathit{Monad}\ \mu \Rightarrow [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta_1]_\mu \to [\beta_1 \rightsquigarrow \beta_2]_\mu \to [\alpha \leftsquigarrow \sigma \rightsquigarrow \beta_2]_\mu$$
$$\mathit{sp} \rhd \mathit{ml} = \mathit{Span}\ \mathit{sp.left}\ (\mathit{sp.right} \mathbin{;} \mathit{ml})$$

To define composition, the basic idea is as follows. Given two spans $[A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $[B \leftsquigarrow S_2 \rightsquigarrow C]_M$ with a common type B "in the middle",

$$\begin{tikzcd}
 & & S_1 \bowtie S_2 \arrow[dl, dotted] \arrow[dr, dotted] & & \\
 & S_1 \arrow[dl] \arrow[dr] & & S_2 \arrow[dl] \arrow[dr] & \\
A & & B & & C
\end{tikzcd}$$

Fig. 1. Composing spans of lenses

we want to form a single span from A to C. The obvious thing to try is to form a pullback of the two monadic lenses from S_1 and S_2 to the common type B, obtaining a span from some common state type S to the state types S_1 and S_2, and composing with the outer legs. (See Fig. 1.) However, the category of monadic lenses doesn't have pullbacks (as Johnson and Rosebrugh note, this is already the case for ordinary lenses). Instead, we construct the appropriate span as follows.

$$\begin{array}{l}
(\bowtie) :: \mathit{Monad}\ \mu \Rightarrow [\sigma_1 \rightsquigarrow \beta]_\mu \to [\sigma_2 \rightsquigarrow \beta]_\mu \to [\sigma_1 \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu \\
l_1 \bowtie l_2 = \mathit{Span}\ (\mathit{MLens}\ \mathit{fst}\ \mathit{put}_{\mathrm{L}}\ \mathit{create}_{\mathrm{L}})\ (\mathit{MLens}\ \mathit{snd}\ \mathit{put}_{\mathrm{R}}\ \mathit{create}_{\mathrm{R}})\ \mathbf{where} \\
\quad \mathit{put}_{\mathrm{L}}\ (\_, s_2)\ s_1' = \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mput}\ s_2\ (l_1.\mathit{mget}\ s_1'); \mathit{return}\ (s_1', s_2') \} \\
\quad \mathit{create}_{\mathrm{L}}\ s_1 \qquad = \mathbf{do}\ \{ s_2' \leftarrow l_2.\mathit{mcreate}\ (l_1.\mathit{mget}\ s_1); \mathit{return}\ (s_1, s_2') \} \\
\quad \mathit{put}_{\mathrm{R}}\ (s_1, \_)\ s_2' = \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mput}\ s_1\ (l_2.\mathit{mget}\ s_2'); \mathit{return}\ (s_1', s_2') \} \\
\quad \mathit{create}_{\mathrm{R}}\ s_1 \qquad = \mathbf{do}\ \{ s_1' \leftarrow l_1.\mathit{mcreate}\ (l_2.\mathit{mget}\ s_2); \mathit{return}\ (s_1', s_2) \}
\end{array}$$

where we write $S_1 \bowtie S_2$ for the type of *consistent* state pairs $\{(s_1, s_2) \in S_1 \times S_2 \mid l_1.\mathit{mget}\ (s_1) = l_2.\mathit{mget}\ (s_2)\}$. In the absence of dependent types, we represent this type as (S_1, S_2) in Haskell, and we need to check that the *mput* and *mcreate* operations respect the consistency invariant.

Lemma 3.8. If $ml_1 :: [S_1 \rightsquigarrow B]_M$ and $ml_2 :: [S_2 \rightsquigarrow B]_M$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [S_1 \leftsquigarrow (S_1 \bowtie S_2) \rightsquigarrow S_2]_\mu$. ♢

Note that (MPutGet) and (MCreateGet) hold by construction and do not need the corresponding properties for l_1 and l_2, but these properties are needed to show that consistency is established by *mcreate* and preserved by *mput*.

We can now define composition as follows:

$$\begin{array}{l}
(;) :: \mathit{Monad}\ \mu \Rightarrow [\alpha \leftsquigarrow \sigma_1 \rightsquigarrow \beta]_\mu \to [\beta \leftsquigarrow \sigma_2 \rightsquigarrow \gamma]_\mu \to [\alpha \leftsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \gamma]_\mu \\
\mathit{sp}_1\ ;\ \mathit{sp}_2 = \mathit{sp}_1.\mathit{left} \lhd (\mathit{sp}_1.\mathit{right} \bowtie \mathit{sp}_2.\mathit{left}) \rhd \mathit{sp}_2.\mathit{right}
\end{array}$$

The well-behavedness of the composition of two well-behaved spans is immediate because $\lhd$ and $\rhd$ preserve well-behavedness of their arguments:

Theorem 3.9. If $\mathit{sp}_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $\mathit{sp}_2 :: [B \leftsquigarrow S_2 \rightsquigarrow C]_M$ are well-behaved spans of monadic lenses, then their composition $\mathit{sp}_1\ ;\ \mathit{sp}_2$ is well-behaved. ♢

Given a span of monadic lenses $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$, we can construct a monadic symmetric lens $sl :: [A \overset{Maybe\ S}{\longleftrightarrow} B]_M$ as follows:

```
span2smlens (left, right) = SMLens mput_R mput_L Nothing where
  mput_R (a, Just s)    = do {s' ← left.mput s a; return (right.mget s', Just s')}
  mput_R (a, Nothing)   = do {s' ← left.mcreate a; return (right.mget s', Just s')}
  mput_L (b, Just s)    = do {s' ← right.mput s b; return (left.mget s', Just s')}
  mput_L (b, Nothing)   = do {s' ← right.mcreate b; return (left.mget s', Just s')}
```

Essentially, these operations use the span's *mput* and *mget* operations to update one side and obtain the new view value for the other side, and use the *mcreate* operations to build the initial S state if the complement is *Nothing*.

Well-behavedness is preserved by the conversion from monadic lens spans to *SMLens*, for arbitrary monads M:

Theorem 3.10. If $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$ is well-behaved, then *span2smlens sp* is also well-behaved. ◇

Given $sl :: [A \overset{C}{\longleftrightarrow} B]_M$, let $S \subseteq A \times B \times C$ be the set of *consistent triples* (a, b, c), that is, those for which $sl.mput_R\ (a, c) = return\ (b, c)$ and $sl.mput_L\ (b, c) = return\ (a, c)$. We construct $sp :: [A \leftsquigarrow S \rightsquigarrow B]_M$ by

```
smlens2span sl = Span (MLens get_L put_L create_L) (MLens get_R put_R create_R)
where
  get_L (a, b, c)      = a
  put_L (a, b, c) a'   = do {(b', c') ← sl.mput_R (a', c); return (a', b', c')}
  create_L a           = do {(b, c) ← sl.mput_R (a, sl.missing); return (a, b, c)}
  get_R (a, b, c)      = b
  put_R (a, b, c) b'   = do {(a', c') ← sl.mput_L (b', c); return (a', b', c')}
  create_R b           = do {(a, c) ← sl.mput_L (b, sl.missing); return (a, b, c)}
```

However, *smlens2span* may not preserve well-behavedness even for simple monads such as *Maybe*, as the following counterexample illustrates.

Example 3.11. Consider the following monadic symmetric lens construction:

```
fail :: [() ⟷^() ()]_Maybe
fail = SMLens Nothing Nothing ()
```

This is well-behaved but *smlens2span fail* is not. In fact, the set of consistent states of *fail* is empty, and each leg of the induced span is of the following form:

```
failMLens :: MLens Maybe ∅ ()
failMLens = MLens (λ_ → ()) (λ_ () → Nothing) (λ_ → Nothing)
```

which fails to satisfy (MGetPut). ◇

For pure symmetric lenses, *smlens2span* does preserve well-behavedness.

Theorem 3.12. If *sl* :: *SMLens Id C A B* is well-behaved, then *smlens2span sl* is also well-behaved, with state space S consisting of the consistent triples of *sl*. ◇

To summarise: spans of monadic lenses are closed under composition, and correspond to well-behaved symmetric monadic lenses. However, there are well-behaved symmetric monadic lenses that do not map to well-behaved spans. It seems to be an interesting open problem to give a direct axiomatisation of the symmetric monadic lenses that are essentially spans of monadic lenses (and are therefore closed under composition).

4 Equivalence of Spans

Hofmann et al. (2011) introduced a bisimulation-like notion of equivalence for pure symmetric lenses, in order to validate laws such as identity, associativity and congruence of composition. Johnson and Rosebrugh (2014) introduced a definition of equivalence of spans and compared it with symmetric lens equivalence. We have considered equivalences based on isomorphism (Abou-Saleh et al. 2015a) and bisimulation (Abou-Saleh et al. 2015b). In this section we consider and relate these approaches in the context of spans of M-lenses.

Definition 4.1 (Isomorphism Equivalence). Two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ are isomorphic ($sp \equiv_{\mathrm{i}} sp'$) if there is an isomorphism $h :: S_1 \to S_2$ on their state spaces such that $h \mathbin{;} sp_2.\mathit{left} = sp_1.\mathit{left}$ and $h \mathbin{;} sp_2.\mathit{right} = sp_1.\mathit{right}$. ◊

Note that any isomorphism $h :: S_1 \to S_2$ can be made into a (monadic) lens; we omit the explicit conversion.

We consider a second definition of equivalence, inspired by Johnson and Rosebrugh (2014), which we call *span equivalence*:

Definition 4.2 (Span Equivalence). Two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ are related by $\curvearrowright$ if there is a full lens $h :: S_1 \rightsquigarrow S_2$ such that $h \mathbin{;} sp_2.\mathit{left} = sp_1.\mathit{left}$ and $h \mathbin{;} sp_2.\mathit{right} = sp_1.\mathit{right}$. The equivalence relation $\equiv_{\mathrm{s}}$ is the least equivalence relation containing $\curvearrowright$. ◊

One important consideration emphasised by Johnson and Rosebrugh is the need to avoid making all compatible spans equivalent to the "trivial" span $[A \leftsquigarrow \emptyset \rightsquigarrow B]_M$. To avoid this problem, they imposed conditions on h: its *get* function must be surjective and *split*, meaning that there exists a function c such that $h.\mathit{get} \cdot c = \mathit{id}$. We chose instead to require h to be a full lens. This is actually slightly stronger than Johnson and Rosebrugh's definition, at least from a constructive perspective, because h is equipped with a specific choice of $c = \mathit{create}$ satisfying $h.\mathit{get} \cdot c = \mathit{id}$, that is, the (CreateGet) law.

We have defined span equivalence as the reflexive, symmetric, transitive closure of $\curvearrowright$. Interestingly, even though span equivalence allows for an arbitrary sequence of (pure) lenses between the respective state spaces, it suffices to consider only spans of lenses. To prove this, we first state a lemma about the ($\bowtie$) operation used in composition. Its proof is straightforward equational reasoning.

Lemma 4.3. Suppose $l_1 :: A \rightsquigarrow B$ and $l_2 :: C \rightsquigarrow B$ are pure lenses. Then $(l_1 \bowtie l_2).\mathit{left} \mathbin{;} l_1 = (l_1 \bowtie l_2).\mathit{right} \mathbin{;} l_2$. ♢

Theorem 4.4. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathrm{s}} sp_2$ then there exists $sp :: S_1 \leftsquigarrow S \rightsquigarrow S_2$ such that $sp.\mathit{left} \mathbin{;} sp_1.\mathit{left} = sp.\mathit{right} \mathbin{;} sp_2.\mathit{left}$ and $sp.\mathit{left} \mathbin{;} sp_1.\mathit{right} = sp.\mathit{right} \mathbin{;} sp_2.\mathit{right}$. ♢

Proof. Let sp_1 and sp_2 be given such that $sp_1 \equiv_{\mathrm{s}} sp_2$. The proof is by induction on the length of the sequence of $\curvearrowright$ or $\curvearrowleft$ steps linking sp_1 to sp_2.

If $sp_1 = sp_2$ then the result is immediate. If $sp_1 \curvearrowright sp_2$ then we can complete a span between S_1 and S_2 using the identity lens. For the inductive case, suppose that the result holds for sequences of up to $n \curvearrowright$ or $\curvearrowleft$ steps, and suppose $sp_1 \equiv_{\mathrm{s}} sp_2$ holds in $n \curvearrowright$ or $\curvearrowleft$ steps. There are two cases, depending on the direction of the first step. If $sp_1 \curvearrowleft sp_3 \equiv_{\mathrm{s}} sp_2$ then by induction we must have a pure span sp between S_3 and S_2 and $sp_1 \curvearrowleft sp_3$ holds by virtue of a lens $h :: S_3 \to S_1$, so we can simply compose h with $sp.\mathit{left}$ to obtain the required span between S_1 and S_2. Otherwise, if $sp_1 \curvearrowright sp_3 \equiv_{\mathrm{s}} sp_2$ then by induction we must have a pure span sp between S_3 and S_2 and we must have a lens $h :: S_1 \to S_3$, so we use Lemma 4.3 to form a span $sp_0 :: S_1 \leftsquigarrow (S_1 \bowtie S_3) \rightsquigarrow S_3$ and extend $sp_0.\mathit{right}$ with $sp.\mathit{right}$ to form the required span between S_1 and S_3. □

Thus, span equivalence is a doubly appropriate name for $\equiv_{\mathrm{s}}$: it is an equivalence of spans witnessed by a (pure) span.

Finally, we consider a third notion of equivalence, inspired by the natural bisimulation equivalence for coalgebraic bx (Abou-Saleh et al. 2015b):

Definition 4.5 (Base Map). Given M-lenses $l_1 :: [S_1 \rightsquigarrow V]_M$ and $l_2 :: [S_2 \rightsquigarrow V]_M$, we say that $h : S_1 \to S_2$ is a *base map* from l_1 to l_2 if

$$\begin{aligned}
&l_1.\mathit{mget}\ s &&= l_2.\mathit{mget}\ (h\ s)\\
&\mathbf{do}\ \{s \leftarrow l_1.\mathit{mput}\ s\ v; \mathit{return}\ (h\ s)\} &&= l_2.\mathit{mput}\ (h\ s)\ v\\
&\mathbf{do}\ \{s \leftarrow l_1.\mathit{mcreate}\ v; \mathit{return}\ (h\ s)\} &&= l_2.\mathit{mcreate}\ v
\end{aligned}$$

Similarly, given two M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ we say that $h :: S_1 \to S_2$ is a base map from sp_1 to sp_2 if h is a base map from $sp_1.\mathit{left}$ to $sp_2.\mathit{left}$ and from $sp_1.\mathit{right}$ to $sp_2.\mathit{right}$. ♢

Definition 4.6 (Bisimulation Equivalence). A *bisimulation* of M-lens spans $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M$ and $sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$ is a M-lens span $sp :: [A \leftsquigarrow R \rightsquigarrow B]_M$ where $R \subseteq S_1 \times S_2$ and fst is a base map from sp to sp_1 and snd is a base map from sp to sp_2. We write $sp_1 \equiv_{\mathrm{b}} sp_2$ when there is a bisimulation of spans sp_1 and sp_2. ♢

Figure 2 illustrates the three equivalences diagrammatically.

Proposition 4.7. Each of the relations $\equiv_{\mathrm{i}}$, $\equiv_{\mathrm{s}}$ and $\equiv_{\mathrm{b}}$ are equivalence relations on compatible spans of M-lenses and satisfy (Identity), (Assoc) and (Cong). ♢

Theorem 4.8. $sp_1 \equiv_{\mathrm{i}} sp_2$ implies $sp_1 \equiv_{\mathrm{s}} sp_2$, but not the converse. ♢

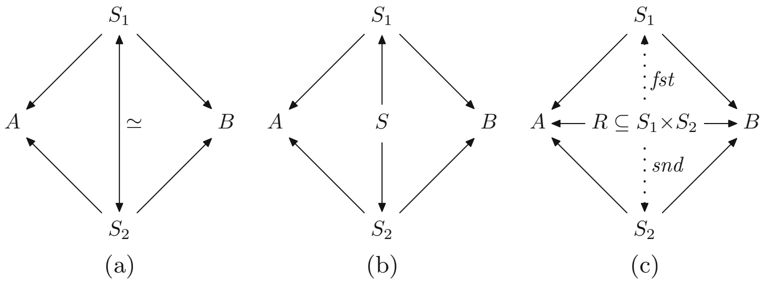


Fig. 2. (a) Isomorphism equivalence ($\equiv_\mathrm{i}$), (b) span equivalence ($\equiv_\mathrm{s}$), and (c) bisimulation ($\equiv_\mathrm{b}$) equivalence. In (c), the dotted arrows are base maps; all other arrows are (monadic) lenses.

Proof. The forward direction is obvious; for the reverse direction, consider

$h :: \mathit{Bool} \rightsquigarrow ()$
$h = \mathit{Lens}\ (\lambda\_ \to ())\ (\lambda a\ () \to a)\ (\lambda() \to \mathit{True})$
$sp_1 :: [() \leftsquigarrow () \rightsquigarrow ()]_\mu$
$sp_1 = \mathit{Span}\ \mathit{idMLens}\ \mathit{idMLens}$
$sp_2 = (h \mathbin{;} sp_1.\mathit{left}, h \mathbin{;} sp_2.\mathit{right})$

Clearly $sp_1 \equiv_\mathrm{s} sp_2$ by definition and all three structures are well-behaved, but h is not an isomorphism: any $k :: () \rightsquigarrow \mathit{Bool}$ must satisfy $k.\mathit{get}\ () = \mathit{True}$ or $k.\mathit{get}\ () = \mathit{False}$, so $(h \mathbin{;} k).\mathit{get} = k.\mathit{get} \cdot h.\mathit{get}$ cannot be the identity function. □

Theorem 4.9. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M, sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_\mathrm{s} sp_2$ then $sp_1 \equiv_\mathrm{b} sp_2$. ♢

Proof. For the forward direction, it suffices to show that a single $sp_1 \curvearrowright sp_2$ step implies $sp_1 \equiv_\mathrm{b} sp_2$, which is straightforward by taking R to be the set of pairs $\{(s_1, s_2) \mid l_1.\mathit{get}\ s_1 = s_2\}$, and constructing an appropriate span $sp : A \leftsquigarrow R \rightsquigarrow B$. Since bisimulation equivalence is transitive, it follows that $sp_1 \equiv_\mathrm{s} sp_2$ implies $sp_1 \equiv_\mathrm{b} sp_2$ as well. □

In the pure case, we can also show a converse:

Theorem 4.10. Given $sp_1 :: A \leftsquigarrow S_1 \rightsquigarrow B, sp_2 :: A \leftsquigarrow S_2 \rightsquigarrow B$, if $sp_1 \equiv_\mathrm{b} sp_2$ then $sp_1 \equiv_\mathrm{s} sp_2$. ♢

Proof. Given R and a span $sp :: A \leftsquigarrow R \rightsquigarrow B$ constituting a bisimulation $sp_1 \equiv_\mathrm{b} sp_2$, it suffices to construct a span $sp' = (l, r) :: S_1 \leftsquigarrow R \rightsquigarrow S_2$ satisfying $l \mathbin{;} sp_1.\mathit{left} = r \mathbin{;} sp_2.\mathit{left}$ and $l \mathbin{;} sp_1.\mathit{right} = r \mathbin{;} sp_2.\mathit{right}$. □

This result is surprising because the two equivalences come from rather different perspectives. Johnson and Rosebrugh introduced a form of span equivalence, and showed that it implies bisimulation equivalence. They did not explicitly address the question of whether this implication is strict. However, there are some differences between their presentation and ours; the most important difference is the fact that we assume lenses to be equipped with a create function, while they consider lenses without create functions but sometimes consider spans of lenses to be "pointed", or equipped with designated initial state values. Likewise, Abou-Saleh et al. (2015b) considered bisimulation equivalence for coalgebraic bx over pointed sets (i.e. sets equipped with designated initial values). It remains to be determined whether Theorem 4.10 transfers to these settings.

We leave it as an open question to determine whether $\equiv_{\mathrm{b}}$ is equivalent to $\equiv_{\mathrm{s}}$ for spans of monadic lenses (we conjecture that they are not), or whether an analogous result to Theorem 4.10 carries over to symmetric lenses (we conjecture that it does).

5 Conclusions

Lenses are a popular and powerful abstraction for bidirectional transformations. Although they are most often studied in their conventional, pure form, practical applications of lenses typically grapple with side-effects, including exceptions, state, and user interaction. Some recent proposals for extending lenses with monadic effects have been made; our proposal for (asymmetric) monadic lenses improves on them because M-lenses are closed under composition for any fixed monad M. Furthermore, we investigated the symmetric case, and showed that *spans* of monadic lenses are also closed under composition, while the obvious generalisation of pure symmetric lenses to incorporate monadic effects is not closed under composition. Finally, we presented three notions of equivalence for spans of monadic lenses, related them, and proved a new result: bisimulation and span equivalence coincide for pure spans of lenses. This last result is somewhat surprising, given that Johnson and Rosebrugh introduced (what we call) span equivalence to overcome perceived shortcomings in Hofmann et al.'s bisimulation-based notion of symmetric lens equivalence. Further investigation is necessary to determine whether this result generalises.

These results illustrate the benefits of our formulation of monadic lenses and we hope they will inspire further research and appreciation of bidirectional programming with effects.

Acknowledgements. The work was supported by the UK EPSRC-funded project *A Theory of Least Change for Bidirectional Transformations* (TLCBX Project 2013–2016) (EP/K020218/1, EP/K020919/1).

A Proofs for Sect. 2

Theorem 2.5. If $l_1 :: [A \rightsquigarrow B]_M$ and $l_2 :: [B \rightsquigarrow C]_M$ are well-behaved, then so is $l_1 \mathbin{;} l_2$.

Proof. Suppose l_1 and l_2 are well-behaved, and let $l = l_1 \mathbin{;} l_2$. We reason as follows for $(\mathsf{MGetPut})$:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, l.mput\ a\ (l.mget\ a)\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow l_2.mput\ (l_1.mget\ a)\ (l_2.mget\ (l_1.mget\ a)); l_1.mput\ a\ b\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow return\ (l_1.mget\ a); l_1.mput\ a\ b\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, l_1.mput\ a\ (l_1.mget\ a)\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& return\ a
\end{array}
$$

For (MPutGet), the proof is as follows:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, a' \leftarrow l.mput\ a\ c; return\ (a', l.mget\ a')\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', l_2.mget\ (l_1.mget\ a'))\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', l_2.mget\ b)\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\,\{\, b \leftarrow l_2.mput\ (l_1.mget\ a)\ c; \\
& \quad a' \leftarrow l_1.mput\ a\ b; \\
& \quad return\ (a', c)\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\,\{\, a' \leftarrow l.mput\ a\ c; return\ (a', c)\}
\end{array}
$$

□

B Proofs for Sect. 3

Proposition 3.6. *setBool* x is well-behaved for $x \in \{True, False\}$, but *setBool True* ; *setBool False* is not well-behaved. ♢

For the first part:

Proof. Let $sl = setBool\ x$. We consider (PutRLM), and (PutLRM) is symmetric.

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, (b, c') \leftarrow (setBool\ x).mput_{\mathrm{R}}\ ((), ()); (setBool\ x).mput_{\mathrm{L}}\ (b, c')\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\, (b, c') \leftarrow \mathbf{do}\,\{\, set\ x; return\ ((), ())\}; set\ x; return\ ((), c')\} \\
= & [\![\ \text{monad associativity}\ ]\!] \\
& \mathbf{do}\,\{\, set\ x; (b, c') \leftarrow return\ ((), ()); set\ x; return\ ((), c')\} \\
= & [\![\ \text{commutativity of } return\ ]\!] \\
& \mathbf{do}\,\{\, set\ x; set\ x; (b, c') \leftarrow return\ ((), ()); return\ ((), c')\} \\
= & [\![\ set\ x; set\ x = set\ x\ ]\!] \\
& \mathbf{do}\,\{\, set\ x; (b, c') \leftarrow return\ ((), ()); return\ ((), c')\}
\end{array}
$$

$$\begin{array}{l}
= \quad [\![\text{ monad associativity }]\!] \\
\quad \mathbf{do}\,\{(b, c') \leftarrow \mathbf{do}\,\{\mathit{set}\ x; \mathit{return}\ ((), ())\}; \mathit{return}\ ((), c')\} \\
= \quad [\![\text{ Definition }]\!] \\
\quad \mathbf{do}\,\{(b, c') \leftarrow (\mathit{setBool}\ x).\mathit{mput}_{\mathrm{R}}\ ((), ()); \mathit{return}\ ((), c')\}
\end{array}$$

For the second part, taking $sl = \mathit{setBool}\ \mathit{True}\ ; \mathit{setBool}\ \mathit{False}$, we proceed as follows:

$$\begin{array}{l}
\quad \mathbf{do}\,\{(c, s') \leftarrow sl.\mathit{mput}_{\mathrm{R}}\ (a, s); sl.\mathit{mput}_{\mathrm{L}}\ (c, s')\} \\
= \quad [\![\text{ let } s = (s_1, s_2) \text{ and } s' = (s_1''', s_2'''); \text{ definition }]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad (c', (s_1'', s_2'')) \leftarrow \mathit{return}\ (c, (s_1', s_2')); \\
\qquad\quad (b', s_2''') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{L}}\ (c', s_2''); \\
\qquad\quad (a', s_2''') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{L}}\ (b', s_1''); \\
\qquad\quad \mathit{return}\ (c, (s_1''', s_2'''))\} \\
= \quad [\![\text{ monad unit }]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad (b', s_2''') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{L}}\ (c', s_2'); \\
\qquad\quad (a', s_2''') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{L}}\ (b', s_1'); \\
\qquad\quad \mathit{return}\ (c, (s_1''', s_2'''))\} \\
= \quad [\![\ (\mathsf{PutRLM}) \text{ for } \mathit{setBool}\ \mathit{False} \ ]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad (b', s_2''') \leftarrow \mathit{return}\ (b, s_2'); \\
\qquad\quad (a', s_2''') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{L}}\ (b', s_1'); \\
\qquad\quad \mathit{return}\ (c, (s_1''', s_2'''))\} \\
= \quad [\![\text{ monad unit }]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad (a', s_2''') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{L}}\ (b, s_1'); \\
\qquad\quad \mathit{return}\ (c, (s_1''', s_2'))\}
\end{array}$$

However, we cannot simplify this any further. Moreover, it should be clear that the shared state will be *True* after this operation is performed. Considering the other side of the desired equation:

$$\begin{array}{l}
\quad \mathbf{do}\,\{(c, s') \leftarrow sl.\mathit{mput}_{\mathrm{R}}\ (a, s); sl.\mathit{mput}_{\mathrm{L}}\ (c, s'')\} \\
= \quad [\![\text{ let } s = (s_1, s_2) \text{ and } s' = (s_1''', s_2'''); \text{ Definition }]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad (c', (s_1'', s_2'')) \leftarrow \mathit{return}\ (c, (s_1', s_2')); \\
\qquad\quad \mathit{return}\ (c', (s_1'', s_2''))\} \\
= \quad [\![\text{ Monad unit }]\!] \\
\quad \mathbf{do}\,\{(b, s_1') \leftarrow (\mathit{setBool}\ \mathit{True}).\mathit{mput}_{\mathrm{R}}\ (a, s_1); \\
\qquad\quad (c, s_2') \leftarrow (\mathit{setBool}\ \mathit{False}).\mathit{mput}_{\mathrm{R}}\ (b, s_2); \\
\qquad\quad \mathit{return}\ (c, (s_1', s_2'))\}
\end{array}$$

it should be clear that the shared state will be *False* after this operation is performed. Therefore, (PutRLM) is not satisfied by *sl*. □

Lemma 3.8. If $ml_1 :: [\sigma_1 \rightsquigarrow \beta]_\mu$ and $ml_2 :: [\sigma_2 \rightsquigarrow \beta]_\mu$ are well-behaved then so is $ml_1 \bowtie ml_2 :: [\sigma_1 \leftrightsquigarrow (\sigma_1 \bowtie \sigma_2) \rightsquigarrow \sigma_2]_\mu$. ♢

Proof. It suffices to consider the two lenses $l_1 = \mathit{MLens}\ \mathit{fst}\ \mathit{put}_{\mathrm{L}}\ \mathit{create}_{\mathrm{L}}$ and $l_2 = \mathit{MLens}\ \mathit{snd}\ \mathit{put}_{\mathrm{R}}\ \mathit{create}_{\mathrm{R}}$ in isolation. Moreover, the two cases are completely symmetric, so we only show the first.

For (MGetPut), we show:

$$\begin{aligned}
&\mathbf{do}\,\{l_1.\mathit{mput}\ (s_1,s_2)\ (l_1.\mathit{mget}\ (s_1,s_2))\}\\
=\ &[\![\ \text{definition}\ ]\!]\\
&\mathbf{do}\,\{\mathit{put}_{\mathrm{L}}\ (s_1,s_2)\ (\mathit{fst}\ (s_1,s_2))\}\\
=\ &[\![\ \text{definition of } \mathit{put}_{\mathrm{L}} \text{ and } \mathit{fst}\ ]\!]\\
&\mathbf{do}\,\{s_2' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ s_1)\}\\
=\ &[\![\ (s_1,s_2) \text{ consistent}\ ]\!]\\
&\mathbf{do}\,\{s_2' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_2.\mathit{mget}\ s_2)\}\\
=\ &[\![\ (\mathsf{MGetPut})\ ]\!]\\
&\mathit{return}\ s
\end{aligned}$$

The proof for (MPutGet) goes as follows. Note that it holds by construction, without appealing to well-behavedness of ml_1 or ml_2.

$$\begin{aligned}
&\mathbf{do}\,\{(s_1',s_2') \leftarrow l_1.\mathit{mput}\ (s_1,s_2)\ a; \mathit{return}\ ((s_1',s_2'), l_1.\mathit{mget}\ (s_1',s_2'))\}\\
=\ &[\![\ \text{definition}\ ]\!]\\
&\mathbf{do}\,\{(s_1',s_2') \leftarrow \mathit{put}_{\mathrm{R}}\ (s_1,s_2)\ a; \mathit{return}\ ((s_1',s_2'), \mathit{fst}\ (s_1',s_2'))\}\\
=\ &[\![\ \text{definition}\ ]\!]\\
&\mathbf{do}\,\{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1',s_2') \leftarrow \mathit{return}\ (a,s_2'');\\
&\qquad \mathit{return}\ ((s_1',s_2'), \mathit{fst}\ (s_1',s_2'))\}\\
=\ &[\![\ \text{definition of } \mathit{fst}\ ]\!]\\
&\mathbf{do}\,\{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1',s_2') \leftarrow \mathit{return}\ (a,s_2'');\\
&\qquad \mathit{return}\ ((s_1',s_2'), s_1')\\
=\ &[\![\ \text{monad laws}\ ]\!]\\
&\mathbf{do}\,\{s_2'' \leftarrow ml_2.\mathit{mput}\ s_2\ (ml_1.\mathit{mget}\ a); (s_1',s_2') \leftarrow \mathit{return}\ (a,s_2'');\\
&\qquad \mathit{return}\ ((s_1',s_2'), a)\}\\
=\ &[\![\ \text{definition}\ ]\!]\\
&\mathbf{do}\,\{(s_1',s_2') \leftarrow \mathit{put}_{\mathrm{L}}\ (s_1,s_2)\ a; \mathit{return}\ ((s_1',s_2'), a)\}\\
=\ &[\![\ \text{definition}\ ]\!]\\
&\mathbf{do}\,\{(s_1',s_2') \leftarrow l_1.\mathit{mput}\ (s_1,s_2)\ a; \mathit{return}\ ((s_1',s_2'), a)\}
\end{aligned}$$

The proof for (MCreateGet) is similar.

Finally, we show that $\mathit{put}_{\mathrm{L}} :: (\sigma_1 \bowtie \sigma_2) \to \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$, and in particular, that it maintains the consistency invariant on the state space $\sigma_1 \bowtie \sigma_2$. Assume that $(s_1,s_2) :: \sigma_1 \bowtie \sigma_2$ and $s_1' :: \sigma_1$ are given. Thus, $ml_1.\mathit{mget}\ s_1 = ml_2.\mathit{mget}\ s_2$. We must show that any value returned by $\mathit{put}_{\mathrm{L}}$ also satisfies this consistency criterion. By definition,

$$put_{\mathrm{L}}\ (s_1, s_2)\ s'_1 = \mathbf{do}\ \{ s'_2 \leftarrow ml_2.mput\ s_2\ (ml_1.mget\ s'_1); return\ (s'_1, s'_2) \}$$

By (MPutGet), any s'_2 resulting from $ml_2.mput\ s_2\ (ml_1.mget\ s'_1)$ will satisfy $ml_2.mget\ s'_2 = ml_1.mget\ s'_1$. The proof that $create_{\mathrm{L}} :: \sigma_1 \to \mu\ (\sigma_1 \bowtie \sigma_2)$ is similar, but simpler. □

Theorem 3.10. If $sp :: [A \leftrightsquigarrow S \rightsquigarrow B]_M$ is well-behaved, then *span2smlens sp* is also well-behaved. ♢

Proof. Let $sl = span2smlens\ sp$. We need to show that the laws (PutRLM) and (PutLRM) hold. We show (PutRLM), and (PutLRM) is symmetric.

We need to show that

$$\begin{aligned}
&\mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_{\mathrm{R}}\ (a, mc); sl.mput_{\mathrm{L}}\ (b', mc') \} \\
=\ & \\
&\mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_{\mathrm{R}}\ (a, mc); return\ (a, mc') \}
\end{aligned}$$

There are two cases, depending on whether the initial state mc is *Nothing* or *Just c* for some c.

If $mc = Nothing$ then we reason as follows:

$$\begin{aligned}
&\mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_{\mathrm{R}}\ (a, Nothing); sl.mput_{\mathrm{L}}\ (b', mc') \} \\
=\ & [\![\text{ Definition }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; (b', mc') \leftarrow return\ (sp.right.mget\ s', Just\ s'); \\
&\qquad sl.mput_{\mathrm{L}}\ (b', mc') \} \\
=\ & [\![\text{ monad unit }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; sl.mput_{\mathrm{L}}\ (sp.right.mget\ s', Just\ s') \} \\
=\ & [\![\text{ definition }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; s'' \leftarrow sp.right.mput\ s'\ (sp.right.mget\ s'); \\
&\qquad return\ (sp.left.mget\ s'', Just\ s'') \} \\
=\ & [\![\text{ (MGetPut) }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; s'' \leftarrow return\ s'; \\
&\qquad return\ (sp.left.mget\ s'', Just\ s'') \} \\
=\ & [\![\text{ monad unit }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; return\ (sp.left.mget\ s', Just\ s') \} \\
=\ & [\![\text{ (MCreateGet) }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; return\ (a, Just\ s') \} \\
=\ & [\![\text{ monad unit }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mcreate\ a; (b', mc') \leftarrow (sp.right.get\ s', Just\ s'); \\
&\qquad return\ (a, mc') \} \\
=\ & [\![\text{ Definition }]\!] \\
&\mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_{\mathrm{R}}\ (a, Nothing); return\ (a, mc') \}
\end{aligned}$$

If $mc = Just\ c$ then we reason as follows:

$$\begin{aligned}
&\mathbf{do}\ \{ (b', mc') \leftarrow sl.mput_{\mathrm{R}}\ (a, Just\ s); sl.mput_{\mathrm{L}}\ (b', mc') \} \\
=\ & [\![\text{ Definition }]\!] \\
&\mathbf{do}\ \{ s' \leftarrow sp.left.mput\ s\ a; (b', mc') \leftarrow (sp.right.mget\ s', Just\ s');
\end{aligned}$$

$$\begin{array}{ll}
& \quad \mathit{sl.mput}_{\mathrm{L}}\ (b', mc')\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{sl.mput}_{\mathrm{L}}\ (\mathit{sp.right.mget}\ s', \mathit{Just}\ s')\} \\
= & [\![\ \text{definition}\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; s'' \leftarrow \mathit{sp.right.mput}\ s'\ (\mathit{sp.right.mget}\ s'); \\
& \quad \mathit{return}\ (\mathit{sp.left.mget}\ s'', \mathit{Just}\ s'')\} \\
= & [\![\ (\mathsf{MGetPut})\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; s'' \leftarrow \mathit{return}\ s'; \\
& \quad \mathit{return}\ (\mathit{sp.left.mget}\ s'', \mathit{Just}\ s'')\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{return}\ (\mathit{sp.left.mget}\ s', \mathit{Just}\ s')\} \\
= & [\![\ (\mathsf{MPutGet})\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; \mathit{return}\ (a, \mathit{Just}\ s')\} \\
= & [\![\ \text{monad unit}\ ]\!] \\
& \mathbf{do}\ \{s' \leftarrow \mathit{sp.left.mput}\ s\ a; (b', mc') \leftarrow (\mathit{sp.right.get}\ s', \mathit{Just}\ s'); \\
& \quad \mathit{return}\ (a, mc')\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{(b', mc') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a, \mathit{Just}\ c); \mathit{return}\ (a, mc')\}
\end{array}$$

□

Theorem 3.12. If $sl :: \mathit{SMLens}\ \mathit{Id}\ C\ A\ B$ is well-behaved, then *smlens2span sl* is also well-behaved, with state space S consisting of the consistent triples of *sl*. ♢

Proof. First we show that, given a symmetric lens *sl*, the operations of $sp = \mathit{smlens2span}\ sl$ preserve consistency of the state. Assume (a, b, c) is consistent. To show that $\mathit{sp.left.mput}\ (a, b, c)\ a'$ is consistent for any a', we have to show that (a', b', c') is consistent, where a' is arbitrary and $\mathit{return}\ (b', c') = \mathit{sl.mput}_{\mathrm{R}}\ (a', c)$. For one half of consistency, we have:

$$\begin{array}{ll}
& \mathit{sl.mput}_{\mathrm{L}}\ (b', c') \\
= & [\![\ \mathit{sl.mput}_{\mathrm{R}}\ (a', c) = \mathit{return}\ (b', c'), \text{ and } (\mathsf{PutRLM})\ ]\!] \\
& \mathit{return}\ (a', c')
\end{array}$$

The proof that $\mathit{sl.mput}_{\mathrm{R}}\ (a', c') = \mathit{return}\ (b', c')$ is symmetric.

$$\begin{array}{ll}
& \mathit{sl.mput}_{\mathrm{R}}\ (a', c') \\
= & [\![\ \text{above, and } (\mathsf{PutLRM})\ ]\!] \\
& \mathit{return}\ (b', c')
\end{array}$$

as required. The proof that $\mathit{sp.right.mput}\ (a, b, c)\ b'$ is consistent is dual.

We will now show that $sp = \mathit{smlens2span}\ sl$ is a well-behaved span for any symmetric lens *sl*. For (MGetPut), we proceed as follows:

$$\begin{array}{ll}
& \mathit{sp.left.mput}\ (a, b, c)\ (\mathit{sp.left.mget}\ (a, b, c)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{(b', c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a, c); \mathit{return}\ (a, b', c')\} \\
= & [\![\ \text{Consistency of } (a, b, c)\ ]\!]
\end{array}$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{return}\ (b,c); \mathit{return}\ (a,b',c')\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathit{return}\ (a,b,c)
\end{array}
$$

For (MPutGet), we have:

$$
\begin{array}{ll}
& \mathbf{do}\,\{s' \leftarrow \mathit{sp.left.put}\ (a,b,c)\ a'; \mathit{return}\ (s', \mathit{sp.left.mget}\ s')\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \\
& \quad \mathit{return}\ (s', \mathit{sp.left.mget}\ s')\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a',c); \\
& \quad \mathit{return}\ ((a',b',c'), \mathit{sp.left.mget}\ (a',b',c'))\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a',c); \mathit{return}\ ((a',b',c'), a')\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a',c); s' \leftarrow \mathit{return}\ (a',b',c'); \mathit{return}\ (s',a')\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{s' \leftarrow \mathit{sp.left.put}\ (a,b,c)\ a'; \mathit{return}\ (s',a')\}
\end{array}
$$

The proof for (MCreateGet) is similar.

$$
\begin{array}{ll}
& \mathbf{do}\,\{s \leftarrow \mathit{sp.left.create}\ a; \mathit{return}\ (s, \mathit{sp.left.mget}\ s)\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a,c); s \leftarrow \mathit{return}\ (a,b',c'); \\
& \quad \mathit{return}\ (s, \mathit{sp.left.mget}\ s)\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a,c); \\
& \quad \mathit{return}\ ((a,b',c'), \mathit{sp.left.mget}\ (a,b',c'))\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a,c); \mathit{return}\ ((a,b',c'), a)\} \\
= & [\![\text{ monad unit }]\!] \\
& \mathbf{do}\,\{(b',c') \leftarrow \mathit{sl.mput}_{\mathrm{R}}\ (a,c); s \leftarrow \mathit{return}\ (a,b',c'); \mathit{return}\ (s,a)\} \\
= & [\![\text{ Definition }]\!] \\
& \mathbf{do}\,\{s \leftarrow \mathit{sp.left.create}\ a; \mathit{return}\ (a,s)\}
\end{array}
$$

□

C Proofs for Sect. 4

Lemma 4.3. Suppose $l_1 :: A \leadsto B$ and $l_2 :: C \leadsto B$ are pure lenses. Then $(l_1 \bowtie l_2).\mathit{left} \mathbin{;} l_1 = (l_1 \bowtie l_2).\mathit{right} \mathbin{;} l_2$. ♢

Proof. Let $(l, r) = l_1 \bowtie l_2$. We show that each component of $l \mathbin{;} l_1$ equals the corresponding component of $r \mathbin{;} l_2$.

For *get*:

$$
\begin{array}{ll}
& (l \mathbin{;} l_1).\mathit{get}\ (a,c) \\
= & [\![\text{ Definition }]\!]
\end{array}
$$

$$\begin{array}{ll}
& l_1.get\ (l.get\ (a,c)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_1.get\ a \\
= & [\![\ \text{Consistency}\ ]\!] \\
& l_2.get\ c \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_2.get\ (r.get\ (a,c)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& (r\ ;\ l_2).get\ (a,c)
\end{array}$$

For *put*:

$$\begin{array}{ll}
& (l\ ;\ l_1).put\ (a,c)\ b \\
= & [\![\ \text{Definition}\ ]\!] \\
& l.put\ (a,c)\ (l_1.put\ (l.get\ (a,c))\ b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l.put\ (a,c)\ (l_1.put\ a\ b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{let}\ a' = l_1.put\ a\ b\ \mathbf{in} \\
& \mathbf{let}\ c' = l_2.put\ c\ (l_1.get\ a')\ \mathbf{in}\ (a',c') \\
= & [\![\ \text{inline } \mathbf{let}\ ]\!] \\
& (l_1.put\ a\ b, l_2.put\ c\ (l_1.get\ (l_1.put\ a\ b))) \\
= & [\![\ (\mathsf{PutGet})\ ]\!] \\
& (l_1.put\ a\ b, l_2.put\ c\ b) \\
= & [\![\ \text{reverse above steps}\ ]\!] \\
& (r\ ;\ l_2).put\ (a,c)\ b
\end{array}$$

Finally, for *create*:

$$\begin{array}{ll}
& (l\ ;\ l_1).create\ b \\
= & [\![\ \text{Definition}\ ]\!] \\
& l.create\ (l_1.create\ b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{let}\ c = l_2.create\ (l_1.get\ (l_1.create\ b))\ \mathbf{in}\ (l_1.create\ b, c) \\
= & [\![\ (\mathsf{CreateGet})\ ]\!] \\
& \mathbf{let}\ c = l_2.create\ b\ \mathbf{in}\ (l_1.create\ b, c) \\
= & [\![\ \text{Inline } \mathbf{let}\ ]\!] \\
& (l_1.create\ b, l_2.create\ b) \\
= & [\![\ \text{reverse above steps}\ ]\!] \\
& (r\ ;\ l_2).create\ b
\end{array}$$

□

Theorem 4.9. Given $sp_1 :: [A \leftsquigarrow S_1 \rightsquigarrow B]_M, sp_2 :: [A \leftsquigarrow S_2 \rightsquigarrow B]_M$, if $sp_1 \equiv_{\mathrm{s}} sp_2$ then $sp_1 \equiv_{\mathrm{b}} sp_2$. ♢

Proof. We give the details for the case $sp_1 \curvearrowright sp_2$. First, write $(l_1, r_1) = sp_1$ and $(l_2, r_2) = sp_2$, and suppose $l :: S_1 \rightsquigarrow S_2$ is a lens satisfying $l_1 = l\,;l_2$ and $r_1 = l\,;r_2$.

We need to define a bisimulation consisting of a set $R \subseteq S_1 \times S_2$ and a span $sp = (l_0, r_0) :: [A \leftsquigarrow R \rightsquigarrow B]_M$ such that *fst* is a base map from *sp* to sp_1 and *snd* is a base map from *sp* to sp_2. We take $R = \{(s_1, s_2) \mid s_2 = l.get\ (s_1)\}$ and proceed as follows:

$$\begin{array}{ll}
l_0 & :: [R \rightsquigarrow A]_M \\
l_0.mget\ (s_1, s_2) & = l_1.mget\ s_1 \\
l_0.mput\ (s_1, s_2)\ a & = \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ a; return\ (s_1', l.get\ s_1')\} \\
l_0.mcreate\ a & = \mathbf{do}\ \{s_1 \leftarrow l_1.mcreate\ a; return\ (s_1, l.get\ s_1)\} \\
r_0 & :: [R \rightsquigarrow B]_M \\
r_0.mget\ (s_1, s_2) & = r_1.mget\ s_1 \\
r_0.mput\ (s_1, s_2)\ b & = \mathbf{do}\ \{s_1' \leftarrow r_1.mput\ s_1\ b; return\ (s_1', l.get\ s_1')\} \\
r_0.mcreate\ b & = \mathbf{do}\ \{s_1 \leftarrow r_1.mcreate\ a; return\ (s_1, l.get\ s_1)\}
\end{array}$$

We must now show that l_0 and r_0 are well-behaved (full) lenses, and that the projections *fst* and *snd* map $sp = (l_0, r_0)$ to sp_1 and sp_2 respectively.

We first show that l_0 is well-behaved; the reasoning for r_0 is symmetric. For (MGetPut) we have:

$$\begin{array}{ll}
& l_0.mput\ (s_1, s_2)\ (l_0.mget\ (s_1, s_2)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ (l_1.mget\ s_1); return\ (s_1', l.get\ s_1')\} \\
= & [\![\ \text{(MPutGet)}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow return\ s_1; return\ (s_1', l.get\ s_1')\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& return\ (s_1, l.get\ s_1) \\
= & [\![\ s_2 = l.get\ s_1\ ]\!] \\
& return\ (s_1, s_2)
\end{array}$$

For (MPutGet) we have:

$$\begin{array}{ll}
& \mathbf{do}\ \{(s_1'', s_2'') \leftarrow l_0.mput\ (s_1, s_2)\ a; return\ ((s_1'', s_2''), l_0.mget\ (s_1'', s_2''))\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ a; (s_1'', s_2'') \leftarrow return\ (s_1', l.get\ s_1'); \\
& \qquad return\ ((s_1'', s_2''), l_1.mget\ s_1')\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ a; return\ ((s_1', l.get\ s_1'), l_1.mget\ s_1')\} \\
= & [\![\ \text{(MPutGet)}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ a; return\ ((s_1', l.get\ s_1'), a)\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\ \{s_1' \leftarrow l_1.mput\ s_1\ a; (s_1'', s_2'') \leftarrow return\ (s_1', l.get\ s_1'); \\
& \qquad return\ ((s_1'', s_2''), a)\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\ \{(s_1'', s_2'') \leftarrow l_0.mput\ (s_1, s_2)\ a; return\ ((s_1'', s_2''), a)\}
\end{array}$$

Finally, for (MCreateGet) we have:

$$\begin{array}{ll}
& \mathbf{do}\ \{(s_1, s_2) \leftarrow l_0.mcreate\ a; return\ ((s_1, s_2), l_0.mget\ (s_1, s_2))\} \\
= & [\![\ \text{Definition}\ ]\!]
\end{array}$$

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, s_1' \leftarrow l_1.\mathit{mcreate}\ a; (s_1, s_2) \leftarrow \mathit{return}\ (s_1', l.\mathit{get}\ s_1'); \\
& \quad \mathit{return}\ ((s_1, s_2), l_1.\mathit{mget}\ s_1)\,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, s_1' \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s_1', l.\mathit{get}\ s_1'), l_1.\mathit{mget}\ s_1')\,\} \\
= & [\![\ (\mathsf{MCreateGet})\ ]\!] \\
& \mathbf{do}\,\{\, s_1' \leftarrow l_1.\mathit{mcreate}\ a; \mathit{return}\ ((s_1', l.\mathit{get}\ s_1'), a)\,\} \\
= & [\![\ \text{Monad unit}\ ]\!] \\
& \mathbf{do}\,\{\, s_1' \leftarrow l_1.\mathit{mcreate}\ a; (s_1, s_2) \leftarrow \mathit{return}\ (s_1', l.\mathit{get}\ s_1'); \\
& \quad \mathit{return}\ ((s_1, s_2), a)\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\, (s_1, s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ ((s_1, s_2), a)\,\}
\end{array}
$$

Next, we show that *fst* is a base map from l_0 to l_1 and *snd* is a base map from l_0 to l_2. It is easy to show that *fst* is a base map from l_0 to l_1 by unfolding definitions and applying of monad laws. To show that *snd* is a base map from l_0 to l_2, we need to verify the following three equations that show that *snd* commutes with *mget*, *mput* and *mcreate*:

$$
\begin{array}{ll}
l_0.\mathit{mget}\ (s_1, s_2) & = l_2.\mathit{mget}\ s_2 \\
\mathbf{do}\,\{\, (s_1', s_2') \leftarrow l_0.\mathit{mput}\ (s_1, s_2)\ a; \mathit{return}\ s_2'\,\} & = l_2.\mathit{mput}\ s_2\ a \\
\mathbf{do}\,\{\, (s_1, s_2) \leftarrow l_0.\mathit{mcreate}\ a; \mathit{return}\ s_2\,\} & = l_2.\mathit{mcreate}\ a
\end{array}
$$

For the *mget* equation:

$$
\begin{array}{ll}
& l_0.\mathit{mget}\ (s_1, s_2) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_1.\mathit{mget}\ s_1 \\
= & [\![\ \text{Assumption}\ l; l_2 = l_1\ ]\!] \\
& (l\ ;\ l_2).\mathit{mget}\ s_1 \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_2.\mathit{mget}\ (l.\mathit{get}\ s_1) \\
= & [\![\ (s_1, s_2) \in R\ ]\!] \\
& l_2.\mathit{mget}\ s_2
\end{array}
$$

For the *mput* equation:

$$
\begin{array}{ll}
& \mathbf{do}\,\{\, (s_1', s_2') \leftarrow l_0.\mathit{mput}\ (s_1, s_2)\ a; \mathit{return}\ s_2'\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\, s_1'' \leftarrow l_1.\mathit{mput}\ s_1\ a; (s_1', s_2') \leftarrow \mathit{return}\ (s_1'', l.\mathit{get}\ s_1''); \mathit{return}\ s_2'\,\} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& \mathbf{do}\,\{\, s_1'' \leftarrow l_1.\mathit{mput}\ s_1\ a; \mathit{return}\ (l.\mathit{get}\ s_1'')\,\} \\
= & [\![\ l\ ;\ l_2 = l_1\ ]\!] \\
& \mathbf{do}\,\{\, s_1'' \leftarrow (l\ ;\ l_2).\mathit{mput}\ s_1\ a; \mathit{return}\ (l.\mathit{get}\ s_1'')\,\} \\
= & [\![\ \text{Definition}\ ]\!] \\
& \mathbf{do}\,\{\, s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; s_1'' \leftarrow \mathit{return}\ (l.\mathit{put}\ s_1\ s_2''); \mathit{return}\ (l.\mathit{get}\ s_1'')\,\} \\
= & [\![\ \text{Monad laws}\ ]\!] \\
& \mathbf{do}\,\{\, s_2'' \leftarrow l_2.\mathit{mput}\ (l.\mathit{get}\ s_1)\ a; \mathit{return}\ (l.\mathit{get}\ (l.\mathit{put}\ s_1\ s_2''))\,\} \\
= & [\![\ (\mathsf{PutGet})\ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\quad \mathbf{do}\,\{ s_2'' \leftarrow l_2.mput\ (l.get\ s_1)\ a; return\ s_2'' \} \\
= \quad [\![\ (s_1,s_2) \in R \text{ so } l.get\ s_1 = s_2 \ ]\!] \\
\quad \mathbf{do}\,\{ s_2'' \leftarrow l_2.mput\ s_2\ a; return\ s_2'' \} \\
= \quad [\![\ \text{Monad laws} \ ]\!] \\
\quad l_2.mput\ s_2\ a
\end{array}
$$

For the *mcreate* equation:

$$
\begin{array}{l}
\quad \mathbf{do}\,\{ (s_1,s_2) \leftarrow l_0.mcreate\ a; return\ s_2 \} \\
= \quad [\![\ \text{Definition} \ ]\!] \\
\quad \mathbf{do}\,\{ s_1' \leftarrow l_1.mcreate\ a; (s_1,s_2) \leftarrow return\ (s_1', l.get\ s_1'); return\ s_2 \} \\
= \quad [\![\ \text{Monad laws} \ ]\!] \\
\quad \mathbf{do}\,\{ s_1' \leftarrow l_1.mcreate\ a; return\ (l.get\ s_1') \} \\
= \quad [\![\ l \,;\, l_2 = l_1 \ ]\!] \\
\quad \mathbf{do}\,\{ s_1' \leftarrow (l \,;\, l_2).mcreate\ a; return\ (l.get\ s_1') \} \\
= \quad [\![\ \text{Definition} \ ]\!] \\
\quad \mathbf{do}\,\{ s_2' \leftarrow l_2.mcreate\ a; s_1' \leftarrow return\ (l.create\ s_2'); return\ (l.get\ s_1') \} \\
= \quad [\![\ \text{Monad laws} \ ]\!] \\
\quad \mathbf{do}\,\{ s_2' \leftarrow l_2.mcreate\ a; return\ (l.get\ (l.create\ s_2')) \} \\
= \quad [\![\ (\mathsf{CreateGet}) \ ]\!] \\
\quad \mathbf{do}\,\{ s_2' \leftarrow l_2.mcreate\ a; return\ s_2' \} \\
= \quad [\![\ \text{Monad laws} \ ]\!] \\
\quad l_2.mcreate\ a
\end{array}
$$

Similar reasoning suffices to show that *fst* is a base map from r_0 to r_1 and *snd* is a base map from r_0 to r_2, so we can conclude that R and (l, r) constitute a bisimulation between sp_1 and sp_2, that is, $sp_1 \equiv_{\mathrm{b}} sp_2$. □

Theorem 4.10. Given $sp_1 :: A \leftarrow\!\!\!\!\sim S_1 \rightsquigarrow B, sp_2 :: A \leftarrow\!\!\!\!\sim S_2 \rightsquigarrow B$, if $sp_1 \equiv_{\mathrm{b}} sp_2$ then $sp_1 \equiv_{\mathrm{s}} sp_2$. ♢

Proof. For convenience, we again write $sp_1 = (l_1, r_1)$ and $sp_2 = (l_2, r_2)$. We are given R and a span $sp_0 :: A \leftarrow\!\!\!\!\sim R \rightsquigarrow B$ constituting a bisimulation $sp_1 \equiv_{\mathrm{b}} sp_2$. Let $sp_0 = (l_0, r_0)$. For later reference, we list the properties that must hold by virtue of this bisimulation for any $(s_1, s_2) \in R$:

$$
\begin{array}{llll}
l_0.get\ (s_1,s_2) & = l_1.get\ s_1 & l_0.get\ (s_1,s_2) & = l_2.get\ s_2 \\
fst\ (l_0.put\ (s_1,s_2)\ a) & = l_1.put\ s_1\ a & snd\ (l_0.put\ (s_1,s_2)\ a) & = l_2.put\ s_2\ a \\
fst\ (l_0.create\ a) & = l_1.create\ s_1 & snd\ (l_0.create\ a) & = l_2.create\ a \\[1ex]
r_0.get\ (s_1,s_2) & = r_1.get\ s_1 & r_0.get\ (s_1,s_2) & = r_2.get\ s_2 \\
fst\ (r_0.put\ (s_1,s_2)\ b) & = r_1.put\ s_1\ b & snd\ (r_0.put\ (s_1,s_2)\ b) & = r_2.put\ s_2\ b \\
fst\ (r_0.create\ b) & = r_1.create\ s_1 & snd\ (r_0.create\ b) & = r_2.create\ b
\end{array}
$$

In addition, it follows that:

$$
\begin{array}{l}
l_0.put\ (s_1,s_2)\ a = (l_1.put\ s_1\ a, l_2.put\ s_2\ a) \in R \\
r_0.put\ (s_1,s_2)\ b = (r_1.put\ s_1\ b, r_2.put\ s_2\ b) \in R
\end{array}
$$

$$\begin{array}{l} l_0.\mathit{create}\ a = (l_1.\mathit{create}\ a, l_2.\mathit{create}\ a) \in R \\ r_0.\mathit{create}\ b = (r_1.\mathit{create}\ b, r_2.\mathit{create}\ b) \in R \end{array}$$

which also implies the following identities, which we call *twists*:

$$\begin{array}{l} r_1.\mathit{get}\ (l_1.\mathit{put}\ s_1\ a) = r_0.\mathit{get}\ (l_1.\mathit{put}\ s_1\ a, l_2.\mathit{put}\ s_2\ a) = r_2.\mathit{get}\ (l_2.\mathit{put}\ s_2\ a) \\ l_1.\mathit{get}\ (r_1.\mathit{put}\ s_1\ b) = l_0.\mathit{get}\ (r_1.\mathit{put}\ s_1\ b, r_2.\mathit{put}\ s_2\ b) = l_2.\mathit{get}\ (r_2.\mathit{put}\ s_2\ b) \\ r_1.\mathit{get}\ (l_1.\mathit{create}\ a) = r_0.\mathit{get}\ (l_1.\mathit{create}\ a, l_2.\mathit{create}\ a) = r_2.\mathit{get}\ (l_2.\mathit{create}\ a) \\ l_1.\mathit{get}\ (r_1.\mathit{create}\ b) = l_0.\mathit{get}\ (r_1.\mathit{create}\ b, r_2.\mathit{create}\ b) = l_2.\mathit{get}\ (r_2.\mathit{create}\ b) \end{array}$$

It suffices to construct a span $sp = (l, r) :: S_1 \leftsquigarrow R \rightsquigarrow S_2$ satisfying $l \mathbin{;} l_1 = r \mathbin{;} l_2$ and $l \mathbin{;} r_1 = r \mathbin{;} r_2$. Define l and r as follows:

$$\begin{array}{ll} l.\mathit{get} & = \mathit{fst} \\ l.\mathit{put}\ (s_1, s_2)\ s_1' & = l_0.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{get}\ s_1') \\ l.\mathit{create}\ s_1 & = l_0.\mathit{create}\ (l_1.\mathit{get}\ s_1) \\ r.\mathit{get} & = \mathit{snd} \\ r.\mathit{put}\ (s_1, s_2)\ s_2' & = l_0.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{get}\ s_2') \\ r.\mathit{create}\ s_2 & = l_0.\mathit{create}\ (l_2.\mathit{get}\ s_2) \end{array}$$

Notice that by construction $l :: R \rightsquigarrow S_1$ and $r :: R \rightsquigarrow S_2$, that is, since we have used l_0 and r_0 to define l and r, we do not need to do any more work to check that the pairs produced by *create* and *put* remain in R. Notice also that l and r only use the lenses l_1 and l_2, not r_1 and r_2; we will show nevertheless that they satisfy the required properties.

First, to show that $l \mathbin{;} l_1 = r \mathbin{;} l_2$, we proceed as follows for each operation. For *get*:

$$\begin{array}{ll} & (l \mathbin{;} l_1).\mathit{get}\ (s_1, s_2) \\ = & [\![\ \text{definition}\ ]\!] \\ & l_1.\mathit{get}\ (l.\mathit{get}\ (s_1, s_2)) \\ = & [\![\ \text{definition of } l.\mathit{get} = \mathit{fst}, \mathit{fst} \text{ commutes with } \mathit{get}\ ]\!] \\ & l_0.\mathit{get}\ (s_1, s_2) \\ = & [\![\ \text{reverse reasoning}\ ]\!] \\ & (r \mathbin{;} l_2).\mathit{get}\ (s_1, s_2) \end{array}$$

For *put*, we have:

$$\begin{array}{ll} & (l \mathbin{;} l_1).\mathit{put}\ (s_1, s_2)\ a \\ = & [\![\ \text{Definition}\ ]\!] \\ & l.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{put}\ s_1\ a) \\ = & [\![\ \text{Definition}\ ]\!] \\ & l_0.\mathit{put}\ (s_1, s_2)\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ s_1\ a)) \\ = & [\![\ (\mathsf{PutGet}) \text{ for } l_1\ ]\!] \\ & l_0.\mathit{put}\ (s_1, s_2)\ a \\ = & [\![\ (\mathsf{PutGet}) \text{ for } l_2\ ]\!] \\ & l_0.\mathit{put}\ (s_1, s_2)\ (l_2.\mathit{get}\ (l_2.\mathit{put}\ s_2\ a)) \end{array}$$

$$
\begin{array}{ll}
= & [\![\text{ Definition }]\!] \\
& r.put\ (s_1, s_2)\ (l_2.put\ s_2\ a) \\
= & [\![\text{ Definition }]\!] \\
& (r\ ;\ l_2).put\ (s_1, s_2)\ a
\end{array}
$$

Finally, for *create* we have:

$$
\begin{array}{ll}
& (l\ ;\ l_1).create\ a \\
= & [\![\text{ Definition }]\!] \\
& l.create\ (l_1.create\ a) \\
= & [\![\text{ Definition }]\!] \\
& l_0.create\ (l_1.get\ (l_1.create\ a)) \\
= & [\![\ (\mathsf{CreateGet}) \text{ for } l_1 \ ]\!] \\
& l_0.create\ a \\
= & [\![\ (\mathsf{CreateGet}) \text{ for } l_2 \ ]\!] \\
& l_0.create\ (l_2.get\ (l_2.create\ a)) \\
= & [\![\text{ Definition }]\!] \\
& r.create\ (l_2.create\ a) \\
= & [\![\text{ Definition }]\!] \\
& (r\ ;\ l_2).create\ a
\end{array}
$$

Next, we show that $l\ ;\ r_1 = r\ ;\ r_2$. For *get*:

$$
\begin{array}{ll}
& (l\ ;\ r_1).get\ (s_1, s_2) \\
= & [\![\text{ Definition }]\!] \\
& r_1.get\ (l.get\ (s_1, s_2)) \\
= & [\![\text{ definition of } l.get = \mathit{fst}, \mathit{fst} \text{ commutes with } r_1.get \ ]\!] \\
& r_0.get\ (s_1, s_2) \\
= & [\![\text{ reverse above reasoning }]\!] \\
& (r\ ;\ r_2).get\ (s_1, s_2)
\end{array}
$$

For *put*, we have:

$$
\begin{array}{ll}
& (l\ ;\ r_1).put\ (s_1, s_2)\ b \\
= & [\![\text{ Definition }]\!] \\
& l.put\ (s_1, s_2)\ (r_1.put\ s_1\ b) \\
= & [\![\text{ Definition }]\!] \\
& l_0.put\ (s_1, s_2)\ (l_1.get\ (r_1.put\ s_1\ b)) \\
= & [\![\text{ Twist equation }]\!] \\
& l_0.put\ (s_1, s_2)\ (l_2.get\ (r_2.put\ s_2\ b)) \\
= & [\![\text{ Definition }]\!] \\
& r.put\ (s_1, s_2)\ (r_2.put\ s_2\ b) \\
= & [\![\text{ Definition }]\!] \\
& (r\ ;\ r_2).put\ (s_1, s_2)\ b
\end{array}
$$

Finally, for *create* we have:

$$
\begin{array}{ll}
& (l\ ;\ r_1).create\ b \\
= & [\![\text{ Definition }]\!]
\end{array}
$$

$$\begin{array}{ll}
& l.create\ (r_1.create\ b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_0.create\ (l_1.get\ (r_1.create\ b)) \\
= & [\![\ \text{Twist equation}\ ]\!] \\
& l_0.create\ (l_2.get\ (r_2.create\ b)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& r.create\ (r_2.create\ b) \\
= & [\![\ \text{Definition}\ ]\!] \\
& (r\ ;\ r_2).create\ b
\end{array}$$

We must also show that l and r are well-behaved full lenses. To show that l is well-behaved, we proceed as follows. For (GetPut):

$$\begin{array}{ll}
& l.get\ (l.put\ (s_1, s_2)\ s_1') \\
= & [\![\ \text{Definition}\ ]\!] \\
& fst\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1')) \\
= & [\![\ fst \text{ commutes with } put\ ]\!] \\
& l_1.put\ s_1\ (l_1.get\ s_1')) \\
= & [\![\ (\mathsf{GetPut}) \text{ for } l_1\ ]\!] \\
& s_1'
\end{array}$$

For (PutGet):

$$\begin{array}{ll}
& l.put\ (s_1, s_2)\ (l.get\ (s_1, s_2)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_0.put\ (s_1, s_2)\ (l_1.get\ s_1) \\
= & [\![\ \text{Eta-expansion for pairs}\ ]\!] \\
& (fst\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1)), snd\ (l_0.put\ (s_1, s_2)\ (l_1.get\ s_1))) \\
= & [\![\ fst, snd \text{ commutes with } put\ ]\!] \\
& (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_1.get\ s_1)) \\
= & [\![\ l_1.get\ s_1 = l_2.get\ s_2\ ]\!] \\
& (l_1.put\ s_1\ (l_1.get\ s_1), l_2.put\ s_2\ (l_2.get\ s_2)) \\
= & [\![\ (\mathsf{PutGet}) \text{ for } l_1, l_2\ ]\!] \\
& (s_1, s_2)
\end{array}$$

For (CreateGet):

$$\begin{array}{ll}
& l.create\ (l.get\ (s_1, s_2)) \\
= & [\![\ \text{Definition}\ ]\!] \\
& l_0.create\ (l_1.get\ s_1) \\
= & [\![\ \text{Eta-expansion for pairs}\ ]\!] \\
& (fst\ (l_0.create\ (l_1.get\ s_1)), snd\ (l_0.create\ (l_1.get\ s_1))) \\
= & [\![\ fst, snd \text{ commutes with } put\ ]\!] \\
& (l_1.create\ (l_1.get\ s_1), l_1.create\ (l_1.get\ s_1)) \\
= & [\![\ l_1.get\ s_1 = l_2.get\ s_2\ ]\!] \\
& (l_1.create\ (l_1.get\ s_1), l_1.create\ (l_2.get\ s_2)) \\
= & [\![\ (\mathsf{CreateGet})\ ]\!] \\
& (s_1, s_2)
\end{array}$$

Finally, notice that l and r are defined symmetrically so essentially the same reasoning shows r is well-behaved.

To conclude, $sp = (l, r)$ constitutes a span of lenses witnessing that $sp_1 \equiv_{\mathrm{s}} sp_2$. $\square$

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 187–214. Springer, Heidelberg (2015a)

Abou-Saleh, F., McKinna, J., Gibbons, J.: Coalgebraic aspects of bidirectional computation. In: BX 2015, CEUR-WS, vol. 1396, pp. 15–30 (2015b)

Cheney, J., McKinna, J., Stevens, P., Gibbons, J., Abou-Saleh, F.: Entangled state monads. In: Terwilliger and Hidaka (2014)

Diviánszky, P.: LGtk API correction. http://people.inf.elte.hu/divip/LGtk/CorrectedAPI.html

Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. TOPLAS **29**(3), 17 (2007)

Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012)

Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: POPL, pp. 371–384. ACM (2011)

Johnson, M., Rosebrugh, R.: Spans of lenses. In: Terwilliger and Hidaka (2014)

Jones, M.P., Duponcheel, L.: Composing monads. Technical report RR-1004, DCS, Yale (1993)

King, D.J., Wadler, P.: Combining monads. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, pp. 134–143 (1992)

Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: POPL, pp. 333–343 (1995)

Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: PEPM, pp. 39–50. ACM (2014). http://doi.acm.org/10.1145/2543728.2543737

Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002)

Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. SoSyM **9**(1), 7–20 (2010)

Terwilliger, J., Hidaka, S. (eds.): BX Workshop (2014). http://ceur-ws.org/Vol-1133/#bx

TLCBX Project: a theory of least change for bidirectional transformations (2013–2016). http://www.cs.ox.ac.uk/projects/tlcbx/, http://groups.inf.ed.ac.uk/bx/

Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**(4), 461–493 (1992). http://dx.org/10.1017/S0960129500001560

Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)

Conflation Confers Concurrency

Robert Atkey[2](✉), Sam Lindley[1], and J. Garrett Morris[1]

[1] The University of Edinburgh, Edinburgh, Scotland, UK
{Sam.Lindley,Garrett.Morris}@ed.ac.uk
[2] University of Strathclyde, Glasgow, Scotland, UK
Robert.Atkey@strath.ac.uk

Abstract. Session types provide a static guarantee that concurrent programs respect communication protocols. Recent work has explored a correspondence between proof rules and cut reduction in linear logic and typing and evaluation of process calculi. This paper considers two approaches to extend logically-founded process calculi. First, we consider extensions of the process calculus to more closely resemble π-calculus. Second, inspired by denotational models of process calculi, we consider conflating dual types. Most interestingly, we observe that these approaches coincide: conflating the multiplicatives ($\otimes$ and $\parr$) allows processes to share multiple channels; conflating the additives ($\oplus$ and $\&$) provides nondeterminism; and conflating the exponentials ($!$ and $?$) yields access points, a rendezvous mechanism for initiating session typed communication. Access points are particularly expressive: for example, they are sufficient to encode concurrent state and general recursion.

1 Introduction

The Curry-Howard correspondence, formulated by Howard (1980) and building on ideas of Curry (1934) and Tait (1965), observes a remarkable correspondence between (propositional) intuitionistic logic and the λ-calculus. The correspondence identifies logical propositions with λ-calculus types, proofs with well-typed terms, and cut elimination with reduction. The correspondence is fruitful from the perspectives of both logic and programming languages (Wadler, 2015).

Recent work, initiated by Caires and Pfenning (2010), attempts a similar identification between linear logic and process calculi. In this case, propositions of (intuitionistic) linear logic are identified with session types, a mechanism for typing interacting processes originally proposed by Honda (1993). Proofs of those propositions are identified with π-calculus processes, and cut elimination in linear logic with π-calculus reduction. However, this identification is not as satisfying as that for intuitionistic logic and the λ-calculus. In particular, numerous features of the π-calculus are excluded by the resulting session-typing discipline, and there is not yet an approach to restore them. Philip Wadler posed the following question at an invited talk by Frank Pfenning at the TLDI Workshop in January 2012 [paraphrased]:

DOI: 10.1007/978-3-319-30936-1_2

> Simply-typed λ-calculus has a propositions as types interpretation as intuitionistic logic, but has limited expressiveness. By adding a fix point operator it becomes Turing-complete. Similarly, Caires and Pfenning's calculus has a propositions as types interpretation as intuitionistic linear logic. Is there a feature, analogous to the fix point operator, that we can add to Caires and Pfenning's calculus in order to recover the full power of π-calculus?

Wadler has since reposed the same question, but with respect to his classical variant of Caires and Pfenning's system (Wadler (2014)).

This paper describes an approach to more expressive, logically founded process calculi, inspired by Wadler's question. We begin by considering extensions of Wadler's CP calculus that increase its expressiveness—at the cost of properties such as deadlock freedom—while retaining session fidelity (well-typed communication). To do so, we explore two approaches. On the one hand, we directly investigate the inclusion of π-calculus terms excluded by CP's type system, bringing CP more in line with the term calculi in most existing presentations of session types. Doing so requires the addition of new typing rules, and we consider their interpretation as proof rules and their logical consequences. On the other hand, inspired by the semantics of the π-calculus described by Abramsky et al. (1996), we attempt to conflate the various dual types. Our primary contribution is the observation that these disparate approaches converge: the new proof rules allow us to show bi-implications between dual types, while assuming such bi-implications make the new logical rules derivable.

The paper proceeds as follows. We begin with a short introduction to session types and their connection to linear logic and its semantics (Sect. 2). We recall Wadler's CP calculus (Sect. 3). We then describe the conflation of the various dual types and their consequences (Sect. 4). Finally, we conclude with a discussion of future work and open questions (Sect. 5).

2 Background

Unlike the propositions of intuitionistic logic, linear logic propositions are finite resources—the assumptions of a proof must each be used exactly once in the course of the proof. When he introduced linear logic, Girard (1987) suggested that it might be suited to reasoning about parallelism. Abramsky (1992) and Bellin and Scott (1994) give embeddings of linear logic proofs in π-calculus, and show that cut reduction is simulated by π-calculus reduction. Their work is not intended to provide a type system for π-calculus: there are many processes which are not the image of some proof.

Session types were originally introduced by Honda (1993) as a type system for π-calculus-like communicating processes. The key constructors of session types include input and output, characterising the exchange of data, and internal and external choice, characterising branching evaluation. Honda's typing discipline assures *session fidelity*, meaning that at each synchronisation the communicating processes agree on the types of values exchanged. His system is extended to π-calculus-like processes by Takeuchi et al. (1994) and Honda et al. (1998). Session typing relies on a substructural type system to assure session fidelity; however, Honda did not relate his types to the propositions of linear logic, and he relies on a self-dual type for closed channels. Kobayashi et al. (1996) give a linear, typed polyadic π-calculus, in which each channel must be used exactly once, and show how it can be used to encode serial uses, as in session types; they do not address choice or nondeterminism.

Recent work by Caires and Pfenning (2010) and Wadler (2014), among others, has developed a propositions-as-types correspondence between session typing and linear logic. Session types are interpreted via the connectives of linear logic—for example, the tensor product $A \otimes B$ characterises processes that output values of type A, and then proceed as B. Caires and Pfenning (2010) interpret the proof rules for intuitionistic linear logic as a type system for the π-calculus; they then show that the reduction steps for well-typed π-calculus terms correspond to the cut elimination rules for intuitionistic linear logic. Wadler (2014) adapts their approach to classical linear logic, emphasising the role of duality in typing; in his system, the semantics of terms is given directly by the cut elimination rules. As a consequence of their logical connections, both systems enjoy deadlock freedom, termination, and a lack of nondeterministic behaviour; however, both systems also impose additional limitations on the underlying process calculus, and differ in some ways from traditional presentations of session types.

Abramsky et al. (1996) propose an alternative approach to typing communicating processes. Their approach is based on a denotational model for processes, called interaction categories. Their canonical interaction category, called **SProc** is sufficient to interpret linear logic; unusually, the interpretations of the dual connectives are conflated. **SProc** is quite expressive: it can faithfully interpret all of Milner's synchronous CCS calculus. However, we are not aware of any work extending interaction categories to interpret channel-passing calculi, such as π-calculus or the calculi of Caires and Pfenning and Wadler.

There have been several type systems for deadlock-free processes not derived from linear logic, including those of Kobayashi (2006), Padovani (2014), and Giachino et al. (2014). These systems all include composition of processes sharing multiple channels, but at the cost of additional type-system complexity. Dardha and Pérez (2015) compare Kobayashi's approach with the linear-logic based approaches of Caires, Pfenning, and Wadler. They observe that the linear logic-based approaches coincide with that of Kobayashi when restricted to one shared channel between processes. They also give a rewriting procedure from Kobayashi-typable processes with multiple shared channels to processes with one shared channel, and show an operational correspondence between the original and rewritten processes.

Some of our results above are reminiscent of results obtained in category-theoretic settings closely related to linear logic. Fiore (2007) shows that conflation of products and coproducts into biproducts is equivalent to the existence of a monoidal structure on morphisms; we will attempt a similar conflation (Sect. 4.3) to give a logical interpretation of nondeterminism. Compact closed categories provide a model of classical linear logic in which $\otimes$ and $\parr$ are identified; we will consider a similar conflation as well (Sect. 4.2). However, we do not yet understand the exact relationship between compact closed categories, and our system with the multicut rule. In compact closed categories it is known that finite products are automatically biproducts Houston (2008), and it would be interesting to see how his proof translates to a logical setting.

Finally, there have been several attempts to relate π-calculus to proof nets, one approach to the semantics of linear logic. Honda and Laurent (2010) demonstrate a two-way correspondence between proof nets for polarised linear logic and a typed π-calculus previous presented by Honda et al. (2004). The type system used for the π-calculus in this work is rather restrictive, however: it ensures that all processes are deterministic and effectively sequential, removing much of the expressivity of the π-calculus. Ehrhard and Laurent (2010) give a translation from finitary π-calculus (i.e., π-calculus without replication) into differential interaction nets. Differential interaction nets are an untyped formalism for representing computation, based on proof nets for differential linear logic. Differential linear logic is an extension of linear logic that, amongst other features adds a form of nondeterminism. It is this nondeterminism that Ehrhard and Laurent use to model the π-calculus. However, Mazza (2015) argues forcefully that Ehrhard and Laurent's translation incorrectly models the nondeterminism present in the π-calculus. In short, Mazza shows that differential proof nets can only model "localised" nondeterminism—nondeterminism that can be resolved via a local coin flip—and not the global nondeterminism that arises when two processes "race" to communicate with another process. Mazza makes the following provocative statement:

> However, although linear logic has kept providing, even in recent times, useful tools for ensuring properties of process algebras, especially via type systems (Kobayashi et al., 1999; Yoshida et al., 2004; Caires and Pfenning, 2010; Honda and Laurent, 2010), all further investigations have failed to bring any deep logical insight into concurrency theory, in the sense that no concurrent primitive has found a convincing counterpart in linear logic, or anything even remotely resembling the perfect correspondence between functional languages and intuitionistic logic. In our opinion, we must simply accept that linear logic is not the right framework for carrying out Abramsky's "proofs as processes" program (which, in fact, more than 20 years after its inception has yet to see a satisfactory completion).

We will show that, while not entirely answering Mazza's critique, our identifications provide some logical justification for π-calculus-like features.

3 Classical Linear Logic and the Process Calculus CP

We begin by reviewing Wadler's CP calculus (Wadler, 2014), its typing, and its semantics. For simplicity, we restrict ourselves to the propositional fragment of CP, omitting second-order existential and universal quantification.

3.1 Types and Duality

The types of the CP calculus are built from the multiplicative, additive, and exponential propositional connectives of Girard's classical linear logic (CLL).

$$\text{Types}\, A, B ::= A \otimes B \mid A \parr B \mid 1 \mid \bot \mid A \oplus B \mid A \,\&\, B \mid 0 \mid \top \mid\, !A \mid ?A$$

Wadler's contribution with the CP calculus was to give the logical connectives of CLL an explicit reading in terms of session types. As is standard with session typing, CP types denote the types of channel end points. The connectives come in dual pairs, indicating complementary obligations for each end point of a channel. The tensor connective $A \otimes B$ means output A and then behave like B; and dually, par $A \parr B$ means input A and then behave like B. The unit of $\otimes$ is 1, empty output; dually, the unit of $\parr$ is $\bot$, empty input. Internal choice $A \oplus B$ means make a choice between A and B; dually, external choice $A \,\&\, B$ means accept a choice of A or B. Replication $!A$ means produce an arbitrary number of copies of A; dually, query $?A$ means consume a copy of A.

The dual relationships between $\otimes$ and $\parr$, 1 and $\bot$, $\oplus$ and $\&$, and ! and ? are formalised in the duality operation $-^\perp$, which takes each type to its dual.

$$\begin{array}{rclcrclcrcl}
(A \otimes B)^\perp &=& A^\perp \parr B^\perp & & (A \oplus B)^\perp &=& A^\perp \,\&\, B^\perp & & & & \\
(A \parr B)^\perp &=& A^\perp \otimes B^\perp & & (A \,\&\, B)^\perp &=& A^\perp \oplus B^\perp & & (!A)^\perp &=& ?(A^\perp) \\
1^\perp &=& \bot & & \top^\perp &=& 0 & & (?A)^\perp &=& !(A^\perp) \\
\bot^\perp &=& 1 & & 0^\perp &=& \top & & & &
\end{array}$$

Example: Sending and Receiving Bits. CP is a rather low-level calculus. The multiplicative fragment ($\otimes$ and $\parr$) handles only sending and receiving of channels, on which data may be subsequently transmitted. The only actual data that may be transmitted between processes are single bits, which take the form of a choice between a pair of sessions. Transmission of a single bit is represented by internal choice between two empty outputs.

$$\mathit{Bool} = 1 \oplus 1$$

Dually, receiving a single bit is represented by an external choice between two empty inputs.

$$\mathit{Bool}^\perp = \bot \,\&\, \bot$$

The other propositional connectives of CLL can now be used to build more complex specifications. For example, we can write down the type of a server that offers a choice of a binary operation on booleans (single bits) and a unary operation on booleans, arbitrarily many times.

$$\mathit{Server} = !((\mathit{Bool}^\perp \parr \mathit{Bool}^\perp \parr \mathit{Bool} \otimes 1) \,\&\, (\mathit{Bool}^\perp \parr \mathit{Bool} \otimes 1))$$

The outer ! indicates that an implementation of this type is obliged to offer the server as many times as a client requires. The inner part of the type is a client-choice, indicated by the external choice ($\&$), between the two operations. The left-hand choice $Bool^{\perp} ⅋ Bool^{\perp} ⅋ Bool \otimes 1$ can be read as "the server must input two booleans, output a boolean, and then signal the end of the session". The right-hand choice is similar, but only specifies the input of a single boolean input.

A compatible client type is obtained by taking the dual of the *Server* type.

$$Client = Server^{\perp} = ?((Bool \otimes Bool \otimes Bool^{\perp} ⅋ \perp) \oplus (Bool \otimes Bool^{\perp} ⅋ \perp))$$

Dually to the server's obligations, the outer ? indicates that the client may make as many uses of this session as it desires. The inner part of the type is an internal choice ($\oplus$), indicating that the implementation of this type must make a choice between the two services. The two choices then describe the same pattern as the server, but from the point of view of the client. The left-hand choice $Bool \otimes Bool \otimes Bool^{\perp} ⅋ \perp$ can be read as "the client must output two booleans, input a boolean, and then signal the end of the session". The right-hand choice is similar, but only specifies the output of a single boolean.

3.2 Terms and Typing

The terms of CP are given by the following grammar.

$$\begin{aligned}\text{Terms}\, P, Q ::= x \leftrightarrow y \mid \nu y\, (P \mid Q) \mid x(y).P \mid x[y].(P \mid Q) \mid !x(y).P \mid ?x[y].P \\ \mid x[in_i].P \mid \mathsf{case}\, x.\{P; Q\} \mid x().P \mid x[].0 \mid \mathsf{case}\, x.\{\}\end{aligned}$$

Figure 1 gives the typing rules of CP. The typing judgement is of the form $P \vdash \Gamma$, where P is a CP process term, and Γ is a channel typing environment. In rule (Bang), $?\Gamma$ denotes a context Γ in which all types are of the form $?A$ for some type A. Note that CP's typing rules implicitly rebind identifiers: for example, in the hypothesis of the rule for ⅋, x identifies a proof of B, while in the conclusion it identifies a proof of $A ⅋ B$.

CP includes two rules that are logically derivable: the axiom rule, which is interpreted as channel forwarding, and the cut rule, which is interpreted as process composition. Two of CP's terms differ from standard π-calculus terms. The first is composition—rather than having distinct name restriction and composition operators, CP provides one combined operator. This syntactically captures the restriction that composed processes must share exactly one channel. The second is output: the CP term $x[y].(P \mid Q)$ includes output, composition, and name restriction (the name y designates a new channel, bound in P). Finally, note that CP includes only replicated input, not arbitrary replicated processes.

A Simpler Send. The CP rule Tensor is appealing because if one erases the terms it is exactly the classical linear logic rule for tensor. However, this correspondence comes at a price. Operationally, the process $x[y].(P \mid Q)$ does three things: it introduces a fresh variable y, it sends y to a freshly spawned process P, and in parallel it continues as process Q. Fortunately, we can straightforwardly

Typing

$$\textsc{Axiom}\ \frac{}{x \leftrightarrow w \vdash x : A, w : A^\perp} \qquad \textsc{Cut}\ \frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^\perp}{\nu x\ (P \mid Q) \vdash \Gamma, \Delta} \qquad \textsc{One}\ \frac{}{x[].0 \vdash x : 1}$$

$$\textsc{Tensor}\ \frac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \qquad \textsc{Par}\ \frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \parr B} \qquad \textsc{Bot}\ \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}$$

$$\textsc{Plus}\ \frac{P \vdash \Gamma, x : A_i}{x[in_i].P \vdash \Gamma, x : A_1 \oplus A_2} \qquad \textsc{With}\ \frac{P \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{\mathsf{case}\ x.\{P; Q\} \vdash \Gamma, x : A \,\&\, B} \qquad \textsc{Top}\ \frac{}{\mathsf{case}\ x.\{\} \vdash \Gamma, x : \top}$$

$$\textsc{Bang}\ \frac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A} \qquad \textsc{Query}\ \frac{P \vdash \Gamma, y : A}{?x[y].P \vdash \Gamma, x : ?A}$$

$$\textsc{Weaken?}\ \frac{P \vdash \Gamma}{P \vdash \Gamma, x : ?A} \qquad \textsc{Contract?}\ \frac{P \vdash \Gamma, y : ?A, z : ?A}{P\{x/y, x/z\} \vdash \Gamma, x : ?A}$$

Fig. 1. CP typing

define the operation that sends a free variable along a channel as syntactic sugar (Lindley and Morris (2015) discuss this in more detail).

$$x\langle y\rangle.P \stackrel{\text{def}}{=} x[z].(y \leftrightarrow z \mid P)$$

Example: Sending and Receiving Bits. As mentioned above, CP is quite low-level, so we use some syntactic sugar for sending and receiving bits.

$$\begin{aligned} x[0].P &\stackrel{\text{def}}{=} x[y].(y[in_1].y[].0 \mid P) \\ x[1].P &\stackrel{\text{def}}{=} x[y].(y[in_2].y[].0 \mid P) \\ \mathsf{case}\ p.\{0 \mapsto P; 1 \mapsto Q\} &\stackrel{\text{def}}{=} \mathsf{case}\ p.\{p().P; p().Q\} \end{aligned}$$

Let us define a logic server process $P \vdash x : \mathit{Server}$, in which the binary operation is "and", and the unary operation is "not".

$$\begin{aligned} P = {}!x(y).\mathsf{case}\ y.\{&y(p).y(q). \\ &\quad \mathsf{case}\ p.\{0 \mapsto \mathsf{case}\ q.\{0 \mapsto y[0].y[].0; 1 \mapsto y[0].y[].0\}; \\ &\qquad\qquad 1 \mapsto \mathsf{case}\ q.\{0 \mapsto y[0].y[].0; 1 \mapsto y[1].y[].0\}\}; \\ &y(p). \\ &\quad \mathsf{case}\ p.\{0 \mapsto y[1].y[].0; 1 \mapsto y[0].y[].0\}\} \end{aligned}$$

This process operates over replicated channel x, accepting a channel y. A client process communicating along the other end of x will begin by choosing between the "and" and "not" operations. If "and" is requested, then two bits (p and q) are received along y, and their logical conjunction is sent back along y. If "not" is requested, then a single bit (p) is received along y, and its logical negation is sent back along y.

We now define a client process Q that uses P to compute "not (p and q)", using the result to choose between two processes P_0 and P_1 as continuations.

$$\begin{aligned} Q = {} & ?x[y].y[in_1].y\langle p\rangle.y\langle q\rangle.y(z).y(). \\ & ?x[y].y[in_2].y\langle z\rangle.y(r).y(). \\ & \mathsf{case}\, r.\{0 \mapsto P_0; 1 \mapsto P_1\} \end{aligned}$$

The process Q connects to P twice using channel x, selecting the "and" operation, and then the "not" operation. We have that $Q \vdash p : \mathit{Bool}^{\perp}, q : \mathit{Bool}^{\perp}, x : \mathit{Client}, \Delta$ whenever $P_0 \vdash \Delta$ and $P_1 \vdash \Delta$.

3.3 Semantics via Cut Reduction

The semantics of CP terms are given by cut reduction, as shown in Fig. 2. We write $fv(P)$ for the free names of process P. Terms are identified up to structural congruence $\equiv$ (as name restriction and composition are combined into one form, composition is not always associative). We write $\longrightarrow_{\mathrm{C}}$ for the cut reduction relation and $\longrightarrow_{\mathrm{CC}}$ for the commuting conversion relation. We denote sequential composition of relations by juxtaposition. We write R^* for the transitive reflexive closure and $R^?$ for the reflexive closure of relation R.

The majority of the cut reduction rules correspond closely to synchronous reductions in π-calculus—for example, the reduction of $\&$ against $\oplus$ corresponds to the synchronisation of an internal and external choice. The rule for reduction of $\parr$ against $\otimes$ is more complex than synchronisation of input and output in π-calculus, as it must also manipulate the implicit name restriction and parallel composition in the output term. We deviate slightly from Wadler's presentation in that substitution and duplication and discarding of replicated processes happens in the structural congruence rules for weakening and contraction and not as part of the rule for dereliction. These choices ensure that reduction only concerns interactions between dual prefixes and other non-intensional behaviour is handled by the structural congruence.

Theorem 1 (Preservation). *The relations $\equiv$, $\longrightarrow_C$, and $\longrightarrow_{CC}$ preserve well-typing: if $P \vdash \Gamma$ then $P \equiv Q$ implies $Q \vdash \Gamma$, $P \longrightarrow_C Q$ implies $Q \vdash \Gamma$, and $P \longrightarrow_{CC} Q$ implies $Q \vdash \Gamma$.*

Just as cut elimination in logic ensures that any proof may be transformed into an equivalent cut-free proof, the reduction rules of CP transform any term into a term blocked on external communication—that is to say, if $P \vdash \Gamma$, then $P\,(\equiv\longrightarrow_{\mathrm{C}})^*(\equiv\longrightarrow_{\mathrm{CC}})^?\,P'$ where $P' \not\equiv \nu x\,(Q \mid Q')$ for any x, Q, Q'. The optional final commuting conversion plays a crucial role in this transformation, moving any remaining internal communication after some external communication (precluding deadlock). Note, however, that commuting conversions do not correspond to computational steps (i.e., any reduction rule in π-calculus).

Structural congruence

$$\begin{aligned}
x \leftrightarrow w &\equiv w \leftrightarrow x \\
\nu x\,(w \leftrightarrow x \mid P) &\equiv P[w/x] \\
\nu x\,(P \mid Q) &\equiv \nu x\,(Q \mid P) \\
\nu x\,(P \mid \nu y\,(Q \mid R)) &\equiv \nu y\,(\nu x\,(P \mid Q) \mid R), \quad \text{if } x \notin fv(R) \\
\nu x\,(!x(y).P \mid Q) &\equiv Q, \qquad x \notin fv(Q) \\
\nu x\,(!x(y).P \mid Q\{x/z\}) &\equiv \nu x\,(!x(y).P \mid \nu z\,(!z(y).P \mid Q))
\end{aligned}$$

Primary cut reduction rules

$$\begin{aligned}
\nu x\,(x[].0 \mid x().P) &\longrightarrow_{\mathrm{C}} P \\
\nu x\,(x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_{\mathrm{C}} \nu x\,(Q \mid \nu y\,(P \mid R)) \\
\nu x\,(x[in_i].P \mid \mathsf{case}\; x.\{Q_1; Q_2\}) &\longrightarrow_{\mathrm{C}} \nu x\,(P \mid Q_i) \\
\nu x\,(!x(y).P \mid ?x[y].Q) &\longrightarrow_{\mathrm{C}} \nu y\,(P \mid Q), \quad \text{if } x \notin fv(Q)
\end{aligned}$$

$$\frac{P \longrightarrow_{\mathrm{C}} P'}{\nu x\,(P \mid Q) \longrightarrow_{\mathrm{C}} \nu x\,(P' \mid Q)}$$

Commuting conversions

$$\begin{aligned}
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(\nu z\,(P \mid R) \mid Q), \qquad \text{if } z \notin fv(Q) \\
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(P \mid \nu z\,(Q \mid R)), \qquad \text{if } z \notin fv(P) \\
\nu z\,(x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} x(y).\nu z\,(P \mid Q) \\
\nu z\,(x().P \mid Q) &\longrightarrow_{\mathrm{CC}} x().\nu z\,(P \mid Q) \\
\nu z\,(x[in_i].P \mid Q) &\longrightarrow_{\mathrm{CC}} x[in_i].\nu z\,(P \mid Q) \\
\nu z\,(\mathsf{case}\; x.\{P; Q\} \mid R) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\; x.\{\nu z\,(P \mid R); \nu z\,(Q \mid R)\} \\
\nu z\,(\mathsf{case}\; x.\{\} \mid Q) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\; x.\{\} \\
\nu z\,(!x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} !x(y).\nu z\,(P \mid Q) \\
\nu z\,(?x[y].P \mid Q) &\longrightarrow_{\mathrm{CC}} ?x[y].\nu z\,(P \mid Q)
\end{aligned}$$

Fig. 2. CP congruences and cut reduction

Example: Sending and Receiving Bits. Recall that, under the assumption of two processes $P_0 \vdash \Delta$ and $P_1 \vdash \Delta$, we have $P \vdash x : \mathit{Server}$ and $Q \vdash p : \mathit{Bool}^{\perp}, q : \mathit{Bool}^{\perp}, x : \mathit{Client}, \Delta$. In order to connect the client and server, we bind x to the replicated channel they communicate along: $PQ = \nu x\,(P \mid Q)$. We can set the values of the bits p and q by placing further processes in parallel with PQ, which allows reduction. For instance,

$$\nu p\,(p[1] \mid \nu q\,(q[1] \mid PQ)) \longrightarrow_{\mathrm{C}}^{*} P_0$$

and:

$$\nu p\,(p[0] \mid \nu q\,(q[1] \mid PQ)) \longrightarrow_{\mathrm{C}}^{*} P_1$$

Properties of Cut Reduction. Unsurprisingly, being a term calculus for classical linear logic, CP is well-behaved. CP cut reduction is terminating, CP does not admit deadlocks, and CP is deterministic.

Theorem 2 (Termination). *The relation $\longrightarrow_C$ is strongly-normalising modulo $\equiv$: if $P \vdash \Gamma$, then there are no infinite $\equiv\longrightarrow_C$ reduction sequences starting from P.*

Theorem 3 (Deadlock-Freedom). *If $P \vdash \Gamma$, then there exist P', Q such that $P\,(\equiv\longrightarrow_C)^*\,P'$ and $P\,(\equiv\longrightarrow_{CC})^?\,Q$, and Q is not a cut.*

Theorem 4 (Determinism). *The relation $\longrightarrow_C$ is confluent modulo $\equiv$: if $P \vdash \Gamma$, $P\,(\equiv\longrightarrow_C)^*\,Q_1$ and $P\,(\equiv\longrightarrow_C)^*\,Q_2$, then there exist R_1, R_2 such that $Q_1\,(\equiv\longrightarrow_C)^*\,R_1$, $Q_2\,(\equiv\longrightarrow_C)^*\,R_2$, and $R_1 \equiv R_2$.*

All of these theorems follow from well known results about classical linear logic.

4 Conflating Duals

In light of the results above about CP's semantics, it would seem that CP is not a particularly expressive calculus. In particular, nondeterminism is frequently seen as a defining characteristic of concurrency, and the guaranteed deterministic behaviour of CP (Theorem 4) would appear to indicate that it is not possible to express much interesting behaviour in CP.

One property of CP that appears to be essential for its behavioural properties is its strict adherence to duality. Each connective has an accompanying dual, and the fact that communicating processes must match dual connectives precisely means that "nothing goes wrong", in the sense of non-termination, deadlock or nondeterministic behaviour. Observing that this precise matching makes CP a relatively inexpressive calculus, we now systematically investigate relaxation of the strict duality of CP to see whether or not it yields greater expressivity by conflating each of the dual pairs of connectives in turn.

How to Conflate Duals. There are choices over exactly how to conflate duals in a proof theory. In a standard session-typed calculus, for instance, one literally replaces 1 and $\bot$ with a single type (usually called end). However, in order to keep our modified calculus conservative with respect to the existing one, we initially take a different approach following Wadler (2014), who considers extensions of CP in which certain maps between duals are derivable. Logically, we can say that we have conflated propositions (session types) A and B if there exist processes P_{AB} and P_{BA} such that $P_{AB} \vdash x : A \multimap B$ and $P_{BA} \vdash x : B \multimap A$ (where $A \multimap B = A^{\perp} \parr B$), which amounts to giving back-and-forth translations between A and B. In this paper, we do not necessarily require that these translations be mutually inverse in any way, though we usually expect this to be the case. Our general pattern will be to add some feature (parallelism, multi-channel cut, nondeterminism, access points) and then observe that this is logically equivalent to conflation of a dual pair.

As indicated above, we define $A \multimap B \overset{\text{def}}{=} A^\perp \parr B$. Furthermore, $\parr$ is invertible

$$\dfrac{P \vdash z : A \parr B \qquad \dfrac{\dfrac{}{w \leftrightarrow x \vdash w : A^\perp, x : A} \qquad \dfrac{}{y \leftrightarrow z \vdash y : B, z : B^\perp}}{\vdash z : A^\perp \otimes B^\perp, x : A, y : B}}{\nu z\,(P \mid z[w].(w \leftrightarrow x \mid y \leftrightarrow z)) \vdash x : A, y : B}$$

so without loss of generality we shall seek P_{AB} and P_{BA} such that $P_{AB} \vdash x : A^\perp, y : B$ and $P_{BA} \vdash x : B^\perp, y : A$ and we shall systematically treat a proof $P_{AB} \vdash x : A^\perp, y : B$ as a proof of $A \multimap B$.

4.1 Concurrency Without Communication (1 and $\perp$)

The first pair of connectives we conflate are 1 and $\perp$. Conflation of this pair corresponds to the addition of communication-free concurrency and inactive processes to CP, via the Mix and 0-Mix rules.

The Mix rule allows processes P and Q to be composed in parallel $P \mid Q$ without creating a communication channel between them.

$$\textsc{Mix}\quad \dfrac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}$$

The unit of Mix is the inactive process 0.

$$\textsc{0-Mix}\quad \dfrac{}{0 \vdash}$$

Using Mix we can prove $\perp \multimap 1$.

$$\dfrac{\dfrac{}{x[].0 \vdash x : 1} \qquad \dfrac{}{y[].0 \vdash y : 1}}{x[].0 \mid y[].0 \vdash x : 1, y : 1}$$

Conversely, if we already have a proof $P_{\perp 1}$ of $\perp \multimap 1$, then we can derive a proof of the Mix rule.

$$\dfrac{\dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp} \qquad \dfrac{\dfrac{Q \vdash \Delta}{y().Q \vdash \Delta, y : \perp} \qquad \dfrac{}{P_{\perp 1} \vdash x : 1, y : 1}}{\nu y\,(Q \mid P_{\perp 1}) \vdash \Delta, x : 1}}{\nu x\,(P \mid \nu y\,(Q \mid P_{\perp 1})) \vdash \Gamma, \Delta}$$

Thus, Mix and $\perp \multimap 1$ are logically equivalent and we choose to define a process $P_{\perp 1}$ in terms of Mix.

$$P_{\perp 1} \overset{\text{def}}{=} x[].0 \mid y[].0$$

In the other direction, using 0-Mix we can prove $1 \multimap \bot$.

$$\dfrac{\dfrac{\overline{0 \vdash}}{y().0 \vdash y : \bot}}{x().y().0 \vdash x : \bot, y : \bot}$$

Conversely, if we already have a proof $P_{1\bot}$ of $1 \multimap \bot$, then we can derive a proof of the 0-Mix rule.

$$\dfrac{\dfrac{\overline{P_{1\bot} \vdash x : \bot, y : \bot} \qquad \overline{y[].0 \vdash y : 1}}{\nu y\,(P_{1\bot} \mid y[].0) \vdash x : \bot} \qquad \overline{x[].0 \vdash x : 1}}{\nu x\,(\nu y\,(P_{1\bot} \mid y[].0) \mid x[].0) \vdash}$$

Thus, 0-Mix and $1 \multimap \bot$ are logically equivalent and we choose to define the process $P_{1\bot}$ in terms of 0-mix.

$$P_{1\bot} \stackrel{\text{def}}{=} x().y().0$$

Semantics. To account for Mix and 0-Mix, we extend the structural congruence with the following rules.

$$\begin{aligned}
P \mid 0 &\equiv P \\
P \mid Q &\equiv Q \mid P \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
\nu x\,(P \mid (Q \mid R)) &\equiv (\nu x\,(P \mid Q)) \mid R, \quad \text{if } x \notin fv(R)
\end{aligned}$$

The only cut reduction rule we add allows reduction under a Mix.

$$\dfrac{P \longrightarrow_{\mathrm{C}} P'}{P \mid Q \longrightarrow_{\mathrm{C}} P' \mid Q}$$

For each standard commuting conversion involving a cut, there is a corresponding one where the cut is replaced by a Mix.

$$\begin{aligned}
x[y].(P \mid Q) \mid R &\longrightarrow_{\mathrm{CC}} x[y].((P \mid R) \mid Q) \\
x[y].(P \mid Q) \mid R &\longrightarrow_{\mathrm{CC}} x[y].(P \mid (Q \mid R)) \\
(x(y).P) \mid Q &\longrightarrow_{\mathrm{CC}} x(y).(P \mid Q) \\
(x().P) \mid Q &\longrightarrow_{\mathrm{CC}} x().(P \mid Q) \\
(x[in_i].P) \mid Q &\longrightarrow_{\mathrm{CC}} x[in_i].(P \mid Q) \\
\mathsf{case}\, x.\{P; Q\} \mid R &\longrightarrow_{\mathrm{CC}} \mathsf{case}\, x.\{P \mid R; Q \mid R\} \\
\mathsf{case}\, x.\{\} \mid Q &\longrightarrow_{\mathrm{CC}} \mathsf{case}\, x.\{\}
\end{aligned}$$

Adding Mix and 0-Mix does not disturb any of the usual meta-theoretic properties of CP: termination, deadlock-freedom, and determinism all still hold. Therefore, the conflation of 1 and $\bot$ does not greatly alter the properties of CP, though it is does weaken the separation between $\otimes$ and $\parr$, as we shall see in the next section.

4.2 Concurrency with Multiple Channels ($\otimes$ and $\parr$)

The second pair of connectives we consider for conflation are $\otimes$ and $\parr$. This conflation is logically equivalent to the addition of a multi-cut rule that enables communication over several channels at the same time, generalising the single channel of the standard Cut rule, and the zero channels of the Mix rule. As we shall see, the introduction of this rule results in the possibility of deadlock.

Conflation of the units 1 and $\bot$ of $\otimes$ and $\parr$ via Mix already yields a translation from the former to the latter.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \leftrightarrow y \vdash x : A^\perp, y : A} \qquad \dfrac{}{w \leftrightarrow z \vdash w : B^\perp, z : B}}{x \leftrightarrow y \mid w \leftrightarrow z \vdash x : A^\perp, w : B^\perp, y : A, z : B}}{y(z).(x \leftrightarrow y \mid w \leftrightarrow z) \vdash x : A^\perp \parr B^\perp, y : A, z : B}}{x(w).y(z).(x \leftrightarrow y \mid w \leftrightarrow z) \vdash x : A^\perp \parr B^\perp, y : A \parr B}$$

Conversely, if we already have a proof $P_{\otimes\ \parr}$ of $A \otimes B \multimap A\ \parr\ B$, then we can prove $\vdash 1, 1$, and hence the Mix rule

$$\dfrac{\dfrac{\dfrac{}{P_{\otimes\parr} \vdash x : 1 \parr 1, y : \bot \parr \bot} \qquad \dfrac{\dfrac{}{\vdash x : 1} \qquad \dfrac{}{\vdash y : 1}}{y[x].(x[].0 \mid y[].0) \vdash y : 1 \otimes 1}}{\nu y\ (P_{\otimes\parr} \mid y[x].(x[].0 \mid y[].0)) \vdash x : 1 \parr 1}}{\nu x\ (\nu y\ (P_{\otimes\parr} \mid y[x].(x[].0 \mid y[].0)) \mid x[w].w \leftrightarrow z \mid x \leftrightarrow y) \vdash y : 1, z : 1}$$

where we make use of the invertibility of $\parr$.

Whereas Mix is a variation on cut in which no channels communicate between the two processes, another natural variation is to allow two channels to communicate simultaneously between two processes.

$$\textsc{BiCut}\quad \dfrac{P \vdash \Gamma, x : A^\perp, y : B^\perp \qquad Q \vdash \Delta, x : A, y : B}{\nu xy\ (P \mid Q) \vdash \Gamma, \Delta}$$

In general one might consider n channels communicating across a cut so the Mix rule is the nullary case of such a multicut.

Using BiCut we can show that $A\ \parr\ B \multimap A \otimes B$.

$$\dfrac{\dfrac{\dfrac{}{y : A^\perp, w : A} \qquad \dfrac{}{x : B^\perp, z : B}}{x : A^\perp \otimes B^\perp, w : A, z : B} \qquad \dfrac{\dfrac{}{x : A, w : A^\perp} \qquad \dfrac{}{y : B, z : B^\perp}}{y : A \otimes B, w : A^\perp, z : B^\perp}}{\begin{array}{l}\nu wz\ (x[y].(w \leftrightarrow y \mid x \leftrightarrow z) \\ \qquad \mid y[x].(w \leftrightarrow x \mid y \leftrightarrow z))\end{array} \vdash x : A^\perp \otimes B^\perp, y : A \otimes B}$$

Conversely, if we already have a proof $P_{\parr\otimes}$ of $A\ \parr\ B \multimap A \otimes B$, then we can derive a proof of the BiCut rule.

$$\dfrac{\dfrac{P \vdash \Gamma, x : A^\perp, y : B^\perp}{y[x].P \vdash \Gamma, y : A^\perp \otimes B^\perp} \qquad \dfrac{Q \vdash \Delta, x : A, y : B}{y(x).Q \vdash \Delta, y : A \parr B}}{\nu y\ (y[x].P \mid y(x).Q) \vdash \Gamma, \Delta}$$

where we write $y[x].P$ as syntactic sugar for $\nu z\ (z(x).P\{z/y\} \mid P_{⅋\otimes})$.

$$\dfrac{\dfrac{\dfrac{P \vdash \Gamma, x : A, y : B}{P\{z/y\} \vdash \Gamma, x : A, z : B}}{z(y).P\{z/y\} \vdash \Gamma, z : A ⅋ B} \qquad \dfrac{}{P_{⅋\otimes} \vdash z : (A ⅋ B)^{\perp}, y : A \otimes B}}{\nu z\ (z(x).P\{z/y\} \mid P_{⅋\otimes}) \vdash \Gamma, y : A \otimes B}$$

Thus, BiCut and $A\ ⅋\ B \multimap A \otimes B$ are logically equivalent and we choose to define the process $P_{⅋\otimes}$ in terms of BiCut.

$$P_{⅋\otimes} = \nu wz\ (x[y].(w \leftrightarrow y \mid x \leftrightarrow z) \mid y[x].(w \leftrightarrow x \mid y \leftrightarrow z))$$

We can also derive 0-Mix using BiCut and Axiom.

$$0 \overset{\text{def}}{=} \nu xy\ (x \leftrightarrow y \mid x \leftrightarrow y) \vdash$$

As 0 does not have any observable behaviour in any context and neither does this term, we can perfectly well take this to be our definition of 0 when working with variants of CP that include BiCut.

From Bicut to Multicut. A natural way to try to define cut reduction in the presence of BiCut is to lift each of the existing cut rules to bicut rules in which the inactive name remains in place. Unfortunately, this does not work for the rule reducing the interaction between $\otimes$ introduction and ⅋ introduction (i.e. sending and receiving a fresh channel).

$$\nu zx\ (x[y].(P \mid Q) \mid x(y).R) \longrightarrow_{\mathrm{C}} \nu zy\ (P \mid \nu x\ (Q \mid R))$$

The problem is that z may be bound in Q, so moving Q across the z cut may require contradictory types for z. One can attempt to work around the issue by analysing whether z occurs in P or Q. But if z has type $?A$ for some A then it may occur in both. To avoid the problem we take a more uniform approach, introducing a multicut construct that generalises Mix, BiCut, and Cut.

$$\textsc{MultiCut}\ \dfrac{P \vdash \Gamma, x_1 : A_1^{\perp}, \ldots, x_n : A_n^{\perp} \qquad Q \vdash \Delta, x_1 : A_1, \ldots, x_n : A_n}{\nu x_1, \ldots, x_n\ (P \mid Q) \vdash \Gamma, \Delta}$$

Now we can construct a send/receive rule that does not move any of the subterms across a cut. (We shall delegate all such cut-shuffling to the structural congruence.)

$$\nu \vec{x}x\ (x[y].(P \mid Q) \mid x(y).R) \longrightarrow_{\mathrm{C}} \nu \vec{x}xy\ ((P \mid Q) \mid R)$$

Extending the pattern for BiCut, one can encode n-cut, for $n > 2$ in terms of $P_{⅋\otimes}$ (and hence BiCut).

$$\nu x_1 \ldots x_n\ (P \mid Q) \overset{\text{def}}{=} \nu x_n\ (x_n[x_1]. \cdots x[x_{n-1}].P \mid x_n(x_1). \cdots x_n(x_{n-1}).Q)$$

This shows us that BiCut is logically equivalent to MultiCut, but it does not help with defining the semantics of BiCut, which we specify using MultiCut anyway.

Semantics. The structural congruence in the presence of MultiCut is as follows.

$$\nu\vec{x}xy\vec{y}\,(P \mid Q) \equiv \nu\vec{x}yx\vec{y}\,(P \mid Q)$$

$$\begin{aligned}
x \leftrightarrow w &\equiv w \leftrightarrow x \\
\nu\vec{x}x\,(w \leftrightarrow x \mid P) &\equiv P[w/x], \quad \text{if } w \notin \vec{x} \\
\nu\vec{x}\,(P \mid Q) &\equiv \nu\vec{x}\,(Q \mid P) \\
\nu\vec{x}\vec{y}\,(P \mid \nu\vec{z}\,(Q \mid R)) &\equiv \nu\vec{y}\vec{z}\,(\nu\vec{x}\,(P \mid Q) \mid R), \\
&\qquad \text{if } \vec{x} \notin fv(R) \text{ and } \vec{z} \notin fv(P) \\
\nu\vec{x}x\,(!x(y).P \mid Q) &\equiv Q, \qquad \text{if } x \notin fv(Q) \\
\nu\vec{x}x\,(!x(y).P \mid Q\{x/z\}) &\equiv \nu\vec{x}xz\,((!x(y).P \mid !z(y).P) \mid Q)
\end{aligned}$$

The first rule allows reordering of bound variables. The remaining rules correspond to the original structural rules, taking into account the possibility of additional cut variables. The third rule is constrained such that a substitution can only be performed if the substituted variable is not bound by the cut. The fifth rule allows us to focus on any pair of processes while moving the cut variables between the multicuts appropriately. The final rule takes advantage of Mix to avoid the problem we already encountered with adapting the original send/receive rule for use with BiCut.

The primary cut rules are as follows.

$$\begin{aligned}
\nu\vec{x}\,(x[].0 \mid x().P) &\longrightarrow_{\mathrm{C}} P \\
\nu\vec{x}x\,(x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_{\mathrm{C}} \nu\vec{x}xy\,((P \mid Q) \mid R) \\
\nu\vec{x}x\,(x[in_i].P \mid \mathsf{case}\, x.\{Q_1; Q_2\}) &\longrightarrow_{\mathrm{C}} \nu\vec{x}x\,(P \mid Q_i) \\
\nu\vec{x}x\,(!x(y).P \mid ?x[y].Q) &\longrightarrow_{\mathrm{C}} \nu\vec{x}y\,(P \mid Q), \quad \text{if } x \notin fv(Q)
\end{aligned}$$

$$\frac{P \longrightarrow_{\mathrm{C}} P'}{\nu\vec{x}\,(P \mid Q) \longrightarrow_{\mathrm{C}} \nu\vec{x}\,(P' \mid Q)}$$

The send/receive rule uses mix and extends the existing multicut. The other rules are straightforward generalisations of the original ones.

Adding MultiCut introduces the possibility of deadlock. Termination (but not cut-elimination) and determinism are preserved.

4.3 Nondeterminism (0 and $\top/\oplus$ and $\&$)

The previous two sections dealt with the conflation of the multiplicative connectives, $1/\bot$ and $\otimes/⅋$. Conflation of the additive connectives $0/\top$, and $\oplus/\&$ yields a calculus that has *local* nondeterminism: processes can be defined as the nondeterministic combination of two processes, or as processes that may fail.

As in π-calculus, we can easily extend CP with a construct to nondeterministically choose between two processes.

Choose
$$\frac{P \vdash \Gamma \qquad Q \vdash \Gamma}{P + Q \vdash \Gamma}$$

The unit of nondeterminism fail makes no choices.

$$\textsc{Fail}\qquad \frac{}{\mathsf{fail} \vdash \Gamma}$$

Binary nondeterminism does not change the properties we can prove, and indeed the Choose rule is derivable in two ways.

$$\dfrac{\dfrac{\dfrac{P\vdash\Gamma}{x().P\vdash\Gamma,x:\bot}\qquad\dfrac{Q\vdash\Gamma}{x().Q\vdash\Gamma,x:\bot}}{\mathsf{case}\,x.\{x().P;x().Q\}\vdash\Gamma,x:\bot\,\&\,\bot}\qquad\dfrac{\dfrac{}{x[].0\vdash x:1}}{x[in_1].x[].0\vdash x:1\oplus 1}}{\nu x\,(\mathsf{case}\,x.\{x().P;x().Q\}\mid x[in_1].x[].0)\vdash\Gamma}$$

$$\dfrac{\dfrac{\dfrac{P\vdash\Gamma}{x().P\vdash\Gamma,x:\bot}\qquad\dfrac{Q\vdash\Gamma}{x().Q\vdash\Gamma,x:\bot}}{\mathsf{case}\,x.\{x().P;x().Q\}\vdash\Gamma,x:\bot\,\&\,\bot}\qquad\dfrac{\dfrac{}{x[].0\vdash x:1}}{x[in_2].x[].0\vdash x:1\oplus 1}}{\nu x\,(\mathsf{case}\,x.\{x().P;x().Q\}\mid x[in_2].x[].0)\vdash\Gamma}$$

The proofs are not cut-free, and they both fail to capture the semantics of nondeterminism: the first always reduces to P and the second to Q.

We do not need to add any features to CP in order to prove that $0 \multimap \top$. Logically 0 represents falsehood, and thus implies everything including $\top$. However, there are two distinct cut-free proofs of $0 \multimap \top$.

$$\frac{}{\mathsf{case}\,x.\{\}\vdash x:\top,y:\top}\qquad\qquad\frac{}{\mathsf{case}\,y.\{\}\vdash x:\top,y:\top}$$

We can combine these using nondeterminism.

$$\mathsf{case}\,x.\{\}+\mathsf{case}\,y.\{\}\vdash x:\top,y:\top$$

Similarly, there are two distinct cut-free proofs of $A\,\&\,B \multimap A\oplus B$ in plain CP.

$$\dfrac{\dfrac{\dfrac{}{x\leftrightarrow y\vdash A^{\perp},A}}{y[in_1].x\leftrightarrow y\vdash x:A^{\perp},y:A\oplus B}}{x[in_1].y[in_1].x\leftrightarrow y\vdash x:A^{\perp}\oplus B^{\perp},y:A\oplus B}$$

$$\dfrac{\dfrac{\dfrac{}{x\leftrightarrow y\vdash x:B^{\perp},y:B}}{y[in_2].x\leftrightarrow y\vdash x:B^{\perp},y:A\oplus B}}{x[in_2].y[in_2].x\leftrightarrow y\vdash x:A^{\perp}\oplus B^{\perp},y:A\oplus B}$$

Again, we can combine these using nondeterminism.

$$(x[in_1].y[in_1].x\leftrightarrow y)+(x[in_2].y[in_2].x\leftrightarrow y)\vdash x:A^{\perp}\oplus B^{\perp},y:A\oplus B$$

Reading CP as a logic, the Fail rule has obvious logical problems: it allows us to prove anything! However, that is not to say that it does not have a meaningful dynamic semantics in terms of cut reduction. Failure immediately yields a proof of $\top \multimap 0$.

$$\frac{}{\mathsf{fail} \vdash x : 0, y : 0}$$

Conversely, if we already have a proof $P_{\top 0}$ of $\top \multimap 0$, then we can derive a proof of the Fail rule.

$$\dfrac{\dfrac{}{\mathsf{case}\, x.\{\} \vdash x : \top} \qquad \dfrac{P_{\top 0} \vdash x : 0, y : 0 \qquad \dfrac{}{\mathsf{case}\, y.\{\} \vdash y : \top, \Gamma}}{\nu y\, (P_{\top 0} \mid \mathsf{case}\, y.\{\}) \vdash x : 0, \Gamma}}{\nu x\, (\mathsf{case}\, x.\{\} \mid \nu y\, (P_{\top 0} \mid \mathsf{case}\, y.\{\})) \vdash \Gamma}$$

Thus, Fail and $\top \multimap 0$ are logically equivalent and we can define $P_{\top 0}$ in terms of Fail.

$$P_{\top 0} \stackrel{\text{def}}{=} \mathsf{fail}$$

Of course, $A \oplus B \multimap A \,\&\, B$ can be proved with fail. We could simply use fail to prove this immediately, but this does not have the right dynamic semantics in terms of nondeterminism. A more satisfying proof is given by the following judgement.

$$\mathsf{case}\, x.\{\mathsf{case}\, y.\{x \leftrightarrow y; \mathsf{fail}\}; \mathsf{case}\, y.\{\mathsf{fail}; x \leftrightarrow y\}\} \vdash \Gamma$$

This judgement captures the idea that reduction can succeed if the two channels make compatible choices. In the other direction, we can prove $\vdash 0$ (and hence $\vdash \Gamma$ for any Γ) from $P_{\oplus \&} \vdash A \oplus B \multimap A \,\&\, B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x[].0 \vdash x{:}1}}{P \vdash x{:}1 \oplus 0} \qquad P_{\oplus \&} \vdash x{:}\bot \,\&\, \top, y{:}1 \,\&\, 0}{\nu x\, (P \mid P_{\oplus \&}) \vdash y : 1 \,\&\, 0} \qquad \dfrac{\dfrac{}{\mathsf{case}\, y.\{\} \vdash y{:}\top, z{:}0}}{Q \vdash y{:}\bot \oplus \top, z{:}0}}{\nu y\, (\nu x\, (x[in_1].x[].0 \mid P_{\oplus \&}) \mid y[in_2].\mathsf{case}\, y.\{\}) \vdash z{:}0}$$

where $\qquad P = x[in_1].x[].0 \qquad Q = y[in_2].\mathsf{case}\, y.\{\}$

Thus, Fail and $A \oplus B \multimap A \,\&\, B$ are logically equivalent and we choose to define the semantics of $P_{\oplus \&}$ in terms of Fail.

$$P_{\oplus \&} \stackrel{\text{def}}{=} \mathsf{case}\, x.\{\mathsf{case}\, y.\{x \leftrightarrow y; \mathsf{fail}\}; \mathsf{case}\, y.\{\mathsf{fail}; x \leftrightarrow y\}\}$$

Bicut and Fail. It turns out that Fail is also derivable from BiCut. If we have BiCut then we can define Fail as follows.

$$\mathsf{fail} \stackrel{\text{def}}{=} \nu xy\, (x \leftrightarrow y \mid \mathsf{case}\, x.\{\})$$

Semantics. Following π-calculus, we add two reduction rules.

$$\begin{array}{l} P + Q \longrightarrow_{\mathrm{C}} P \\ P + Q \longrightarrow_{\mathrm{C}} Q \end{array}$$

Nondeterminism (without BiCut) does not disturb cut-elimination. Termination and deadlock-freedom are preserved. Of course, determinism is lost. However, the determinism we have added is *local*, in the sense that while an individual process may nondeterministically evolve, this nondeterminism does not arise from two separate processes racing for some shared resource. If we are to capture interesting concurrent behaviour then it seems natural to allow for racy communication. We will see in the next section how to introduce races.

4.4 Access Points (! and ?)

Access points provide a dynamic match-making service for initiating session communication. An access point a of type $\star A$ allows an arbitrary number of channels of type A to nondeterministically accept requests from an arbitrary number of channels of type $A^\perp$. Each channel of type A is nondeterministically paired up with a matching channel of type $A^\perp$. If at any point the numbers of A channels and $A^\perp$ channels differ, then any unmatched channels block until a matching channel becomes available (which may never happen).

It is not clear to us if it is possible to conflate ! and ? in quite the same way as we did for the other type constructors. Nevertheless, we can achieve a similar effect by a slightly different route, whereby we replace ! and ? with access points.

The rules for *accepting* and *requesting* a connection through an access point replace the Bang and Query rules.

$$\textsc{Accept}\;\frac{P \vdash \Gamma, x : !A, y : A}{\star x(y).P \vdash \Gamma, x : !A} \qquad \textsc{Request}\;\frac{P \vdash \Gamma, x : ?A, y : A}{\star x[y].P \vdash \Gamma, x : ?A}$$

The two differences from the Bang and Query rules are that: in each rule x is bound in P, allowing the access point to be reused in the continuation; and there are no restrictions on Γ in Accept. The weakening and contraction capabilities are extended to propositions of the form $!A$.

$$\textsc{Weaken!}\;\frac{P \vdash \Gamma}{P \vdash \Gamma, x : !A} \qquad \textsc{Contract!}\;\frac{P \vdash \Gamma, y : !A, z : !A}{P\{x/y, x/z\} \vdash \Gamma, x : !A}$$

The original Bang and Query rules are derivable from the above rules:

$$\frac{\dfrac{P \vdash ?\Gamma, y : A}{P \vdash ?\Gamma, x : !A, y : A}}{\star x(y).P \vdash ?\Gamma, x : !A} \qquad \frac{\dfrac{P \vdash \Gamma, y : A}{P \vdash \Gamma, x : ?A, y : A}}{\star x[y].P \vdash \Gamma, x : ?A}$$

Conversely, if for all A we have proofs of $!A \multimap ?A$ and $?A \multimap !A$, then we can straightforwardly derive Accept, Request, Weaken!, and Contract!.

Access points liberate name restriction from parallel composition. We will use the following syntactic sugar.

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \nu a\,(\star a[x].x[].0 \mid \star a[x].x[].0) \\ (\nu a)P &\stackrel{\text{def}}{=} \nu a\,(P \mid 0) \\ (\nu a_1 \ldots a_n)P &\stackrel{\text{def}}{=} (\nu a) \ldots (\nu a_n)P \end{aligned}$$

In addition to the changes to the typing rules, we also add a type equation

$$\star A = !A = ?A^{\perp}$$

capturing the idea that access points play the role of both ! and ?. This equation immediately implies that $!A \multimap ?A^{\perp}$ and $?A \multimap !A^{\perp}$, which does not quite fit with our previous pattern. However, if it is the case that A and all of its subformulas are self-dual then we can show that $!A \multimap ?A$ and $?A \multimap !A$ are admissible, as we might expect. As it turns out, we can encode Mix, MultiCut, and Choice using access points, so indeed these properties are admissible and all types are self-dual.

$$\begin{aligned} P \mid Q &\stackrel{\text{def}}{=} \nu a\,(P \mid Q) \\ \nu x_1 \ldots x_n\,(P \mid Q) &\stackrel{\text{def}}{=} (\nu a_1 \ldots a_n)(\star a_1(x_1).\cdots\star a_n(x_n).P \mid \star a_n[x_1].\cdots\star a_n[x_n].Q) \end{aligned}$$

$$\begin{aligned} \mathsf{fail} &\stackrel{\text{def}}{=} \nu x\,(\mathsf{case}\,x.\{\} \mid (\nu a)\star a(y).x \leftrightarrow y) \\ P + Q &\stackrel{\text{def}}{=} \nu a\,((\star a(x).x[in_1].x[].0 \mid \star a(x).x[in_2].x[].0) \mid \star a[x].\mathsf{case}\,x.\{x().P; x().Q\}) \end{aligned}$$

Semantics. In the presence of access points, it makes sense to add a garbage collection rule to the structural congruence to dispense with unused cut variables.

$$\nu x\vec{x}\,(P \mid Q) \equiv \nu\vec{x}\,(P \mid Q), \quad \text{if } x \notin fv(P) \cup fv(Q)$$

The reduction for ! against ? need not operate across the cut at which the access point is bound, and the channel is not discarded as the access point can be used an arbitrary number of times.

$$\nu\vec{x}\,(\star a(x).P \mid \star a[x].Q) \longrightarrow_{\mathrm{C}} \nu\vec{x}x\,(P \mid Q)$$

In order to give a closed system of reduction rules we assume MultiCut. We disable the structural congruence rules for explicitly performing replication and garbage collection of ! channels, as these are no longer necessary due to weakening and contraction on !.

The commuting conversions for access points are unsurprising.

$$\begin{aligned} \nu\vec{z}\,(\star x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} \star x(y).\nu\vec{z}\,(P \mid Q) \\ \nu\vec{z}\,(\star x[y].P \mid Q) &\longrightarrow_{\mathrm{CC}} \star x[y].\nu\vec{z}\,(P \mid Q) \end{aligned}$$

CP extended with access points does not enjoy termination, deadlock-freedom, or determinism. In return for losing these properties, access points significantly increase the expressiveness of CP. Indeed, the stateful nondeterminism inherent in the semantics of access points also admits a meaningful notion of racy communication. A critical feature that CP lacks, even if we add multicut and nondeterminism, is any form of concurrent shared state. For instance, it is not possible to represent the standard session typing example of a book seller in CP in such a way that different buyers can observe any information about which books have been sold by the book seller to other buyers. Plain replication only allows us to copy the entire book seller. Access points do allow us to implement such examples.

Example: State. We implement a state cell of type A as an access point of type $\star A$.

$$\begin{aligned} \mathit{State}\,A &\overset{\text{def}}{=} \star A \\ \mathit{new}(v).(\nu r)P &\overset{\text{def}}{=} (\nu r)(\star r(x).x \leftrightarrow v \mid P) \\ \mathit{put}(r,v).P &\overset{\text{def}}{=} \star r[v'].(\star r(x).x \leftrightarrow v \mid P) \\ \mathit{get}(r).(\nu v)P &\overset{\text{def}}{=} \star r[v].(\star r(x).x \leftrightarrow v \mid P) \end{aligned}$$

The *new* operation allocates a new reference cell (r) initialised to the supplied value. The *put* operation discards the existing state (v'). The *get* operation duplicates the existing state (v).

Using this encoding of state we can construct Landin's knot (Landin, 1964), which in sequential languages shows how higher-order state entails recursion. In CP, this provides a way of implementing recursive and replicated processes. For example, a nonterminating process can be defined by

$$\mathit{forever} \overset{\text{def}}{=} \nu f\ (\mathit{id}(f) \mid \mathit{new}(f).(\nu r)\nu g\ (\mathit{suspend}(g, \mathit{GF}(r)) \mid \mathit{put}(r,g).\mathit{GF}(r)))$$

where

$$\begin{aligned} \mathit{suspend}(f,P) &\overset{\text{def}}{=} \star f(x).P \\ \mathit{force}(f) &\overset{\text{def}}{=} \star f[x].0 \\ \mathit{id}(f) &\overset{\text{def}}{=} \mathit{suspend}(f,0) \\ \mathit{GF}(r) &\overset{\text{def}}{=} \mathit{get}(r).(\nu f)\mathit{force}(f) \end{aligned}$$

The contents of the reference cell has type $\star(\star A)$, where A can by any type.

More usefully, we can implement replication in the same way as *forever*, except the replicated process is placed in parallel with the recursive call represented by the body of the suspension.

$$\begin{aligned} !x(y).P \overset{\text{def}}{=} \nu f\ (\mathit{id}(f) \mid \mathit{new}(f).(\nu r)\nu g\,(&\,\mathit{suspend}(g, (\mathit{GF}(r) \mid \star x(y).P)) \\ &\mid \mathit{put}(r,g).\mathit{GF}(r))) \end{aligned}$$

Example: Phil's Bookstore. As a simple example of using access points to capture concurrent state, we model a scenario in which Phil has three books: two copies of *Introduction to Functional Programming* (Bird and Wadler, 1988) and one copy of *Java Generics and Collections* (Naftalin and Wadler, 2006), which he is willing to give to his students.

$$\begin{aligned}
Phil &\overset{\text{def}}{=} !phil(x).\mathsf{case}\ x.\{FP \to \star fp[y].x \leftrightarrow y; JG \to \star jg[y].x \leftrightarrow y\} \\
Fp &\overset{\text{def}}{=} \star fp(x).x[Yes].\star fp(x).x[Yes].\star fp(x).x[No].0 \\
Jg &\overset{\text{def}}{=} \star jg(x).x[Yes].\star jg(x).x[No].0
\end{aligned}$$

For clarity, we tag the sums, as we did for booleans in (Sect. 3). The *Phil* process accepts communications on the *phil* channel, dispatching requests for *Java Generics and Collections* to the *Jg* process and requests for *Introduction to Functional Programming* to the *Fp* process. The types of the access point channels are as follows.

$$\begin{aligned}
phil &: \star(JG : (Yes : \star 0 \oplus No : \star 0) \,\&\, FP : (Yes : \star 0 \oplus No : \star 0)\} \\
fp &: \star(Yes : \star 0 \oplus No : \star 0) \\
jg &: \star(Yes : \star 0 \oplus No : \star 0)
\end{aligned}$$

We exploit the type $\star 0$ to allow processes to be implicitly terminated. Now let us define processes to represent Phil's PhD students.

$$\begin{aligned}
Jack &\overset{\text{def}}{=} \star phil[x].x[JG].\mathsf{case}\ x.\{Yes \to JackHappy; No \to JackSad\} \\
Jakub &\overset{\text{def}}{=} \star phil[x].x[JG].\mathsf{case}\ x.\{Yes \to JakubHappy; No \to JakubSad\} \\
Shayan &\overset{\text{def}}{=} \star phil[x].x[FP].\mathsf{case}\ x.\{Yes \to ShayanHappy; No \to ShayanSad\} \\
Simon &\overset{\text{def}}{=} \star phil[x].x[FP].\mathsf{case}\ x.\{Yes \to SimonHappy; No \to SimonSad\}
\end{aligned}$$

Jack and Jakub request *Java Generics and Collections.* Shayan and Simon request *Introduction to Functional Programming.* We can compose all of these processes in parallel.

$$(Phil \mid Jg \mid Fp) \mid Jack \mid Jakub \mid Shayan \mid Simon$$

Reduction will nondeterministically assign each book to a student, until at some point Phil runs out of the *Java Generics and Collections* book and either Jack or Jakub has a request denied, having to go to the library instead. Note that the choice of whether Jack or Jakub is sad is not made locally by any of the *Jack*, *Jakub* or *Phil* processes, but as a result of the racy interaction between them. Without access points it is not possible in CP to represent this kind of stateful and racy interaction.

Do We need Access Points? In classical linear logic, and hence CP, weakening and contraction are derivable for each type built up from the "negative" connectives ($\parr$, $\bot$, $\&$, $\top$, $?$). Once we conflate dual connectives, weakening and contraction also become derivable for each type built up from corresponding the positive connectives as well.

Given that conflating duals allows us to define weakening and contraction at all types, even in the multiplicative additive fragment of CP (without ! and ?), one might suspect that this would result in the effective identification of ! and ?, and hence we should be able to define unlimited state without using access points. This suggests a contradiction, because we claim that the resulting calculus is terminating.

In fact, we can simulate a form of unlimited state in this manner, but with a caveat. The implementation of weakening and contraction in this encoding is explicit. Correspondingly, the size of the definition of each operation is at least linear in the number of times the reference cell is accessed. Non-terminating programs written using Landin's knot necessarily access the same reference cell an infinite number of times. Hence, a process (without access points) encoding such a program would necessarily be infinite. Thus, we cannot encode full higher-order state and Landin's knot without access points.

5 Conclusions and Future Work

We have explored extensions to Wadler's CP calculus that make it more expressive, inspired syntactically by session types and the π-calculus and semantically by Abramsky et al.'s canonical interaction category **SProc**. In doing so, we have discovered an unexpected coincidence: that adding π-calculus terms "missing" from CP allows us to realise the identification of dual types present in **SProc**, while introducing bi-implications between the dual types allows us to derive the logical rules corresponding to the missing terms. Our approach seems to cover much of the expressivity gap between CP and π-calculus, including nondeterminism, concurrent state, and recursion. We conclude by sketching several future directions suggested by this work.

SProc has more structure than that necessary to express linear logic; in particular, individual **SProc** processes may have internal notions of state, and thus changing behaviour, without changing types. In contrast, CP (and other linear logic-inspired process calculi) capture all state explicitly in the types, and each copy of a replicated process always behave identically. This poses two questions. First, can an approach similar to interaction categories be adapted to describe channel passing behaviour? Second, how do notions of state, as encoded using access points (i.e., the identification of the dual access points) correspond to the notion of state internal to an **SProc** process?

Access points are rather a blunt tool. While they do yield concurrent state, they also destroy many of the nice meta-theoretic properties of CP. We believe that it should be possible to distil better behaved abstractions that still provide some of the extra expressiveness of access points.

While we believe that the extended calculus is sufficient to encode the (typed) π-calculus, we think it would be valuable to demonstrate such a semantics-preserving encoding explicitly. We would also like to address quantification, both first-order (corresponding to value-passing calculi) and second-order (corresponding to polymorphic channel-passing calculi).

In recent work (Lindley and Morris, 2015), we demonstrate a tight correspondence between cut-reduction in CP and a small-step operational semantics for GV, a linear lambda-calculus extended with primitives for session-typing. It would be interesting to extend that correspondence to include additional features such as access points. Moreover, we are also studying well-founded recursive and corecursive session types in CP and their relationship to inductive linear data types in GV. We conjecture that conflating well-founded recursive and corecursive session types in CP will yield non-well-founded recursive session types and hence a way of encoding the entirety of the untyped π-calculus through a universal session type.

Acknowledgements. This work was supported by the EPSRC grant: *From Data Types to Session Types—A Basis for Concurrency and Distribution* (EP/K034413/1).

References

Abramsky, S.: Proofs as processes. Theor. Comput. Sci. **135**(1), 5–9 (1992)
Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany, pp. 35–113 (1996)
Bellin, G., Scott, P.J.: On the π-Calculus and linear logic. Theoret. Comput. Sci. **135**(1), 11–65 (1994)
Bird, R., Wadler, P.: An Introduction to Functional Programming. Prentice Hall International (UK) Ltd., Hertfordshire (1988)
Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
Curry, H.B.: Functionality in combinatory logic. Proc. Nat. Acad. Sci. **20**, 584–590 (1934)
Dardha, O., Pérez, J.A.: Comparing deadlock-free session typed processes. In: EXPRESS/SOS, 2015, Madrid, Spain, 31 August 2015, pp. 1–15 (2015)
Ehrhard, T., Laurent, O.: Interpreting a finitary pi-calculus in differential interaction nets. Inf. Comput. **208**(6), 606–633 (2010). http://dx.org/10.1016/j.ic.2009.06.005
Fiore, M.P.: Differential structure in models of multiplicative biadditive intuitionistic linear logic. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 163–177. Springer, Heidelberg (2007)
Giachino, E., Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 63–77. Springer, Heidelberg (2014)
Girard, J.Y.: Linear logic. Theoret. Comput. Sci. **50**(1), 1–101 (1987)
Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715. Springer, Heidelberg (1993)
Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. **411**(22–24), 2223–2238 (2010). http://dx.doi.org/10.1016/j.tcs.2010.01.028
Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

Honda, K., Yoshida, N., Berger, M.: Control in the π-calculus. In: Fourth ACM-SIGPLAN Continuation Workshop, CW04, 2004. Online proceedings (2004)

Houston, R.: Finite products are biproducts in a compact closed category. J. Pure Appl. Algebra **212**(2), 394–400 (2008). http://www.sciencedirect.com/science/article/pii/S0022404907001454

Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, Boston (1980)

Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006). http://dx.doi.org/10.1007/11817949_16

Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the π-calculus. In: POPL. ACM (1996)

Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999)

Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964)

Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 560–584. Springer, Heidelberg (2015)

Mazza, D.: The true concurrency of differential interaction nets. Mathematical Structures in Computer Science (2015, to appear)

Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly Media, Inc., Sebastopol (2006)

Padovani, L.: Deadlock and lock freedom in the linear π-calculus. In: LICS. ACM (2014). http://doi.acm.org/10.1145/2603088.2603116

Tait, W.W.: Infinitely long terms of transfinite type. In: Crossley, J.N., Dummett, M.A.E. (eds.) Formal Systems and Recursive Functions. North-Holland, Amsterdam (1965)

Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817. Springer, Heidelberg (1994)

Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2–3), 384–418 (2014)

Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). http://doi.acm.org/10.1145/2699407

Counting Successes: Effects and Transformations for Non-deterministic Programs

Nick Benton[1(✉)], Andrew Kennedy[2], Martin Hofmann[3], and Vivek Nigam[4]

[1] Microsoft Research, Cambridge, UK
nick@microsoft.com
[2] Facebook, London, UK
[3] Ludwig-Maximilians-Universität, München, Germany
[4] UFPB, Joao Pessoa, Brazil

Abstract. We give a simple effect system for non-deterministic programs, tracking static approximations to the number of results that may be produced by each computation. A relational semantics for the effect system establishes the soundness of both the analysis and its use in effect-based program transformations.

Dedicated to Philip Wadler on the occasion of his 60th *birthday.*

1 Introduction

Back in 1998, Wadler showed [30,31] how type-and-effect systems, a form of static analysis for impure programs that was first introduced by Gifford and Lucassen [15,21], may be re-presented, both syntactically and algorithmically, in terms of a variant of Moggi's computational metalanguage [22] in which the computation type constructor is refined by annotations that delimit the possible effects of computations. The same year, Tolmach described the use of a hierarchy of monadic types in optimizing compilation [26] and, in the same conference as Wadler's paper, two of us presented an optimizing compiler for Standard ML that used the same kind of refined monadic metalanguage as its intermediate language, inferring state, exception and divergence effects to enable optimizing transformations [9].

"That's all very well in practice," we thought, "but how does it work out in theory?" But devising a satisfactory semantics for effect-refined types that both interprets them as properties of the original, un-refined terms, and validates program transformations predicated on effect information proved surprisingly tricky, until we adopted another Wadleresque idea [29]: the relational interpretation of types. Interpreting static analyses in terms of binary relations, rather than unary predicates, deals naturally with independence properties (such as secure information flow or not reading parts of the store), is naturally extensional (by contrast with, say, instrumenting the semantics with a trace of side-effecting operations), and accounts for the soundness of program transformations at the same time as soundness of the analysis [2]. We have studied a series of effect systems, of

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 56–72, 2016.
DOI: 10.1007/978-3-319-30936-1_3

ever-increasing sophistication, using relations, concentrating mainly on tracking uses of mutable state [4,7,8].

Here we consider a different effect: non-determinism. Wadler studied non-determinism in a famous thirty-year-old paper on how lazy lists can be used to program exception handling, backtracking and pattern matching in pure functional languages [28], and returned to it in his work on query languages [23]. That initial paper draws a distinction between two cases. The first is the use of lists to encode errors, or exceptions, where computations either fail, represented by the empty list, or succeed, returning a singleton list. The second is more general backtracking, encoding the kind of search found in logic programming languages, where computations can return many results. This paper is in the spirit of formalizing that distinction. We refine a non-determinism monad with effect annotations that approximate how many (different) results may be returned by each computation, and give a semantics that validates transformations that depend on that information. To keep everything as simple as possible, we work with a total language and a semantics that uses powersets, rather than lists or multisets, so we do not observe the order or multiplicity of results. The basic ideas could, however, easily be adapted to a language with recursion or a semantics with lists instead of sets.

2 Effects for Non-determinism

2.1 Base Language

We consider a monadically-typed, normalizing, call-by-value lambda calculus with operations for failure and non-deterministic choice. A more conventionally-typed impure calculus may be translated into the monadic one via the usual 'call-by-value translation' [6], and this extends to the usual style of presenting effect systems in which every judgement has an effect, and function arrows are annotated with 'latent effects' [31].

We define value types A, computation types TA and contexts Γ as follows:

$$\begin{aligned} A, B &:= \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid A \times B \mid A \to TB \\ \Gamma &:= x_1 : A_1, \ldots, x_n : A_n \end{aligned}$$

Value judgements, $\Gamma \vdash V : A$, and computation judgements, $\Gamma \vdash M : TA$, are defined by the rules in Fig. 1. The presence of types on lambda-bound variables makes typing derivations unique, and addition and comparison should be considered just representative primitive operations.

Our simple language has an elementary denotational semantics in the category of sets and functions. The semantics of types is as follows:

$$[\![\mathtt{unit}]\!] = 1 \qquad [\![\mathtt{int}]\!] = \mathbb{Z} \qquad [\![\mathtt{bool}]\!] = \mathbb{B} \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$
$$[\![A \to TB]\!] = [\![A]\!] \to [\![TB]\!] \qquad [\![TA]\!] = \mathbb{P}_{\mathit{fin}}([\![A]\!])$$

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 + V_2 : \mathtt{int}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 > V_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i\, V : A_i}$$

$$\frac{\Gamma, x : A \vdash M : TB}{\Gamma \vdash \lambda x : A.M : A \to TB} \qquad \frac{\Gamma \vdash V_1 : A \to TB \quad \Gamma \vdash V_2 : A}{\Gamma \vdash V_1\, V_2 : TB} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{val}\, V : TA}$$

$$\frac{\Gamma \vdash M : TA \quad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathtt{let}\, x \Leftarrow M \,\mathtt{in}\, N : TB} \qquad \frac{\Gamma \vdash V : \mathtt{bool} \quad \Gamma \vdash M : TA \quad \Gamma \vdash N : TA}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : TA}$$

$$\frac{}{\Gamma \vdash \mathtt{fail} : TA} \qquad \frac{\Gamma \vdash M_1 : TA \quad \Gamma \vdash M_2 : TA}{\Gamma \vdash M_1\, \mathtt{or}\, M_2 : TA}$$

Fig. 1. Simple computation type system

The interpretation of the computation type constructor is the finite powerset monad. The meaning of contexts is given by $[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$, and we can then give the semantics of judgements

$$[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \qquad \text{and} \qquad [\![\Gamma \vdash M : TA]\!] : [\![\Gamma]\!] \to [\![TA]\!]$$

inductively in a standard way. The interesting cases are

$$\begin{aligned}
[\![\Gamma \vdash \mathtt{val}\, V : TA]\!]\, \rho &= \{[\![\Gamma \vdash V : A]\!]\, \rho\} \\
[\![\Gamma \vdash \mathtt{let}\, x \Leftarrow M \,\mathtt{in}\, N]\!]\, \rho &= \textstyle\bigcup_{v \in [\![\Gamma \vdash M : A]\!]\, \rho} [\![\Gamma, x : A \vdash N : TB]\!]\, (\rho, v) \\
[\![\Gamma \vdash \mathtt{fail} : TA]\!]\, \rho &= \emptyset \\
[\![\Gamma \vdash M_1\, \mathtt{or}\, M_2 : TA]\!]\, \rho &= ([\![\Gamma \vdash M_1 : TA]\!]\, \rho) \cup ([\![\Gamma \vdash M_1 : TA]\!]\, \rho)
\end{aligned}$$

So, for example

$$\begin{aligned}
&[\![\vdash \mathtt{let}\, f \Leftarrow \mathtt{val}\, (\lambda x : \mathtt{int}.\mathtt{if}\ x < 6\ \mathtt{then}\ \mathtt{val}\, x\ \mathtt{else}\ \mathtt{fail})\ \mathtt{in} \\
&\quad \mathtt{let}\, x \Leftarrow \mathtt{val}\, 1\, \mathtt{or}\, \mathtt{val}\, 2\ \mathtt{in}\ \mathtt{let}\, y \Leftarrow \mathtt{val}\, 3\, \mathtt{or}\, \mathtt{val}\, 4\ \mathtt{in}\ f(x+y) : T\mathtt{int}]\!] = \{4, 5\}
\end{aligned}$$

The semantics is adequate for the obvious operational semantics and a contextual equivalence observing, say, the set of unit values produced by a closed program.

2.2 Effect System

We now present an effect analysis that refines the simple type system by annotating the computation type constructor with information about *how many results* a

$$\frac{}{X \le X} \qquad \frac{X \le Y \quad Y \le Z}{X \le Z} \qquad \frac{X \le X' \quad Y \le Y'}{X \times Y \le X' \times Y'}$$

$$\frac{X' \le X \quad T_\varepsilon Y \le T_{\varepsilon'} Y'}{(X \to T_\varepsilon Y) \le (X' \to T_{\varepsilon'} Y')} \qquad \frac{\varepsilon \le \varepsilon' \quad X \le X'}{T_\varepsilon X \le T_{\varepsilon'} X'}$$

Fig. 2. Subtyping refined types

computation may produce. Formally, define *refined* value types X, computation types $T_\varepsilon X$ and contexts Θ by

$$\begin{aligned} X, Y &:= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid X \times Y \mid X \to T_\varepsilon Y \\ \varepsilon &\in \{\texttt{0}, \texttt{1}, \texttt{01}, \texttt{1+}, \texttt{N}\} \\ \Theta &:= x_1 : X_1, \ldots, x_n : X_n \end{aligned}$$

A computation of type $T_0 X$ will always fail, i.e. produce zero results. One of type $T_1 X$ is deterministic, i.e. produces exactly one result. More generally, writing $|S|$ for the cardinality of a finite set S, a computation of type $T_\varepsilon X$ can only produce sets of results S such that $|S| \in [\![\varepsilon]\!]$, where $[\![\varepsilon]\!] \subseteq \mathbb{N}$:

$$\begin{aligned} [\![\texttt{0}]\!] &= \{0\} & [\![\texttt{1+}]\!] &= \{n \mid n \ge 1\} \\ [\![\texttt{1}]\!] &= \{1\} & [\![\texttt{N}]\!] &= \mathbb{N} \\ [\![\texttt{01}]\!] &= \{0, 1\} \end{aligned}$$

There is an obvious order on effect annotations, given by $\varepsilon \le \varepsilon' \iff [\![\varepsilon]\!] \subseteq [\![\varepsilon']\!]$:

$$\begin{tikzcd} & & \texttt{N} \arrow[dl, dash] \arrow[dr, dash] & \\ & \texttt{01} \arrow[dl, dash] \arrow[dr, dash] & & \texttt{1+} \arrow[dl, dash] \\ \texttt{0} & & \texttt{1} & \end{tikzcd}$$

This order induces a subtyping relation on refined types, which is axiomatised in Fig. 2. The refined type assignment system is shown in Fig. 3. The erasure map, $U(\cdot)$, takes refined types to simple ones by forgetting the effect annotations:

$$\begin{gathered} U(\texttt{int}) = \texttt{int} \qquad U(\texttt{bool}) = \texttt{bool} \qquad U(\texttt{unit}) = \texttt{unit} \\ U(X \times Y) = U(X) \times U(Y) \\ U(X \to T_\varepsilon Y) = U(X) \to U(T_\varepsilon Y) \\ U(T_\varepsilon X) = T(U(X)) \end{gathered}$$

$$U(x_1 : X_1, \ldots, x_n : X_n) = x_1 : U(X_1), \ldots, x_n : U(X_n)$$

Lemma 1. *If $X \le Y$ then $U(X) = U(Y)$, and similarly for computations.* □

The use of erasure on bound variables means that the subject terms of the refined type system are the same as those of the unrefined one.

$$\frac{}{\Theta \vdash n : \texttt{int}} \qquad \frac{}{\Theta \vdash b : \texttt{bool}} \qquad \frac{}{\Theta \vdash () : \texttt{unit}} \qquad \frac{}{\Theta, x : X \vdash x : X}$$

$$\frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 + V_2 : \texttt{int}} \qquad \frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 > V_2 : \texttt{bool}}$$

$$\frac{\Theta \vdash V_1 : X \quad \Theta \vdash V_2 : Y}{\Theta \vdash (V_1, V_2) : X \times Y} \qquad \frac{\Theta \vdash V : X_1 \times X_2}{\Theta \vdash \pi_i\, V : X_i} \qquad \frac{\Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon Y}$$

$$\frac{\Theta \vdash V_1 : X \to T_\varepsilon Y \quad \Theta \vdash V_2 : X}{\Theta \vdash V_1\, V_2 : T_\varepsilon Y} \qquad \frac{\Theta \vdash V : X}{\Theta \vdash \texttt{val}\ V : T_1 X}$$

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N : T_{\varepsilon \cdot \varepsilon'} Y} \qquad \frac{\Theta \vdash V : \texttt{bool} \quad \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon X}{\Theta \vdash \texttt{if}\ V\ \texttt{then}\ M\ \texttt{else}\ N : T_\varepsilon X}$$

$$\frac{}{\Theta \vdash \texttt{fail} : T_0 X} \qquad \frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X}{\Theta \vdash M_1\ \texttt{or}\ M_2 : T_{\varepsilon_1 + \varepsilon_2} X}$$

$$\frac{\Theta \vdash V : X \quad X \leq X'}{\Theta \vdash V : X'} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \leq T_{\varepsilon'} X'}{\Theta \vdash M : T_{\varepsilon'} X'}$$

Fig. 3. Refined type system

Lemma 2. *If $\Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$, and similarly for computations.* □

It is also the case that the refined system does not rule out any terms from the original language. Let $G(\cdot)$ be the map from simple types to refined types that adds the 'top' effect N to all computation types; we can then show the following:

Lemma 3. *If $\Gamma \vdash V : A$ then $G(\Gamma) \vdash V : G(A)$ and similarly for computations.* □

The interesting aspect of the refined type system is the use it makes of abstract multiplication (in the `let`-rule) and addition (in the `or` rule) operations on effects. The definitions are:

·	0	1	01	1+	N
0	0	0	0	0	0
1	0	1	01	1+	N
01	0	01	01	N	N
1+	0	1+	N	1+	N
N	0	N	N	N	N

+	0	1	01	1+	N
0	0	1	01	1+	N
1	1	1+	1+	1+	1+
01	01	1+	N	1+	N
1+	1+	1+	1+	1+	1+
N	N	1+	N	1+	N

The operations endow our chosen set of effect annotations with the structure of a commutative semiring with idempotent multiplication.

Lemma 4. *The $+$ operation is associative and commutative, with $\mathtt{0}$ as a unit. The $\cdot$ operation is associative, commutative and idempotent, with $\mathtt{1}$ as unit and $\mathtt{0}$ as zero. We also have the distributive law $(\varepsilon_1 + \varepsilon_2) \cdot \varepsilon_3 = \varepsilon_1 \cdot \varepsilon_3 + \varepsilon_2 \cdot \varepsilon_3$.* □

The correctness statement concerning the abstract operations that we will need later is a consequence of a trivial fact about the cardinality of unions:

$$|A| \;\leq\; |A \cup B| \;\leq |A| + |B|$$

which leads to the following:

Lemma 5. *For any ε_1,ε_2,*

$$\begin{aligned}&\textstyle\bigcup_{a\in[\![\varepsilon_1]\!],\, b\in[\![\varepsilon_2]\!]}\{n \mid \mathit{max}(a,b) \leq n \;\mathit{and}\; n \leq a+b\} \;\subseteq [\![\varepsilon_1 + \varepsilon_2]\!]\\ &\textstyle\bigcup_{a\in[\![\varepsilon_1]\!]}\bigcup_{(b_1,\dots,b_a)\in[\![\varepsilon_2]\!]^a}\{n \mid \forall i, b_i \leq n \;\mathit{and}\; n \leq \Sigma_i b_i\} \subseteq [\![\varepsilon_1 \cdot \varepsilon_2]\!].\end{aligned}$$ □

The intuition behind the awkward-looking correctness condition for multiplication deserves some explanation. Consider how many results may be produced by $\mathtt{let}\, x \Leftarrow M \;\mathtt{in}\, N$ when M produces results $x_1, \dots, x_a$ for some $a \in [\![\varepsilon_1]\!]$ and for each such result, x_i, $N(x_i)$ produces a set of results of size $b_i \in [\![\varepsilon_2]\!]$. Then, by the inequality above, the number n of results of the `let`-expression is bounded below by each of the b_is and above by the sum of the b_is. The set of all possible cardinalities for the `let`-expression is then obtained by unioning the cardinality sets for each possible a and each possible tuple $(b_1, \dots, b_a)$.

The reader may also be surprised by the asymmetry of the condition for multiplication, given that we observed above that the abstract operation is commutative. But that commutativity is actually an accidental consequence of our particular choice of sets of cardinalities to track. Indeed, if M produces a single result and for each x, $N(x)$ produces exactly two results (a case we do not track here), then $\mathtt{let}\, x \Leftarrow M \;\mathtt{in}\, N$ produces two results. Conversely, however, if M produces two results and for each x, $N(x)$ produces a single result, then $\mathtt{let}\, x \Leftarrow M \;\mathtt{in}\, N$ can produce either one *or* two distinct results. This case also shows that, in general, `1` will only be left unit for multiplication. Idempotency also fails to hold in general.

We remark that the abstract operations are an example of the Cousots' $\alpha\gamma$ framework for abstract interpretation [12], and were in fact derived using a little list-of-successes ML program that computes with abstractions and concretions.

2.3 Semantics of Effects

The meanings of simple types are just sets, out of which we will carve the meanings of refined types as subsets, *together* with a coarser notion of equality.

We first recall some notation. If R is a (binary) relation on A and Q a relation on B, then we define relations on Cartesian products and function spaces by

$$\begin{aligned}R \times Q &= \{((a,b),(a',b')) \in (A \times B) \times (A \times B) \;\mid (a,a') \in R,\, (b,b') \in Q\}\\ R \rightarrow Q &= \{(f,f') \in (A \rightarrow B) \times (A \rightarrow B) \;\mid \forall (a,a') \in R.\, (f\, a,\, f'\, a') \in Q\}\end{aligned}$$

A binary relation on a set is a *partial equivalence relation* (PER) if it is symmetric and transitive. If R and Q are PERs, so are $R \to Q$ and $R \times Q$. Write Δ_A for the diagonal relation $\{(a,a) \mid a \in A\}$, and $a : R$ for $(a,a) \in R$. If R is a PER on A and $a \in A$ then we define the 'equivalence class' $[a]_R$ to be $\{a' \in A | (a,a') \in R\}$, noting that this is empty unless $a : R$.

We can now define the semantics of each refined type as a partial equivalence relation on the semantics of its erasure as follows:

$$\begin{aligned}
[\![X]\!] &\subseteq [\![U(X)]\!] \times [\![U(X)]\!] \\
[\![\mathtt{int}]\!] = \Delta_{\mathbb{Z}} \qquad [\![\mathtt{bool}]\!] &= \Delta_{\mathbb{B}} \qquad [\![\mathtt{unit}]\!] = \Delta_1 \\
[\![X \times Y]\!] &= [\![X]\!] \times [\![Y]\!] \\
[\![X \to T_\varepsilon Y]\!] &= [\![X]\!] \to [\![T_\varepsilon Y]\!] \\
[\![T_\varepsilon X]\!] &= \{(S,S') \mid S \sim_X S' \text{ and } |S/[\![X]\!]| \in [\![\varepsilon]\!]\}
\end{aligned}$$

The key clause is the last one, in which $S \sim_X S'$ means $\forall x \in S, \exists x' \in S', (x,x') \in [\![X]\!]$ and vice versa. The $\sim_X$ relation is a lifting of $[\![X]\!]$ to sets of values; this is a familar, canonical construction that appears, for example, in the definition of bisimulation or of powerdomains. The quotient $S/[\![X]\!]$ is defined to be $\{[x]_{[\![X]\!]} \mid x \in S\}$.[1] We observe that if $S \sim_X S'$ then $S/[\![X]\!] = S'/[\![X]\!]$ and $\emptyset \notin S/[\![X]\!]$.

The way one should understand the clause for computation types is that two sets S,S' are related when they have the same elements *up to* the notion of equivalence associated with the refined type X and, moreover, the cardinality of the sets (again, as sets of Xs, *not* as sets of the underlying type UX) is accurately reflected by ε.

We also extend the relational interpretation of refined types to refined contexts in the natural way:

$$\begin{aligned}
[\![\Theta]\!] &\subseteq [\![U(\Theta)]\!] \times [\![U(\Theta)]\!] \\
[\![x_1 : X_1, \ldots, x_n : X_n]\!] &= [\![X_1]\!] \times \cdots \times [\![X_n]\!]
\end{aligned}$$

Lemma 6. *For any Θ, X and ε, all of $[\![\Theta]\!]$, $[\![X]\!]$ and $[\![T_\varepsilon X]\!]$ are partial equivalence relations.* □

The interpretation of a refined type with the top effect annotation everywhere is just equality on the interpretation of its erasure:

Lemma 7. *For all A, $[\![G(A)]\!] = \Delta_{[\![A]\!]}$.* □

The following establishes semantic soundness for our subtyping relation:

Lemma 8. *If $X \le Y$ then $[\![X]\!] \subseteq [\![Y]\!]$, and similarly for computation types.* □

And we can then show the 'fundamental theorem' that establishes the soundness of the effect system itself:

[1] It is tempting to replace $S \sim_X S'$ by $S/[\![X]\!] = S'/[\![X]\!]$, but $S/[\![X]\!]$ contains the empty set when there is an $x \in S$ with $(x,x) \notin [\![X]\!]$.

Theorem 1.

1. *If* $\Theta \vdash V : X$, $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V : U(X)]\!]\,\rho') \in [\![X]\!]$$

2. *If* $\Theta \vdash M : T_\varepsilon X$, $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho, [\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho')\ \in [\![T_\varepsilon X]\!]$$

Proof. A largely standard induction; we just sketch the interesting cases.

Trivial Computations. Let $\Gamma = U(\Theta)$ and $A = U(X)$. Given $(\rho, \rho') \in [\![\Theta]\!]$ we need to show

$$([\![\Gamma \vdash \texttt{val}\, V : TA]\!]\,\rho,\ [\![\Gamma \vdash \texttt{val}\, V : TA]\!]\,\rho') \in [\![T_1 X]\!]$$

which means

$$(\{[\![\Gamma \vdash V : A]\!]\rho\},\ \{[\![\Gamma \vdash V : A]\!]\rho'\}) \in \{(S, S') \mid S \sim_X S' \text{ and } |S/[\![X]\!]| = 1\}$$

Induction gives $([\![\Gamma \vdash V : A]\!]\rho, [\![\Gamma \vdash V : A]\!]\rho') \in [\![X]\!]$, which deals with the $\cdot \sim_X \cdot$ condition, and it is clear that $|\{[[\![\Gamma \vdash V : A]\!]]_{[\![X]\!]}\}| = 1$.

Choice. We want firstly that

$$[\![\Gamma \vdash M_1\, \texttt{or}\, M_2 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1\, \texttt{or}\, M_2 : TA]\!]\rho'$$

which is

$$[\![\Gamma \vdash M_1 : TA]\!]\rho \cup [\![\Gamma \vdash M_2 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1 : TA]\!]\rho' \cup [\![\Gamma \vdash M_2 : TA]\!]\rho'$$

Induction gives $[\![\Gamma \vdash M_1 : TA]\!]\rho \sim_X [\![\Gamma \vdash M_1 : TA]\!]\rho'$ and similarly for M_2, from which the result is immediate. Secondly, we want

$$|[\![\Gamma \vdash M_1\, \texttt{or}\, M_2 : TA]\!]\rho\ /\ [\![X]\!]| \in [\![\varepsilon_1 + \varepsilon_2]\!]$$

and because quotient distributes over union, this is

$$|[\![\Gamma \vdash M_1 : TA]\!]\rho\,/[\![X]\!]\ \cup\ [\![\Gamma \vdash M_2 : TA]\!]\rho\,/[\![X]\!]| \in [\![\varepsilon_1 + \varepsilon_2]\!]$$

By induction, $|[\![\Gamma \vdash M_1 : TA]\!]\rho\,/[\![X]\!]| \in [\![\varepsilon_1]\!]$, and similarly for M_2, so we are done by Lemma 5.

Sequencing. Pick $y \in [\![\Gamma \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : TB]\!]\rho$. By the semantics of `let`, there's an $x \in [\![\Gamma \vdash M : TA]\!]\rho$ such that $y \in [\![\Gamma, x : A \vdash N : TB]\!](\rho, x)$. By induction on M, there's an $x' \in [\![\Gamma \vdash M : TA]\!]\rho'$ such that $(x, x') \in [\![X]\!]$. So by induction on N, $[\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \sim_Y [\![\Gamma, x : A \vdash N : TB]\!](\rho', x')$, and therefore $\exists y' \in [\![\Gamma, x : A \vdash N : TB]\!](\rho', x')$ with $(y, y') \in [\![Y]\!]$. Then as $y' \in [\![\Gamma \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : TB]\!]\rho'$, we are done.

For the cardinality part, note that

$$\begin{aligned}
&\left(\bigcup_{x\in[\![\Gamma\vdash M:TA]\!]\rho}[\![\Gamma, x : A \vdash N : TB]\!](\rho, x)\right) /[\![Y]\!]\\
&= \bigcup_{x\in[\![\Gamma\vdash M:TA]\!]\rho} ([\![\Gamma, x : A \vdash N : TB]\!](\rho, x)/[\![Y]\!])\\
&= \bigcup_{[x]\in[\![\Gamma\vdash M:TA]\!]\rho/[\![X]\!]} ([\![\Gamma, x : A \vdash N : TB]\!](\rho, x)/[\![Y]\!])
\end{aligned}$$

and then since, by induction, $|\Gamma \vdash M : TA]\!]\rho/[\![X]\!]| \in [\![\varepsilon_1]\!]$, and also for any $[x] \in [\![\Gamma \vdash M : TA]\!]\rho/[\![X]\!]$,

$$|[\![\Gamma, x : A \vdash N : TB]\!](\rho, x)/[\![Y]\!]| \in [\![\varepsilon_2]\!]$$

we are done by Lemma 5. □

2.4 Basic Equations

The semantics validates all the generic equations of the computational metalanguage: congruence laws, β and η laws for products, function spaces, booleans and computation types. We show some of these rules in Fig. 4. The powerset monad also validates a number of more specific equations that hold without restrictions on the involved effects. These are shown in Fig. 5: choice is associative, commutative and idempotent with `fail` as a unit, the monad is commutative, and choice and failure distribute over `let`.

The correctness of the basic congruence laws subsumes Theorem 1. Note that, slightly subtly, the reflexivity PER rule is invertible. This is sound because our effect annotations are purely descriptive (*Curry-style*, or *extrinsic* in Reynolds's terminology [24]) whereas the simple types are more conventionally prescriptive (*Church-style*, which Reynolds calls *intrinsic*). We actually regard the rules of Fig. 3 as abbreviations for a subset of the equational judgements of Fig. 4; thus we can allow the refined type of the conclusion of interesting equational rules to be different from (in particular, have a smaller effect than) the rules in Fig. 3 would assign to one side. This shows up already: most of the rules in Fig. 5 are type correct in simple syntactic sense as a consequence of Lemma 4. But the idempotency rule for choice is not, because the abstract addition is, rightly, not idempotent. The idempotency law effectively extends the refined type system with a rule saying that if M has type $T_\varepsilon X$, so does M `or` M.

In practical terms, having equivalences also improve typing allows inferred effects to be improved locally as transformations are performed, rather than requiring periodic reanalysis of the whole program to obtain the best results.

3 Using Effect Information

More interesting equivalences are predicated on the effect information. We present these in Fig. 6.

The **Fail** transformation allows any computation with the `0` effect, i.e. that produces no results, to be replaced with `fail`.

PER rules (+ similar for computations):

$$\frac{\Theta \vdash V : X}{\Theta \vdash V = V : X} \qquad \frac{\Theta \vdash V = V' : X}{\Theta \vdash V' = V : X} \qquad \frac{\Theta \vdash V = V' : X \quad \Theta \vdash V' = V'' : X}{\Theta \vdash V = V'' : X}$$

$$\frac{\Theta \vdash V = V' : X \quad X \leq X'}{\Theta \vdash V = V' : X'}$$

Congruence rules (extract):

$$\frac{\Theta \vdash V_1 = V_1' : \texttt{int} \quad \Theta \vdash V_2 = V_2' : \texttt{int}}{\Theta \vdash (V_1 + V_2) = (V_1' + V_2') : \texttt{int}} \qquad \frac{\Theta \vdash V = V' : X_1 \times X_2}{\Theta \vdash \pi_i V \ = \ \pi_i V' : X_i}$$

$$\frac{\Theta, x : X \vdash M = M' : T_\varepsilon Y}{\Theta \vdash (\lambda x : U(X).M) = (\lambda x : U(X).M') : X \to T_\varepsilon Y}$$

β rules (extract):

$$\frac{\Theta, x : X \vdash M : T_\varepsilon Y \quad \Theta \vdash V : X}{\Theta \vdash (\lambda x : U(X).M)\, V \ = \ M[V/x] : T_\varepsilon Y} \qquad \frac{\Theta \vdash V : X \quad \Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \texttt{let}\, x \Leftarrow \texttt{val}\, V\, \texttt{in}\, M = M[V/x] : T_\varepsilon Y}$$

η rules (extract):

$$\frac{\Theta \vdash V : X \to T_\varepsilon Y}{\Theta \vdash V = (\lambda x : U(X).V\, x) : X \to T_\varepsilon Y} \qquad \frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash (\texttt{let}\, x \Leftarrow M\, \texttt{in}\, \texttt{val}\, x) = M : T_\varepsilon X}$$

Commuting conversions:

$$\frac{\Theta \vdash M : T_{\varepsilon_1} Y \quad \Theta, y : Y \vdash N : T_{\varepsilon_2} X \quad \Theta, x : X \vdash P : T_{\varepsilon_3} Z}{\Theta \vdash \texttt{let}\, x \Leftarrow (\texttt{let}\, y \Leftarrow M\, \texttt{in}\, N)\, \texttt{in}\, P = \texttt{let}\, y \Leftarrow M\, \texttt{in}\, \texttt{let}\, x \Leftarrow N\, \texttt{in}\, P : T_{\varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3} Z}$$

Fig. 4. Monad-independent equivalences

The **Dead Computation** transformation allows the removal of a computation, M, whose value is unused, provided the effect of M indicates that it always produces at least one result. If M can fail then its removal is generally unsound, as that could transform a failing computation into one that succeeds.

The **Duplicated Computation** transformation allows two evaluations of a computation M to be replaced by one, provided that M produces at most one result. This is, of course, generally unsound, as, for example,

$$\begin{aligned} &\texttt{let}\, x \Leftarrow \texttt{val}\, 1\, \texttt{or}\, \texttt{val}\, 2\, \texttt{in}\, \texttt{let}\, y \Leftarrow \texttt{val}\, 1\, \texttt{or}\, \texttt{val}\, 2\, \texttt{in}\, \texttt{val}\, (x + y) \\ \neq\ &\texttt{let}\, x \Leftarrow \texttt{val}\, 1\, \texttt{or}\, \texttt{val}\, 2\, \texttt{in}\, \texttt{val}\, (x + x). \end{aligned}$$

The **Pure Lambda Hoist** transformation allows a computation to be hoisted out of a lambda abstraction, so it is performed once, rather than every

Choice:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X}{\Theta \vdash M_1 \,\texttt{or}\, M_2 = M_2 \,\texttt{or}\, M_1 : T_{\varepsilon_1 + \varepsilon_2} X}$$

$$\frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash M \,\texttt{or}\, M = M : T_\varepsilon X} \qquad \frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash M \,\texttt{or}\, \texttt{fail} = M : T_\varepsilon X}$$

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X \quad \Theta \vdash M_3 : T_{\varepsilon_3} X}{\Theta \vdash M_1 \,\texttt{or}\, (M_2 \,\texttt{or}\, M_3) = (M_1 \,\texttt{or}\, M_2) \,\texttt{or}\, M_3 : T_{\varepsilon_1 + \varepsilon_2 + \varepsilon_3} X}$$

Commutativity:

$$\frac{\Theta \vdash M : T_{\varepsilon_1} Y \quad \Theta \vdash N : T_{\varepsilon_2} X \quad \Theta, x : X, y : Y \vdash P : T_{\varepsilon_3} Z}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{let}\, y \Leftarrow N \,\texttt{in}\, P = \texttt{let}\, y \Leftarrow N \,\texttt{in}\, \texttt{let}\, x \Leftarrow M \,\texttt{in}\, P : T_{\varepsilon_1 \cdot \varepsilon_2 \cdot \varepsilon_3} Z}$$

Distribution:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \quad \Theta \vdash M_2 : T_{\varepsilon_2} X \quad \Theta, x : X \vdash N : T_{\varepsilon_3} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow (M_1 \,\texttt{or}\, M_2) \,\texttt{in}\, N = (\texttt{let}\, x \Leftarrow M_1 \,\texttt{in}\, N) \,\texttt{or}\, (\texttt{let}\, x \Leftarrow M_2 \,\texttt{in}\, N) : T_{(\varepsilon_1 + \varepsilon_2) \cdot \varepsilon_3} Y}$$

$$\frac{\Theta, x : X \vdash N : T_\varepsilon Y}{\Theta \vdash \texttt{let}\, x \Leftarrow \texttt{fail} \,\texttt{in}\, N = \texttt{fail} : T_0 Y}$$

$$\frac{\Theta \vdash M : T_{\varepsilon_3} X \quad \Theta, x : X \vdash N_1 : T_{\varepsilon_1} Y \quad \Theta, x : X \vdash N_2 : T_{\varepsilon_2} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, (N_1 \,\texttt{or}\, N_2) = (\texttt{let}\, x \Leftarrow M \,\texttt{in}\, N_1) \,\texttt{or}\, (\texttt{let}\, x \Leftarrow M \,\texttt{in}\, N_2) : T_{\varepsilon_3 \cdot (\varepsilon_1 + \varepsilon_2)} Y}$$

$$\frac{\Theta \vdash M : T_\varepsilon X}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{fail} = \texttt{fail} : T_0 Y}$$

Fig. 5. Monad-specific, effect-independent equivalences

Fail:

$$\frac{\Theta \vdash M : T_0 X}{\Theta \vdash M = \texttt{fail} : T_0 X}$$

Dead Computation:

$$\frac{\Theta \vdash M : T_{1+} X \quad \Theta \vdash N : T_\varepsilon Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N = N : T_\varepsilon Y}$$

Duplicated Computation:

$$\frac{\Theta \vdash M : T_{01} X \quad \Theta, x : X, y : X \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{c} \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{let}\, y \Leftarrow M \,\texttt{in}\, N \\ = \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N[x/y] \end{array} : T_{01 \cdot \varepsilon} Y}$$

Pure Lambda Hoist:

$$\frac{\Theta \vdash M : T_1 Z \quad \Theta, x : X, y : Z \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{c} \texttt{val}\, (\lambda x : U(X).\texttt{let}\, y \Leftarrow M \,\texttt{in}\, N) \\ = \texttt{let}\, y \Leftarrow M \,\texttt{in}\, \texttt{val}\, (\lambda x : U(X).N) \end{array} : T_1(X \to T_\varepsilon Y)}$$

Fig. 6. Effect-dependent equivalences

time the function is applied, provided that it returns exactly one result (and, of course, that it does not depend on the function argument).

Theorem 2. *All of the equations shown in Figs. 4, 5, and 6 are soundly modelled in the semantics:*

- *If $\Theta \vdash V = V' : X$ then $\Theta \models V = V' : X$.*
- *If $\Theta \vdash M = M' : T_\varepsilon X$ then $\Theta \models M = M' : T_\varepsilon X$.*

Proof. We present proofs for the equivalences in Fig. 6.

Dead Computation. If we let $\Gamma = U(\Theta)$, $A = U(X)$ and $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$ then we have to show

$$([\![\Gamma \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N : TB]\!]\, \rho,\ [\![\Gamma \vdash N : TB]\!]\, \rho') \in [\![T_\varepsilon Y]\!]$$

which is

$$\begin{array}{l} \bigcup_{x \in [\![\Gamma \vdash M : TA]\!]\rho} [\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \sim_Y [\![\Gamma \vdash N : TB]\!]\rho' \quad \text{and} \\ \left| \bigcup_{x \in [\![\Gamma \vdash M : TA]\!]\rho} [\![\Gamma, x : A \vdash N : TB]\!](\rho, x) \,/\, [\![Y]\!] \right| \in [\![\varepsilon]\!]. \end{array}$$

Since for any x, $[\![\Gamma, x : A \vdash N : TB]\!](\rho, x) = [\![\Gamma \vdash N : TB]\!]\rho$, and induction on M tells us that $|[\![\Gamma \vdash M : TA]\!]\rho / [\![X]\!]| > 0$, so $|[\![\Gamma \vdash M : TA]\!]\rho| > 0$, that's just

$$[\![\Gamma \vdash N : TB]\!]\rho \sim_Y [\![\Gamma \vdash N : TB]\!]\rho' \quad \text{and} \quad |[\![\Gamma \vdash N : TB]\!]\rho / [\![Y]\!]| \in [\![\varepsilon]\!]$$

which is immediate by induction on N.

Duplicated Computation. Let $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$ and $(\rho, \rho') \in [\![\Theta]\!]$. We want (eliding contexts and types in semantic brackets to reduce clutter)

$$\begin{array}{l} \bigcup_{x \in [\![M]\!]\rho} \bigcup_{y \in [\![M]\!]\rho} [\![N]\!](\rho, x, y) \sim_Y \bigcup_{x' \in [\![M]\!]\rho'} [\![N[x/y]]\!](\rho', x') \\ \text{and} \left| \bigcup_{x \in [\![M]\!]\rho} \bigcup_{y \in [\![M]\!]\rho} [\![N]\!](\rho, x, y) \,/\, [\![Y]\!] \right| \in [\![\mathtt{01} \cdot \varepsilon]\!] \end{array}$$

Let $a = |[\![M]\!]\rho / [\![X]\!]|$. By induction, $a \in [\![\mathtt{01}]\!]$. If $a = 0$ then we must have $[\![M]\!]\rho = \emptyset$ and (also by induction) $[\![M]\!]\rho' = \emptyset$, so the first clause above is satisfied. For the second, we just have to check that $0 \in [\![\mathtt{01} \cdot \varepsilon]\!]$ for any ε, which is true.

If $a = 1$ we can pick any $x \in [\![M]\!]\rho$ and $x' \in [\![M]\!]\rho'$ and know $\forall y \in [\![M]\!]\rho, (x, y) \in [\![X]\!]$ as well as $\forall y' \in [\![M]\!]\rho', (x, y') \in [\![X]\!]$. Then by induction on N and the fact that $S \sim_Y S'$ implies $S \cup S' \sim_Y S$ we have

$$\begin{array}{rl} \bigcup_{x \in [\![M]\!]\rho} \bigcup_{y \in [\![M]\!]\rho} [\![N]\!](\rho, x, y) & \sim_Y [\![N]\!](\rho, x, x) \\ & \sim_Y [\![N]\!](\rho', x', x') \\ & \sim_Y \bigcup_{x' \in [\![M]\!]\rho'} [\![N]\!](\rho', x', x') \\ & = \bigcup_{x' \in [\![M]\!]\rho'} [\![N[x/y]]\!](\rho', x') \end{array}$$

For the second part, we get $|[\![N]\!](\rho, x, x) \,/\, [\![Y]\!]| \in [\![\varepsilon]\!]$ by induction, and we then just need to know that $[\![\varepsilon]\!] \subseteq [\![\mathtt{01} \cdot \varepsilon]\!]$, which is easily checked.

Pure Lambda Hoist. Define $\Gamma = U(\Theta)$, $A = U(X)$, $B = U(Y)$, $C = U(Z)$ and pick $(\rho, \rho') \in [\![\Theta]\!]$. We need

$$\Big(\Big\{\lambda x \in [\![A]\!]. \bigcup_{z \in [\![M]\!]\rho} [\![N]\!](\rho, x, z)\Big\}, \bigcup_{z \in [\![M]\!]\rho'} \{\lambda x \in [\![A]\!].[\![N]\!](\rho', x, z)\}\Big) \\ \in [\![T_1(X \to T_\varepsilon Y)]\!]$$

Since the first component of the pair above is a singleton, the cardinality constraint associated with the outer computation type is easily satisfied. For the $\sim$ part, we look at typical elements of the first and second components above. By induction on M, we can pick $z' \in [\![M]\!]\rho'$ and we claim that for any such z',

$$\Big(\lambda x \in [\![A]\!]. \bigcup_{z \in [\![M]\!]\rho} [\![N]\!](\rho, x, z),\ \lambda x \in [\![A]\!].[\![N]\!](\rho', x, z')\Big) \in [\![X \to T_\varepsilon Y]\!]$$

which will suffice. So assume $(x, x') \in [\![X]\!]$ and we want

$$\Big(\bigcup_{z \in [\![M]\!]\rho} [\![N]\!](\rho, x, z),\ [\![N]\!](\rho', x', z')\Big) \in [\![T_\varepsilon Y]\!]$$

The cardinality part of the above is immediate by induction on N. If y is an element of the union, then $y \in [\![N]\!](\rho, x, z)$ for some $z \in [\![M]\!]\rho$. But then $(z, z') \in [\![Z]\!]$ because $|[\![M]\!]\rho/[\![Z]\!]| = 1$, so $\exists y' \in [\![N]\!](\rho', x', z')$ with $(y, y') \in [\![Y]\!]$. Conversely, if $y' \in [\![N]\!](\rho', x', z')$ then for any $z \in [\![M]\!]$ there's $y \in [\![N]\!](\rho, x, z)$ with $(y, y') \in [\![Z]\!]$, so the two expressions are in the $\sim_Z$ relation, as required. $\square$

For example, if we define

$$f_1 = \lambda g : \mathtt{unit} \to T\mathtt{int}.\mathtt{let}\ x \Leftarrow g\,()\ \mathtt{in}\ \mathtt{let}\ y \Leftarrow g\,()\ \mathtt{in}\ \mathtt{val}\ x + y$$
$$f_2 = \lambda g : \mathtt{unit} \to T\mathtt{int}.\mathtt{let}\ x \Leftarrow g\,()\ \mathtt{in}\ \mathtt{val}\ x + x$$

then we have $\vdash f_1 = f_2 : (\mathtt{unit} \to T_{\mathtt{01}}\mathtt{int}) \to T_{\mathtt{01}}\mathtt{int}$ and hence, for example,

$$\vdash (\mathtt{val}\ f_1)\ \mathtt{or}\ (\mathtt{val}\ f_2)\ =\ \mathtt{val}\ f_2 : T_1((\mathtt{unit} \to T_{\mathtt{01}}\mathtt{int}) \to T_{\mathtt{01}}\mathtt{int}).$$

Note that the notion of equivalence really is type-specific. We have

$$\not\vdash f_1 = f_2 : (\mathtt{unit} \to T_{\mathtt{N}}\mathtt{int}) \to T_{\mathtt{N}}\mathtt{int}$$

and that equivalence indeed does not hold in the semantics, even though both f_1 and f_2 are related to themselves at (i.e. have) that type.

Extensions. The syntactic rules can be augmented with anything proved sound in the model. For example, one can add a subtyping rule $T_{1+}X \le T_1 X$ for any X such that $|[\![UX]\!]/[\![X]\!]| = 1$. Or one can manually add typing or equational judgements that have been proved by hand, without compromising the general equational theory. For example, Wadler [28] considers parsers that we could give types of the form $P_\varepsilon X = \mathtt{string} \to T_\varepsilon(X \times \mathtt{string})$. One type of the alternation combinator

$$alt(p_1, p_2) = \lambda s : \mathtt{string}. \begin{array}{l} \mathtt{let}\ (v, s') \Leftarrow p_1 s\ \mathtt{in}\ \mathtt{val}\ (\mathtt{inl} v, s') \\ \mathtt{or}\ \mathtt{let}\ (v, s') \Leftarrow p_2 s\ \mathtt{in}\ \mathtt{val}\ (\mathtt{inr} v, s') \end{array}$$

is $P_{01}X \times P_{01}Y \to P_{\mathtt{N}}(X+Y)$. But if we know that $p_1 : P_{01}X$ and $p_2 : P_{01}Y$ cannot both succeed on the same string, then we can soundly ascribe $alt(p_1, p_2)$ the type $P_{01}(X+Y)$.

A further extension is to add pruning. One way is

$$\frac{\Gamma \vdash M_1 : TA \Gamma \vdash M_2 : TA}{\Gamma \vdash M_1 \,\texttt{orelse}\, M_2 : TA}$$

$$[\![\Gamma \vdash M_1 \,\texttt{orelse}\, M_2 : TA]\!]\rho = \begin{cases} [\![\Gamma \vdash M_2 : TA]\!]\rho \text{ if } [\![\Gamma \vdash M_1 : TA]\!]\rho = \emptyset \\ [\![\Gamma \vdash M_1 : TA]\!]\rho \text{ otherwise} \end{cases}$$

with refined typing

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X \Theta \vdash M_2 : T_{\varepsilon_2} X}{\Theta \vdash M_1 \,\texttt{orelse}\, M_2 : T_{\varepsilon_1 \rhd \varepsilon_2} X}$$

where $\varepsilon_1 \rhd \varepsilon_2$ is defined to be $\varepsilon_1 + \varepsilon_2$ if $\mathtt{0} \leq \varepsilon_1$ and ε_1 otherwise. The `orelse` operation can be used to improve efficiency in search-style uses of non-determinism and is, of course, the natural combining operation to use in error-style uses.

4 Discussion

We have given an elementary relational semantics to a simple effect system for non-deterministic programs, and shown how it may be used to establish effect-dependent program equivalences. Extending or adapting the constructions to richer languages or slightly different monads should be straightforward. One can also enrich the effect language itself, for example by adding conjunctive refinements and effect polymorphism, as we have done previously [3]. The simple style of effect system presented here seems appropriate for fairly generic compilation of a source language with a pervasively non-deterministic semantics, but for which much code could actually be expected to be deterministic. For serious optimization of non-trivially non-deterministic code, one would need to combine effects with refinements on values, to formalize the kind of reasoning used in the parser example above.

Non-determinism monads are widely used to program search, queries, and pattern matching in functional languages. In Haskell, the basic constructs we use here are abstracted as the `MonadPlus` class, though different instances satisfy different laws, and there has been much debate about which laws one should expect to hold in general [25,27,32].[2] Several researchers have studied efficient implementations of functional non-determinism and their various equational properties [14,17].

Static analysis of functional non-determinism is not so common, though Kammar and Plotkin have developed a general theory of effects and effect-based transformations, based on the theory of algebraic effects [18]. Non-determinism

[2] Phil was involved in this debate at least as far back as 1997 [11].

is just one example of that theory, and Kammar and Plotkin establish some equational laws that are very similar to the ones presented here. One interesting difference between their work and that we describe here is that our refinements of the computation type are not necessarily monads in their own right. The interpretation of $T_0 X$ is $\{(\emptyset, \emptyset)\}$, which is not preserved by the underlying monadic unit $a \mapsto \{a\}$. If we were to track slightly more refined cardinalities (e.g. sets of size two) then, as we have already observed, the abstract multiplication would no longer be idempotent (or commutative), which also implies that the $T_\varepsilon(\cdot)$s would no longer be themselves monads.

Katsumata has presented an elegant general theory of effect systems, using monoidal functors from a preordered monoid (the effect annotations) to endofunctors on the category of values [19]. The effect system given here is an instance of Katsumata's theory. Our very concrete approach to specific effects is by comparison, perhaps rather unsophisticated. On the other hand, the elementary approach seems to scale more easily to richer effect systems, for example for concurrency [5]. (Indeed, it would be natural to augment concurrent state effects with non-determinism information.) Ahman and Plotkin are developing a still more general framework for refining algebraic effects, which can express temporal properties and of which our analysis should be a special case [1].

There is considerable literature on determinism and cardinality analyses in the context of logic programming (e.g. [10,13]) with applications including introducing cuts and improving the efficiency of parallel search. Many of these analyses can also detect mutual exclusion between tests [20]. Mercury allows programmers to specify determinism using (we were pleased to discover) the same cardinalities as we do here ($\texttt{0} = \texttt{failure}, \texttt{1} = \texttt{det}, \texttt{01} = \texttt{semidet}, \texttt{1+} = \texttt{multidet}, \texttt{N} = \texttt{nondet}$) and similar abstract operations in the checking algorithm [16].

Acknowledgements. We thank the referees for their thorough and helpful comments.

References

1. Ahman, D., Plotkin, G.D.: Refinement types for algebraic effects. In: Abstracts of the 21st Meeting 'Types for Proofs and Programs' (TYPES), Institute of Cybernetics, Tallinn University of Technology, pp. 10–11 (2015)
2. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 14–25. ACM (2004)
3. Benton, N., Buchlovsky, P.: Semantics of an effect analysis for exceptions. In: 3rd ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI), pp. 15–26. ACM (2007)
4. Benton, N., Hofmann, M., Nigam, V.: Abstract effects and proof-relevant logical relations. In: 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 619–632. ACM (2014)
5. Benton, N., Hofmann, M., Nigam, V.: Effect-dependent transformations for concurrent programs arXiv:1510.02419 (2015)

6. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, p. 42. Springer, Heidelberg (2002)
7. Benton, A., Kennedy, L. Beringer, N., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), pp. 87–96. ACM (2007)
8. Benton, N., Kennedy, A., Hofmann, M.O., Beringer, L.: Reading, writing and relations: towards extensional semantics for effects analyses. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 114–130. Springer, Heidelberg (2006)
9. Benton, N., Kennedy, A., Russell, G.: Compiling Standard ML to Java bytecodes. In: Third ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 129–140. ACM (1998)
10. Braem, C., Le Charlier, B., Modart, S., Van Hentenryck, P.: Cardinality analysis of Prolog. In: International Symposium on Logic Programming, pp. 457–471. MIT Press (1994)
11. Claessen, K., Wadler, P. et al.: Laws for monads with zero and plus. In: Haskell mailing list, May 1997. http://www.mail-archive.com/haskell%40haskell.org/msg01583.html. Accessed January 2016
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977)
13. Debray, S.K., Warren, D.S.: Functional computations in logic programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **11**(3), 451–481 (1989)
14. Fischer, S., Kiselyov, O., Shan, C.-C.: Purely functional lazy nondeterministic programming. J. Funct. Program. **21**(4/5), 413–465 (2011)
15. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: ACM Conference on LISP and Functional Programming, pp. 28–38. ACM (1986)
16. Henderson, F., Somogyi, Z., Conway, T.: Determinism analysis in the Mercury compiler. In: Proceedings of the Australian Computer Science Conference, pp. 337–34 (1996)
17. Hinze, R.: Deriving backtracking monad transformers. In: Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 186–197. ACM (2000)
18. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimizations. In: 39th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 349–360. ACM (2012)
19. Katsumata, S.: Parametric effect monads and semantics of effect systems. In: 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 633–646. ACM (2014)
20. López-García, P., Bueno, F., Hermenegildo, M.V.: Determinacy analysis for logic programs using mode and type information. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 19–35. Springer, Heidelberg (2005)
21. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 47–57. ACM (1988)
22. Moggi, E.: Computational lambda-calculus and monads. In: 4th Annual Symposium on Logic in Computer Science (LICS), pp. 14–23. IEEE Computer Society (1989)

23. Peyton Jones, S., Wadler, P.: Comprehensive comprehensions. In: ACM SIGPLAN Workshop on Haskell, pp. 61–72. ACM (2007)
24. Reynolds, J.C.: The meaning of types - from intrinsic to extrinsic semantics. Technical report BRICS RS-00-32, BRICS, University of Aarhus, December 2000
25. Rivas, E., Jaskelioff, M., Schrijvers, T.: From monoids to near-semirings: the essence of monadplus and alternative. In: 17th International Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 196–207. ACM (2015)
26. Tolmach, A.: Optimizing ML using a hierarchy of monadic types. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 97–115. Springer, Heidelberg (1998)
27. Uustalu, T.: A divertimento on MonadPlus and nondeterminism. J. Logical Algebraic Methods Program. (2016, to appear)
28. Wadler, P.: How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)
29. Wadler, P.: Theorems for free!. In: Fourth International Symposium on Functional Programming Languages and Computer Architecture (FPCA), pp. 347–359. ACM (1989)
30. Wadler, P.: The marriage of effects and monads. In: Third ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 63–74. ACM (1998)
31. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Logic **4**(1), 1–32 (2003)
32. Yakeley, A. et al.: MonadPlus reform proposal (2006). https://wiki.haskell.org/MonadPlus_reform_proposal. Accessed January 2016

The Essence of Inheritance

Andrew P. Black[1(✉)], Kim B. Bruce[2], and James Noble[3]

[1] Portland State University, Portland, OR, USA
black@cs.pdx.edu
[2] Pomona College, Claremont, CA, USA
kim@cs.pomona.edu
[3] Victoria University of Wellington, Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Abstract. Programming languages serve a dual purpose: to communicate programs to computers, and to communicate programs to humans. Indeed, it is this dual purpose that makes programming language design a constrained and challenging problem. Inheritance is an *essential* aspect of that second purpose: it is a tool to improve communication. Humans understand new concepts most readily by *first* looking at a number of concrete examples, and *later* abstracting over those examples. The essence of inheritance is that it mirrors this process: it provides a formal mechanism for moving from the concrete to the abstract.

Keywords: Inheritance · Object-oriented programming · Programming languages · Abstraction · Program understanding

1 Introduction

Shall I be abstract or concrete?

An abstract program is more general, and thus has greater potential to be reused. However, a concrete program will usually solve the specific problem at hand more simply.

One factor that should influence my choice is the ease with which a program can be understood. Concrete programs ease understanding by making manifest the action of their subcomponents. But, sometimes a seemingly small change may require a concrete program to be extensively restructured, when judicious use of abstraction would have allowed the same change to be made simply by providing a different argument.

Or, I could use inheritance.

The essence of inheritance is that it lets us avoid the unsatisfying choice between abstract and concrete. Inheritance lets us start by writing a concrete program, and then later on abstracting over a concrete element. This abstraction step is *not* performed by editing the concrete program to introduce a new parameter. That is what would be necessary without inheritance. To the contrary: inheritance allows us to treat the concrete element *as if it were a parameter*, without actually changing the code. We call this ex post facto parameterization; we will illustrate the process with examples in Sects. 2 and 3.

Inheritance has been shunned by the designers of functional languages. Certainly, it is a difficult feature to specify precisely, and to implement efficiently, because it means (at least in the most general formulations) that any apparent constant might, once inherited, become a variable. But, as Einstein is reputed to have said, in the middle of difficulty lies opportunity. The especial value of inheritance is as an aid to program understanding. It is particularly valuable where the best way to understand a complex program is to start with a simpler one and approach the complex goal in small steps.

Our emphasis on the value of inheritance as an aid to human understanding, rather than on its formal properties, is deliberate, and long overdue. Since the pioneering work of Cook and Palsberg (1989), it has been clear that, from a formal point of view, inheritance is equivalent to parameterization. This has, we believe, caused designers of functional languages to regard inheritance as unimportant, unnecessary, or even undesirable, arguing (correctly) that it can be simulated using higher-order parameterization. This line of argument misses the point that two formally-equivalent mechanisms may behave quite differently with respect to human cognition.

It has also long been known that *Inheritance is Not Subtyping* (Cook et al. 1990). In spite of this, many programming languages conflate subtyping and inheritance; Java, for example, restricts the use of inheritance so that the inheritance hierarchy is a sub-tree of the type hierarchy. Our goal in this paper is to consider inheritance as a mechanism in its own right, quite separate from the subtyping relation. We are aided in this goal by casting our example in the Grace programming language (Black et al. 2012), which cleanly separates inheritance and subtyping.

The form and content of this paper are a homage to Wadler's "Essence of Functional Programming" (1992), which was itself inspired by Reynold's "Essence of ALGOL" (1981). The first example that we use to illustrate the value of inheritance, discussed in Sect. 2, is based on Wadler's interpreter for a simple language, which is successively extended to include additional features. (Reynolds (1972) had previously defined a series of interpreters in a classic paper that motivated the use of continuations.) The second example is based on Armstrong's description of the Erlang OTP Platform (Armstrong 2007, Chap. 16); this is discussed in Sect. 3. We conclude in Sect. 4, where we also discuss some of the consequences for type-checking.

2 Interpreting Inheritance

This section demonstrates the thesis that inheritance enhances understandability, by presenting several variations of an implementation of a fragment of an expression language. Wadler (1992) uses an interpreter for the lambda calculus, but we use a simpler language inspired by the "First Monad Tutorial" (Wadler 2013).

We show our examples in Grace (Black et al. 2012), a simple object-oriented language designed for teaching, but no prior knowledge of the language is assumed. What the reader will require is a passing familiarity with the basics of object-oriented programming; for general background, see Cox (1986), or Black et al. (2009). Like Haskell, Grace does not require the programmer to specify types on every declaration, although, also like Haskell, the programmer may well find that adding type annotations makes it easier to find errors.

2.1 Variation Minus One: Object-Oriented Evaluation

We start with a simple object-oriented evaluator for Wadler's tutorial example (2013) — mathematical expressions consisting only of constants con and division div — so 1/2 is represented as div(con 1, con 2). The object-oriented style integrates the data and the descriptions of the computations on that data. Thus, there is no separation between the "interpreter" and the "terms" that it interprets; instead the object-oriented program contains "nodes" of various kinds, which represent *both* expressions *and* the rules for their evaluation.

In Grace, con and div are classes; they correspond to Wadler's algebraic data constructors. Grace classes (like those in O'Caml and Python) are essentially methods that return new objects. Evaluation of an expression in the example language is accomplished by requesting that the objects created by these classes execute their own eval methods, rather than by invoking a globally-defined function interp. The resulting representation of simpleExpressions is straightforward. In addition to the eval methods, the objects are also given asString methods that play a role similar to that of Wadler's show∗ functions: they return strings for human consumption. (Braces in a string constructor interpolate values into the string.)

```
module "simpleExpressions"
class simpleExpressions {
    class con(n) {
        method asString −> String { "con {n}" }
        method eval −> Number { n }
    }
    class div(l, r) {
        method asString −> String { "div({l}, {r})" }
        method eval −> Number { l.eval / r.eval }
    }
}
```

Given these definitions, we can instantiate an expression tree, evaluate it, and print out the result. (Grace's **def** is similar to ML and Scala's **val**).

```
import "simpleExpressions" as s
def e = s.simpleExpressions
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)        // prints 42
def error = e.div(e.con 1, e.con 0)
print (error.eval)         // prints infinity
```

Wadler (1992) does not present code equivalent to this variation, which is why we have labelled it "Variation Minus One". Instead, he starts with "Variation Zero", which is a monadic interpreter based on a trivial monad.

2.2 What is a Monad?

For our purposes, a monad is an object that encapsulates an effectful computation. If the underlying computation produces an answer a, then the monad can be thought of as "wrapping" the computation of a. If you already understand monads, you should feel free to skip to Sect. 2.3.

All monad objects provide a "bind" method, which, following Haskell, we spell $\gg=$. (In Grace, binary methods can be named by arbitrary sequences of operator symbols; there is no precedence.) In general, a monad will also encapsulate some machinery specialized to its particular purpose and accessible to its computations. For example, the output monad of Sect. 2.6 encapsulates a string and an operation by which a computation can append its output thereto.

The bind method $\gg=$ represents sequencing. The right-hand argument of $\gg=$ is a Grace code block, which denotes a λ-expression or function, and which represents the computation that should succeed the one in the monad. (The successor computation is like a continuation; Wadler (1992, Sect. 3) explores the similarities and differences.) The request `m>>= { a -> ... }` asks the monad object m to build a new monad that, when executed, will first execute m's computation, bind the answer to the parameter a of the block, and then execute the successor computation represented by .... Think of the symbol $\gg=$ as piping the result of the computation to its left into the function object to its right. The block should answer a monad, so the type of $\gg=$ is `(Block1<A, Monad>) -> Monad`. (The Grace type `Block1<X, Y>` describes a λ-expression that takes an argument of type X and returns a result of type Y.) The monad type is actually parameterized by the type of its contents, but Grace allows us to omit type parameters, which we do here for simplicity. (In Grace's gradual type system, omitted types and type parameters are assumed to be Unknown, which means that we are choosing to say nothing about the type of the corresponding object.)

In the object-oriented formulation of monads, the monad classes construct monad objects, and thus subsume the operation that Haskell calls unitM or **return**. In the sections that follow, we will introduce four monad classes: pure(contents), which encapsulates pure computations that have no effects; raise(reason), which encapsulates computations that raise exceptions; stateMonad(contents), which encapsulates stateful computations; and monad(contents)output(str), which encapsulates computations that produce output. The name of this last class is an example of a method name with multiple parts, a Grace feature designed to improve readability that is similar to Smalltalk method selectors. The class monad()output() expects two argument lists, each with a single argument.

Informally, the monad classes get us into a monad, and $\gg=$ gets us around the monad. How do we get out of the monad? In general, such operations require a more *ad hoc* design. In this article, it will suffice to provide the monads with asString methods, which answer strings.

2.3 Variation Zero: Monadic Evaluation

We now *adapt* the code of Sect. 2.1 to be monadic. Here we see the key difference between an approach that uses inheritance and one that does not: rather than define the new expression evaluation mechanism from scratch, we make a new monadicExpression module that uses inheritance to modify the components imported from the previous simpleExpressions module.

For a monadic interpreter, we of course need a monad, which we define in Grace as a class called pure, since it encapsulates effect-free computations that answer pure values. The bind method ≫= applies its parameter k to the contents of the monad; in Grace, application of a λ-expression is performed explicitly using the method apply.

module "monadicExpressions"

```
   import "simpleExpressions" as se
 2
   type Monad = type {
 4     ≫= (k:Block1) −> Monad
   }
 6 class monadicExpressions {
       inherits se.simpleExpressions
 8
       class pure(contents) −> Monad {
10         method asString −> String { "pure({contents})" }
           method ≫= (k) −> Monad { k.apply(contents) }
12     }

14     class con(n) {
           inherits se.simpleExpressions.con(n)
16             alias simpleEval = eval
           method eval −> Monad is override { pure(simpleEval) }
18     }
       class div(l, r) {
20         inherits se.simpleExpressions.div(l, r)
           method eval −> Monad is override {
22             l.eval ≫= { a −>
                   r.eval ≫= { b −>
24                     pure(a / b) } }
           }
26     }
   }
```

Having established this framework, we then define the nodes of this monadic variant of expressions; these differ from the simple ones of Sect. 2.1 *only* in the eval method, so the *only* content of these new classes are eval methods that override the inherited ones. The **inherits** statement on line 15 says that the methods of con in monadicExpressions are the same as those of con in simpleExpressions, *except* for the overridden method eval. The **alias** clause on the **inherits** statement provides a new name for the inherited eval, so that the overriding method body can lift the value produced by the inherited eval method into the monad. The eval method of div is similar. We ask readers concerned about type-checking this instance of "the expression problem" to suspend disbelief until Sect. 4.2.

Notice that the original definition of se.simpleExpressions in Sect. 2.1 did not specify that eval was a parameter. On the contrary: it was specified as a simple concrete method. The method eval becomes a parameter only when an inheritor of se.simpleExpressions chooses to override it. This is what we meant by *ex post facto parameterization* in the introduction.

Here is a test program parallel to the previous one: the output indicates that the results are in the monad.

```
import "monadicExpressions" as m
def e = m.monadicExpressions
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)          // prints pure(42)
def error = e.div(e.con 1, e.con 0)
print (error.eval)          // prints pure(infinity)
```

In the remainder of this section, we show how three of Wadler's variations on this monadic interpreter — for exceptions, state, and output — can be implemented incrementally using inheritance.

2.4 Variation One: Evaluation with Exceptions

The basic idea behind adding exceptions is that the result of an evaluation is no longer always a simple value (like 42) but may also be an exception. Moreover, exceptions encountered during the evaluation of an expression terminate the evaluation and emit the exception itself as the answer. In our toy language, the only source of exceptions will be division by zero.

We have to extend our monad to deal with exceptional results. Working in Haskell, Wadler redefines the algebraic data type contained in the monad, starting from scratch. Working in Grace, we define exceptionExpressions by inheriting from monadicExpressions, and need write code only for the differences. We start by defining a new monad class raise (lines 5–8), whose bind method stops execution when an exception is encountered by immediately returning **self**, thus discarding any computations passed to it as its parameter k.

```
module "exceptionExpressions"
   import "monadicExpressions" as m
 2
   class exceptionExpressions {
 4     inherits m.monadicExpressions
       class raise(reason) -> m.Monad {
 6         method asString -> String is override { "raise({reason})" }
           method >>= (k) -> m.Monad is override { self }
 8     }
       class div(l, r) {
10         inherits m.monadicExpressions.div(l, r)
           method eval -> m.Monad is override {
12             l.eval >>= { a ->
                   r.eval >>= { b ->
14                     if (b == 0)
                           then { raise "Divide {a} by zero" }
16                         else { pure(a / b) } } } }
       }
18 }
```

We must also change the eval methods to raise exceptions at the appropriate time. Since the evaluation of constants cannot raise an exception, the con class is unchanged, *so we say nothing*, and use the inherited con. In contrast, evaluating a div can raise an exception, so we have to provide a new eval method for div, shown on lines 11–16. This code returns a value in the raise monad when the divisor is zero.

If you are familiar with Wadler's addition of error messages to his monadic interpreter (Wadler 1992, Sect. 2.3) this will look quite familiar. In fact, the developments are so similar that it is easy to overlook the differences:

1. At the end of his description of the changes necessary to introduce error messages, Wadler writes: "To modify the interpreter, substitute monad E for monad M, and replace each occurrence of unitE Wrong by a suitable call to errorE. ... No other changes are required." Wadler is giving us editing instructions! In contrast, the box above represents a file of real Grace code that can be compiled and executed. It contains, if you will, not only the new definitions for the eval method and the monad, but *also the editing instructions* required to install them.
2. Not only does the boxed code represent a unit of compilation, it also represents a *unit of understanding*. We believe that it is easier to understand a complex structure like a monad with exceptions by first understanding the monad pure, and *then* understanding the exceptions. Perhaps Wadler agrees with this, for he himself uses just this form of exposition in his *paper*. But, lacking inheritance, he cannot capture this stepwise exposition in his *code*.

2.5 Variation Three: Propagating State

To illustrate the manipulation of state, Wadler keeps a count of the number of reductions; we will count the number of divisions. Each computation is given an input state in which to execute, and returns a potentially different output state; the difference between the input and output states reflect the changes to the state effected by the computation. Rather than representing $\langle result, state \rangle$ pairs with anonymous tuples, we will use Response objects with two methods, shown below.

module "response"

```
type Response = type {
    result -> Unknown
    state -> Unknown
}
class result(r) state(s) -> Response {
    method result { r }
    method state { s }
    method asString { "result({r}) state({s})" }
}
```

What changes must be made to monadicExpressions to support state? The key difference between the stateMonad and the pure monad is in the $\gg=$ method.

We follow the conventional functional approach, with the contents of the monad being a function that, when applied to a state, returns a Response. The method executeIn captures this idea.

module "statefulEvaluation"

```
   import "monadicExpressions" as m
 2 import "response" as rp

 4 class statefulEvaluation {
       inherits m.monadicExpressions
 6     class stateMonad(contents:Block1) -> m.Monad {
           method asString -> String { "stateMonad({executeIn 0})" }
 8         method executeIn(s) -> rp.Response { contents.apply(s) }
           method ≫= (k:Block) -> m.Monad {
10             stateMonad { s ->
                   def resp = executeIn(s)
12                 k.apply(resp.result).executeIn(resp.state) }
           }
14     }
       method pure(a) -> m.Monad is override {
16         stateMonad { s -> rp.result(a) state(s) }
       }
18     method tally -> m.Monad {
           stateMonad { s -> rp.result(done) state(s + 1) }
20     }
       class div(l, r) is override {
22         inherits m.monadicExpressions.div(l, r)
               alias monadicEval = eval
24         method eval -> m.Monad is override {
               tally ≫= { x -> monadicEval }
26         }
       }
28 }
```

The ≫= method of the state monad (lines 9–13) returns a new state monad that, when executed, will first execute its contents in the given state s, then apply its argument k to the result, and then execute the contents of that monad in the threaded state. Given this definition of ≫=, and a redefinition of the pure method to ensure that values are lifted into the correct monad, we do not need to redefine *any* of the tree nodes—so long as their evaluation does not involve using the state.

Of course, the *point* of this extension is to enable some evaluations to depend on state. We will follow Wadler in using state in a somewhat artificial way: the state will be a single number that counts the number of divisions. The tally method increments the count stored in the state and answers done (Grace's unit), indicating that its purpose is to cause an effect rather than to compute a result.

Naturally, counting divisions requires extending the eval operation in the div class (but not elsewhere) to do the counting. The overriding eval method on lines 24–26 first counts using tally, and then calls the inherited eval method. Notice also that the asString operation of the monad (line 7) executes the contents of the monad in the state 0, representing the initial value of the counter. This is exactly parallel to Wadler's function showS.

Here is an example program similar to the previous one:

```
import "statefulEvaluation" as se
def e = se.statefulEvaluation

def simple = e.con 34
print (simple.eval)          // prints stateMonad(result(34) state(0))

def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)          // prints stateMonad(result(42) state(2))

def error = e.div(e.con 1, e.con 0)
print (error.eval)          // prints stateMonad(result(infinity) state(1))
```

2.6 Variation Four: Output

Our final example builds up an output string, using similar techniques. We reuse monadicExpressions, this time overriding the inherited eval methods to produce output. We provide a new monad that holds both result and output, and is equipped with a bind operator that accumulates the output at each step.

module "outputEvaluation"

```
import "monadicExpressions" as m

class outputEvaluation {
    inherits m.monadicExpressions
    class con(n) {
        inherits m.monadicExpressions.con(n)
        method eval −> m.Monad {
            out "{self.asString} = {n}\n" >>= { _ −> pure(n) }
        }
    }
    class div(l, r) {
        inherits m.monadicExpressions.div(l, r)
        method eval −> m.Monad is override {
            l.eval >>= { a: Number −>
                r.eval >>= { b: Number −>
                    out "{self.asString} = {a / b}\n" >>= { _ −> pure(a / b) } } }
        }
    }
    method pure(a) −> m.Monad is override { monad(a) output "" }
    method out(s) −> m.Monad { monad(done) output(s) }
    class monad(contents') output(output') {
        method contents {contents'}
        method output {output'}
        method asString {"output monad contents: {contents} output:\n{output}"}
        method >>= (k) {
            def next = k.apply(contents)
            monad(next.contents) output(output ++ next.output)
        }
    }
}
```

Here is an example program, with its output.

```
import "outputEvaluation" as o

def e = o.outputEvaluation
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)
// prints output monad contents: 42 output:
// con 1932 = 1932
// con 2 = 2
// div(con 1932, con 2) = 966
// con 23 = 23
// div(div(con 1932, con 2), con 23) = 42
def error = e.div(e.con 1, e.con 0)
print (error.eval)

// prints output monad contents: infinity output:
// con 1 = 1
// con 0 = 0
// div(con 1, con 0) = infinity
```

3 The Erlang OTP Platform

The essential role that inheritance can play in *explaining* how a software system works was brought home to Black in 2011. Black was on sabbatical in Edinburgh, hosted by Phil Wadler. Black and Wadler had been discussing ways of characterizing encapsulated state — principally monads and effect systems. Erlang came up in the conversation as an example of a language in which state is handled explicitly, and Black started studying Armstrong's *Programming Erlang* (2007).

The Erlang OTP "generic server" provides properties such as transactions, scalability, and dynamic code update for arbitrary server behaviours. Transactions can be implemented particularly simply because state is explicit. Armstrong writes: "Put simply, the [generic server] solves the nonfunctional parts of the problem, while the callback solves the functional part. The nice part about this is that the nonfunctional parts of the problem (for example, how to do live code upgrades) are the same for all applications." (Armstrong 2007, p. 286)

A reader of this section of Armstrong's book can't help but get the sense that this material is significant. In a separate italicized paragraph, Armstrong writes: "This is the most important section in the entire book, so read it once, read it twice, read it 100 times — just make sure the message sinks in."

The best way that Armstrong can find to explain this most-important-of-messages is to write a small server program in Erlang, and then generalize this program to add first transactions, and then hot code swapping. The best way that Black could find to understand this message was to re-implement Armstrong's server in another language — Smalltalk. Using Smalltalk's inheritance, he was able to reflect Armstrong's development, not as a series of *separate* programs, but in the stepwise development of a *single* program.

3.1 Armstrong's Goal

What does Armstrong seek to achieve with the generic server? In this context, a "server" is a computational engine with access to persistent memory. Servers typically run on a remote computer, and in the Erlang world the primary server is backed-up by a secondary server that takes over if the primary should fail. For uniformity with the proceeding example, we here present a simplified version of the OTP server in Grace; in addition to Armstrong's Erlang version, our code is also based on Parkinson, and Noble's translation of the OTP server to a Java-like language (2008). For brevity, and to focus attention on the use of inheritance, our version omits name resolution, remote requests, concurrency and failover.

To be concrete, let's imagine two particular servers: a "name server" and a "calculation server". The name server remembers the locations of objects, with interface:

```
type NameServer = {
    add(name:String) place(p:Location) −> Done
    whereIs(name:String) −> Location
}
```

The name of the first method in the above type is add()place(), a two-part name with two parameters.

The calculation server acts like a one-function calculator with a memory; clear clears the memory, and add adds its argument to the memory, and stores the result in the memory as well as returning it. The calculation server has the interface

```
type CalculationServer = {
    clear −> Number
    add(e:Number) −> Number
}
```

Both of these servers maintain state; we will see later why this is relevant.

Armstrong refers to the actual server code as "the callback". His goal is to write these callbacks in a simple sequential style, but with explicit state. The generic server can then add properties such as transactions, failover and hot-swapping. The simple sequential implementation of the name server is shown on the following page.

The state of this "callback" is represented by a Dictionary object that stores the *name* → *location* mapping. The method initialState returns the initial state: a new, empty, Dictionary. The method add()place()state() is used to implement the client's add()place() method. The generic server provides the additional state argument, an object representing the callback's state. The method returns a Response (as in Sect. 2.5) comprising the newState dictionary, and the actual result of the operation, p. Similarly, the method whereIs()state() is used to implement the client's whereIs() method. The generic server again provides the additional state argument. This method returns a Response comprising the result of looking-up name in dict and the (unchanged) state.

```
module "nameServer"
import "response" as r

type Location = Unknown
type NameServer = {
    add(name:String) place(p:Location) −> Done
    whereIs(name:String) −> Location
}
type NsState = Dictionary<String, Location>

class callback {
    method initialState −> NsState { dictionary.empty }
    method add(name:String) place(p) state (dict:NsState) −> r.Response {
        def newState = dict.copy
        newState.at(name) put(p)
        r.result(p) state(newState)
    }
    method whereIs(name:String) state(dict:NsState) −> r.Response {
        def res = dict.at(name)
        r.result(res) state(dict)
    }
}
```

Finally, let's consider what happens if this name server callback is asked for the location of a name that is not in the dictionary. The lookup dict.at(name) will raise an exception, which the callback itself does not handle.

Notice that our *nameServer* module contains a class callback whose instances match the type

```
type Callback<S> = type {
    initialState −> S
}
```

for appropriate values of S. This is true of all server callback modules. Particular server callbacks extend this type with additional methods, all of which have a name that ends with the word state, and which take an extra argument of type S that represents their state.

3.2 The Basic Server

Our class server corresponds to Armstrong's module server1. This is the "generic server" into which is installed the "callback" that programs it to provide a particular function (like name lookup, or calculation).

A Request encapsulates the name of an operation and an argument list. The basic server implements two methods: handle(), which processes a single incoming request, and serverLoop(), which manages the request queue.

module "basicServer"

```
import "mirrors" as m

type Request = type {
    name -> String
    arguments -> List<Unknown>
}

class server(callbackName:String) {
    var callbackMirror
    var state
    startUp(callbackName)

    method startUp(name) {
        def callbackModule = m.loadDynamicModule(name)
        def callbackObject = callbackModule.callback
        callbackMirror := m.reflect(callbackObject)
        state := callbackObject.initialState
    }
    method handle(request:Request) {
        def cbMethodMirror = callbackMirror.getMethod(request.name ++ "state")
        def arguments = request.arguments ++ [state]
        def ans = cbMethodMirror.requestWithArgs(arguments)
        state := ans.state
        ans.result
    }
    method serverLoop(requestQ) {
        requestQ.do { request ->
            def res = handle(request)
            log "handle: {request.name} args: {request.arguments}"
            log "    result: {res}"
        }
    }
    method log(message) { print(message) }
}
```

A server implements three methods. Method startUp(name) loads the callback module name and initializes the server's state to that required by the newly-loaded callback.

The method handle accepts an incoming request, such as "add()place()", and appends the string "state", to obtain method name like "add()place()state". It then requests that the callback executes its method with this name, passing it the arguments from the request and an additional state argument. Thus, a request like add "BuckinghamPalace" place "London" might be transmitted to the name-Server as add "BuckinghamPalace" place "London" state (dictionary.empty). The state component of the response would then be a dictionary containing the mapping from "BuckinghamPalace" to "London"; this new dictionary would provide the state argument for the next request.

The method serverLoop is a simplified version of Armstrong's loop/3 that omits the code necessary to receive messages and send back replies, and instead uses a local queue of messages and Grace's normal method-return mechanism.

Here is some code that exercises the basic server:

```
import "basicServer" as basic

class request(methodName)withArgs(args) {
    method name { methodName }
    method arguments { args }
}
def queue = [
    request "add()place()" withArgs ["BuckinghamPalace", "London"],
    request "add()place()" withArgs ["EiffelTower", "Paris"],
    request "whereIs()" withArgs ["EiffelTower"]
]
print "starting basicServer"
basic.server("nameServer").serverLoop(queue)
print "done"
```

To keep this illustration as simple as possible, this code constructs the requests explicitly; in a real remote server system, the requests would be constructed using reflection, or an RPC stub generator. This is why they appear as, for example, `request "add()place()" withArgs ["BuckinghamPalace", "London"]`, instead of as `add "BuckinghamPalace" place "London"`. Here is the log output:

```
starting basicServer
handle: add()place() args: [BuckinghamPalace, London]
        result: London
handle: add()place() args: [EiffelTower, Paris]
        result: Paris
handle: whereIs() args: [EiffelTower]
        result: Paris
done
```

Note that if the server callback raises an exception, it will crash the whole server.

3.3 Adding Transactions

Armstrong's server2 adds transaction semantics: if the requested operation raises an exception, the server loop continues with the *original* value of the state. In contrast, if the requested operation completes normally, the server continues with the *new* value of the state.

Lacking inheritance, the only way that Armstrong can explain this to his readers is to present the *entire text* of a new module, server2. The reader is left to compare each function in server2 with the prior version in server1. The start functions seem to be identical; the rpc functions are similar, except that the receive clause in server2 has been extended to accommodate an additional component in the reply messages. The function loop seems to be completely different.

In the Grace version, the differences are much easier to find. In the *transactionServer* module, the class server is derived from *basicServer*'s server using inheritance:

```
module "transactionServer"
import "mirrors" as mirrors
import "basicServer" as basic

type Request = basic.Request

class server(callbackName:String) {
    inherits basic.server(callbackName)
        alias basicHandle = handle

    method handle(request:Request) is override {
        try {
            basicHandle(request)
        } catch { why ->
            log "Error — server crashed with {why}"
            "!CRASH!"
        }
    }
}
```

The handle method is overridden, but *nothing else changes*. It's easy to see that the extent of the change is the addition of the try()catch() clause to the handle method.

If we now try and make bogus requests:

```
import "transactionServer" as transaction

class request(methodName)withArgs(args) {
    method name { methodName }
    method arguments { args }
}
def queue = [
    request "add()place()" withArgs ["BuckinghamPalace", "London"],
    request "add()place()" withArgs ["EiffelTower", "Paris"],
    request "whereIs()" withArgs ["EiffelTower"],
    request "boojum()" withArgs ["EiffelTower"],
    request "whereIs()" withArgs ["BuckinghamPalace"]
]
print "starting transactionServer"
transaction.server("nameServer").serverLoop(queue)
print "done"
```

they will be safely ignored:

```
starting transactionServer
handle: add()place() args: [BuckinghamPalace, London]
        result: London
handle: add()place() args: [EiffelTower, Paris]
        result: Paris
handle: whereIs() args: [EiffelTower]
        result: Paris
Error — server crashed with NoSuchMethod: no method boojum()state in mir-
ror for a callback
```

```
handle: boojum() args: [EiffelTower]
      result: !CRASH!
handle: whereIs() args: [BuckinghamPalace]
      result: London
done
```

Armstrong emphasizes that the same server callback can be run under both the basic server and the transaction sever. This is also true for the Grace version, but that's not what we wish to emphasize. Our point is that inheritance makes it much easier to understand the critical differences between *basicServer* and *transactionServer* than does rewriting the whole server, as Armstrong is forced to do.

3.4 The Hot-Swap Server

Armstrong's server3 adds "hot swapping" to his server1; once again he is forced to rewrite the whole server from scratch, and the reader must compare the two versions of the code, line by line, to find the differences. Our Grace version instead adds hot swapping to the transactionServer, again using inheritance.

module "hotSwapServer"

```
import "mirrors" as mirrors
import "transactionServer" as base

type Request = base.Request

class server(callbackName:String) {
    inherits base.server(callbackName)
        alias baseHandle = handle

    method handle(request:Request) is override {
        if ( request.name == "!HOTSWAP!" ) then {
            def newCallback = request.arguments.first
            startUp(newCallback)
            "{newCallback} started."
        } else {
            baseHandle(request)
        }
    }
}
```

In *hotSwapServer*, class server overrides the handle method with a version that checks for the special request !HOTSWAP!. Other requests are delegated to the handle method inherited from *transactionServer*. Once again, it is clear that *nothing else changes*.

Now we can try to change the name server into a calculation server:

```
import "hotSwapServer" as hotSwap

class request(methodName) withArgs(args) {
    method name { methodName }
```

```
        method arguments { args }
    }
    def queue = [
        request "add()place()" withArgs ["EiffelTower", "Paris"],
        request "whereIs()" withArgs ["EiffelTower"],
        request "!HOTSWAP!" withArgs ["calculator"],
        request "whereIs()" withArgs ["EiffelTower"],
        request "add()" withArgs [3],
        request "add()" withArgs [4]
    ]
    print "starting hotSwapServer"
    hotSwap.server("nameServer").serverLoop(queue)
    print "done"
```

Here is the output:

```
starting hotSwapServer
handle: add()place() args: [EiffelTower, Paris]
        result: Paris
handle: whereIs() args: [EiffelTower]
        result: Paris
handle: !HOTSWAP! args: [calculator]
        result: calculator started.
Error — server crashed with NoSuchMethod: no method whereIs()state in mir-
ror for a callback
handle: whereIs() args: [EiffelTower]
        result: !CRASH!
handle: add() args: [3]
        result: 3
handle: add() args: [4]
        result: 7
done
```

3.5 Summary

In summary, we can say that what Armstrong found necessary to present as a series of completely separate programs, can be conveniently expressed in Grace as one program that uses inheritance. As a consequence, the final goal — a hot-swappable server that supports transactions — cannot be found in a single, monolithic piece of code, but instead is a composition of three modules. But far from being a problem, this is an *advantage*. The presentation using inheritance lets each feature be implemented, and understood, separately. Indeed, the structure of the code mirrors quite closely the structure of Armstrong's own exposition in his book.

4 Discussion and Conclusion

We will be the first to admit that a few small examples prove nothing. Moreover, we are aware that there are places where our argument needs improvement. Here we comment on two of these.

4.1 From the Abstract to the Concrete

Inheritance can be used in many ways other than the way illustrated here. Our own experience with inheritance is that we sometimes start out with a concrete class X, and then write a new concrete class Y doing something similar. We then abstract by creating an (abstract) superclass A that contains the attributes that A and B have in common, and rewrite X and Y to inherit from A.

When programmers have a lot of experience with the domain, they might even *start out* with the abstract superclass. We did exactly this when building the Grace collections library, which contains abstract superclasses like iterable, which can be inherited by any concrete class that defines an iterator method. Syntax aside, an abstract superclass is essentially a functor returning a set of methods, and the abstract method(s) of the superclass are its *explicit* parameters. Used in this way, inheritance is little more than a convenient parameterization construct.

If we admit that this use of inheritance as a parameterization construct is common, the reader may wonder why have we used the bulk of this paper to emphasize the use of inheritance as a construct for differential programming. The answer is that we wish to capture the *essence* of inheritance. According to *The Blackwell Dictionary of Western Philosophy*, "essence is the property of a thing without which it could not be what it is." (Bunnin and Yu 2004, p. 223). In this sense, the essence of inheritance is its ability to override a concrete entity, and thus effectively turn a constant into a parameter. There is nothing at all wrong with using inheritance to share abstract entities designed for reuse, that is, to move from the abstract to the concrete. But what makes inheritance unique is its ability to create parameters out of constants — what we have called ex post facto parameterization.

4.2 What About Types?

Anyone who has read both Wadler's "The Essence of Functional Programming" and our Sect. 2 can hardly fail to notice that Wadler's code is rich with type information, while our code omits many type annotations. This prompts the question: can the inheritance techniques that we advocate be well-typed?

The way that inheritance is used in Sect. 2 — overriding the definition of the monad in descendants — poses some fundamental difficulties for typechecking. As the components of the expression class are changed to add new functionality, the types that describe those components also change. For example, the parameters l and r of the class div in module "simpleExpressions" have eval methods that answer numbers, whereas the corresponding parameters to div in module "monadicExpressions" have eval methods that answer monads. If there were other methods that used the results of eval, these would also need to be changed to maintain type correctness. Even changing a type to a subtype is not safe in general, because the type may be used to specify a requirement on a parameter as well as a property of a result.

Allowing an overriding method in a subclass to have a type different from that of the overridden method clashes with one of the sacred cows of type-theory:

"modular typechecking" (Cardelli 1997). The idea behind modular typechecking is that each component of the program can be typechecked once and for all, regardless of how and where it is used. Since the methods of an object are generally interdependent, it seems clear that we cannot permit unconstrained changes to the type of a method after the typecheck has been performed. This runs contrary to the need to override methods to generalize functionality, as we have done in these examples.

Two paths are open to us. The first, exemplified by Ernst (2001), Bruce (2003), and Nystrom et al. (2004), is to devise clever restrictions that permit both useful overriding and early typechecking. The second is to question the value of modular typechecking.

While modular typechecking seems obviously desirable for a subroutine library, or for a class that we are to *instantiate*, its importance is less clear for a class that we are going to *inherit*. After all, inheritance lets us interfere with the internal connections between the methods of the inherited class, substituting arbitrary new code in place of existing code. It is quite possible to break the inherited class by improper overriding: it is up to the programmer who uses inheritance to understand the internal structure of the inherited class. In this context, running the typechecker over both the inherited code *and* the inheriting code seems like a small price to pay for the security that comes from being able to override method types and have the resulting composition be certified as type-safe.

4.3 Further Extensions

An issue we have not yet discussed is the difficulty of combining separate extensions, for example, including both output and exceptions in the expression language of Sect. 2. Some form of multiple inheritance would seem to be indicated, but most attempts at multiple inheritance run into difficulties when the inherited parts conflict. A particularly troublesome situation is the "diamond problem" (Bracha and Cook 1990; Snyder 1986), also known as "fork-join inheritance" (Sakkinen 1989); this occurs when a class inherits a state component from the *same* base class via multiple paths.

In our view, the most promising solution seems to be to restrict what can be multiply inherited. For example, traits, as defined by Ducasse, Nierstrasz, Schärli, Wuyts, and Black (2006), consist of collections of methods that can be "mixed into" classes. Traits do not contain state; they instead access state declared elsewhere using appropriate methods. The definition of a trait can also specify "required methods" that must be provided by the client. In exchange for these modest restrictions on expressiveness, traits avoid almost all of the problems of multiple inheritance. Traits have been adopted by many recent languages; we are in the process of adding them to Grace. Delegation may provide an alternative solution, but seems to be further from the mainstream of object-oriented language development.

One of the anonymous reviewers observed that Edwards' *Subtext* (2005a) is also based on the idea that humans are better at concrete thinking than at abstraction. Subtext seeks to “decriminalize copy and paste” (Edwards 2005b) by supporting *post hoc* abstraction. Because Subtext programs are structures in a database rather than pieces of text, the same program can be viewed in multiple ways. Thus, whether someone reading a program sees a link to an abstraction or an embedded implementation of that abstraction depends on a check-mark on a style sheet, not on a once-and-for-all decision made by the writer of the program. Moreover, the view can be changed in real time to suit the convenience of the reader. Similar ideas were explored by Black and Jones (2004).

As we have shown in this article, inheritance can lead to highly-factored code. This can be a double-edged sword. We have argued that the factoring is advantageous, because it lets the reader of a program digest it in small, easily understood chunks. But it can sometimes also be a disadvantage, because it leaves the reader with a less integrated view of the program. The ability to view a program in multiple ways can give us the best of both worlds, by allowing the chunks either to be viewed separately, or to be “flattened” into a single larger chunk without abstraction boundaries. A full exploration of these ideas requires not only that we “step away from the ASR-33” (Kamp 2010), but that we move away from textual representations of programs altogether.

4.4 Conclusion

Introducing a new concept by means of a series of examples is a technique as old as pedagogy itself. All of us were taught that way, and teach that way. We do not start instruction in mathematics by first explaining the abstractions of rings and fields, and then introducing arithmetic on the integers as a special case. On the contrary: we introduce the abstractions only after the student is familiar with not one, but several, concrete examples.

As Baniassad and Myers (2009) have written,

> the code that constitutes a program actually forms a higher-level, program-specific language. The symbols of the language are the abstractions of the program, and the grammar of the language is the set of (generally unwritten) rules about the allowable combinations of those abstractions. As such, a program is both a language definition, and the only use of that language. This specificity means that reading a never-before encountered program involves learning a new natural language

It follows that when we write a program, we are teaching a new language. Isn’t it our duty to use all available technologies to improve our teaching? So, next time you find yourself explaining a complex program by introducing a series of simpler programs of increasing complexity, think how that *explanation*, as well as the final program, could be captured in your programming language. And if your chosen programming language does not provide the right tools, ask whether it could be improved.

References

Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)

Baniassad, E., Myers, C.: An exploration of program as language. In: OOPSLA 2009 Companion, pp. 547–556 (2009)

Bierman, G., Parkinson, M., Boyland, J.: Upgradej: incremental typechecking for class upgrades. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 235–259. Springer, Heidelberg (2008)

Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by example. Square Bracket Associates (2009). http://pharobyexample.org

Black, A.P., Bruce, K.B., Homer, M., Noble, J.: Grace: the absence of (inessential) difficulty. In: Onward! 2012: Proceedings 12th SIGPLAN Symposium on New Ideas in Programming and Reflections on Software, pp. 85–98. ACM, New York (2012). http://doi.acm.org/10.1145/2384592.2384601

Black, A.P., Jones, M.P.: The case for multiple views. In: Grundy, J.C., Welland, R., Stoeckle, H. (eds.) ICSE Workshop on Directions in Software Engineering Environments (WoDiSEE), May 2004

Bracha, G., Cook, W.: Mixin-based inheritance. In: ECOOP/OOPSLA 1990, pp. 303–311. ACM Press, Ottawa (1990). http://java.sun.com/people/gbracha/oopsla90.ps

Bruce, K.B.: Some challenging typing issues in object-oriented languages. Electr. Notes Theor. Comput. Sci. **82**(7), 1–29 (2003). http://dx.doi.org/10.1016/S1571-0661(04)80799-0

Bunnin, N., Yu, J.: The Blackwell Dictionary of Western Philosophy. Blackwell, Oxford (2004)

Cardelli, L.: Program fragments, linking, and modularization. In: Conference Record of POPL 1997: Twenty-fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, pp. 266–277, January 1997

Cook, W., Palsberg, J.: A denotational semantics of inheritance and its correctness. In: Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1989, pp. 433–443. ACM Press, New Orleans (1989)

Cook, W.R., Hill, W.L., Canning, P.S.: Inheritance is not subtyping. In: Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages, San Francisco, CA, USA, pp. 125–135 (1990)

Cox, B.J.: Object Oriented Programming: An Evolutionary Approach. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: a mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. **28**(2), 331–388 (2006)

Edwards, J.: Subtext: uncovering the simplicity of programming. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 505–518. ACM, New York (2005a). http://doi.acm.org/10.1145/1094811.1094851

Edwards, J.: Subtext: uncovering the simplicity of programming (2005b). video demonstration. http://subtextual.org/demo1.html

Ernst, E.: Family polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)

Kamp, P.H.: Sir, please step away from the ASR-33!. ACM Queue **8**(10), 40:40–40:42 (2010). http://doi.acm.org/10.1145/1866296.1871406

Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, pp. 99–115. ACM, New York (2004). http://doi.acm.org/10.1145/1028976.1028986

Reynolds, J.C.: Definitional interpreters for higher order programming languages. In: Proceedings of ACM 1972 Annual Conference, pp. 717–740 (1972)

Reynolds, J.C.: The essence of ALGOL. In: de Bakker, J.W., van Vliet, J.C. (eds.) Proceedings of the International Symposium on Algorithmic Languages, pp. 345-372. North-Holland (1981). Reprinted in Algol-like Languages, O'Hearn, P.W., Tennent, R.D. (eds.), vol. 1, pp. 67-88, Birkhäuser (1997)

Sakkinen, M.: Disciplined inheritance. In: Cook, S. (ed.) ECOOP 1989: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, 10–14 July 1989, pp. 39–56. Cambridge University Press, Cambridge (1989)

Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In: OOPSLA 1986, pp. 38–45 (1986). Also published as ACM SIGPLAN Notices, vol. 21, November 1986

Wadler, P.: The essence of functional programming. In: Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages, pp. 1–14. ACM Press, Albuquerque (1992)

Wadler, P.: The first monad tutorial, December 2013. http://homepages.inf.ed.ac.uk/wadler/papers/yow/monads-scala.pdf

Subtyping Supports Safe Session Substitution

Simon J. Gay[(✉)]

School of Computing Science, University of Glasgow, Glasgow, UK
Simon.Gay@glasgow.ac.uk

Abstract. Session types describe the structure of bi-directional point-to-point communication channels by specifying the sequence and format of messages on each channel. A session type defines a communication protocol. Type systems that include session types are able to statically verify that communication-based code generates, and responds to, messages according to a specified protocol. It is natural to consider subtyping for session types, but the literature contains conflicting definitions. It is now folklore that one definition is based on safe substitutability of channels, while the other is based on safe substitutability of processes. We explain this point in more detail, and show how to unify the two views.

1 Prologue

My first encounter with Phil Wadler was at the Marktoberdorf Summer School in 1992, where he lectured on monads in functional programming. His course had the characteristically punning title "Church and State: taming effects in functional languages", and during his first lecture he removed his shirt to reveal a T-shirt printed with the category-theoretic axioms for a monad. The following year I met him again, at POPL in Charleston, South Carolina. He presented the paper "Imperative Functional Programming", also covering the monadic approach and co-authored with Simon Peyton Jones, which in 2003 won the award for the most influential paper from POPL 1993.

At that time, in the early 1990s, Phil was a professor at the University of Glasgow, and I was a PhD student at Imperial College London, attending POPL to present my own very first paper, which was about types for pi calculus. We PhD students in the Imperial theory group had a view of the world in which a small number of other departments were admired for their theoretical research, and the rest were ignored. The Glasgow functional programming group was admired, as was the LFCS in Edinburgh, where Phil now works. More than twenty years later, I myself am a professor at the University of Glasgow, but when I arrived here as a lecturer in 2000, Phil had already departed for the USA. When he returned to Scotland to take a chair in theoretical computer science

S.J. Gay—Supported by EPSRC (EP/K034413/1 "From Data Types to Session Types: A Basis for Concurrency and Distribution" and EP/L00058X/1 "Exploiting Parallelism through Type Transformations for Hybrid Manycore Systems") and COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems".

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 95–108, 2016.
DOI: 10.1007/978-3-319-30936-1_5

at Edinburgh, we started the Scottish Programming Languages Seminar, which is still active. Eventually the opportunity for a technical collaboration arose, when Phil developed an interest in session types because of the Curry-Howard correspondence between session types and linear logic, discovered by Luís Caires and Frank Pfenning. This led to a successful EPSRC grant application by Phil, myself, and Nobuko Yoshida, which is still running.

I am delighted to contribute this paper to Phil's 60th birthday Festschrift.

2 Introduction

Session types were introduced by Honda et al. (1998; 1994), based on earlier work by Honda (1993), as a way of expressing communication protocols type-theoretically, so that protocol implementations can be verified by static typechecking. The original formulation, now known as binary session types, covers pairwise protocols on bidirectional point-to-point communication channels. Later, the theory was extended to multiparty session types (Honda et al., 2008), in order to account for collective protocols among multiple participants.

Gay and Hole (1999; 2005) extended the theory of session types by adding subtyping, to support more flexible interaction between participants in a protocol. Many papers now include subtyping. However, not all of them use the same definition as Gay and Hole; some papers give the opposite variance to the session type constructors. The alternative definition first appears in the work of Carbone et al. (2007) and has been used in many papers by Honda et al., including one by Demangeon and Honda (2011) whose main focus is subtyping. For example, consider two simple session types with branching (external choice), one which offers a choice between labels a and b, and one which offers label a alone. According to Gay and Hole's definition, the session type constructor for branch is covariant in the set of labels, so that $\&\langle \mathsf{a} : \mathsf{end}\rangle \leqslant \&\langle \mathsf{a} : \mathsf{end}, \mathsf{b} : \mathsf{end}\rangle$. According to the Honda et al. definition, branch is contravariant, so that $\&\langle \mathsf{a} : \mathsf{end}, \mathsf{b} : \mathsf{end}\rangle \sqsubseteq \&\langle \mathsf{a} : \mathsf{end}\rangle$. Each paper defines a type system and an operational semantics and proves that typability guarantees runtime safety. How can both definitions of subtyping be correct?

The fundamental way of justifying the definition of subtyping for any language is by Liskov and Wing's (1994) concept of safe substitutability. Type T is a subtype of type U if a value of type T can be safely used whenever a value of type U is expected, where "safely" means without violating the runtime safety property that the type system guarantees. In this paper we will analyse safe substitutability in the setting of session types, to see how each definition of subtyping can be justified. The answer lies in careful consideration of what is being safely substituted. Gay and Hole's definition is based on safe substitution of communication channels, whereas Honda et al.'s definition is based on safe substitution of processes. This situation contrasts with that of λ-calculus, where the syntax consists only of expressions and there is no choice about which kind of entity is being substituted.

The remainder of the paper is structured as follows. Section 3 reviews the concepts of session types and motivates the need for subtyping. Section 4 considers safe substitution of channels, and gives several justifications of the Gay and Hole definition. Section 5 considers safe substitution of processes, and justifies the Honda et al. definition. Section 6 describes a session type system for higher-order communication, introduced by Mostrous and Yoshida (2007; 2015), and shows how that setting allows the two definitions of subtyping to be reconciled. Finally, Sect. 7 concludes. In order to reduce the volume of definitions, we focus on intuition and explanation, and do not give full details of the languages and type systems that we discuss.

3 Session Types and Subtyping

To illustrate the need for a theory of subtyping for session types, consider the example of a server for mathematical operations (Gay and Hole, 2005). There are two services: addition of integers, which produces an integer result, and testing of integers for equality, which produces a boolean result. A client and a server run independently and communicate on a bidirectional point-to-point channel, which has two endpoints, one for the client and one for the server. The protocol is specified by defining a pair of dual session types, one for each endpoint. The type of the server's endpoint is S, defined as follows.

$$\begin{aligned} S = \&\langle\ & \mathsf{add}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end},\\ & \mathsf{eq}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{aligned}$$

The $\&\langle\dots\rangle$ constructor specifies that a choice is offered between, in this case, two options, labelled add and eq. It is an external choice, often called "branch": the server must be prepared to receive either label. Each label leads to a type describing the continuation of the protocol, different in each case. The constructors $?[\dots]$ and $![\dots]$ indicate receiving and sending, respectively. Sequencing is indicated by . and end marks the end of the interaction. For simplicity this protocol allows only a single service invocation.

In a run of the protocol, the first message is one of the labels plus and eq, sent from the client to the server. Subsequently, the client sends messages according to the continuation of the label that was chosen, and then receives a message from the server. The type of the client's endpoint is $\overline{S}$, the dual of S.

$$\begin{aligned} \overline{S} = \oplus\langle\ & \mathsf{add}:![\mathsf{int}].![\mathsf{int}].?[\mathsf{int}].\mathsf{end},\\ & \mathsf{eq}:![\mathsf{int}].![\mathsf{int}].?[\mathsf{bool}].\mathsf{end}\ \rangle \end{aligned}$$

The structure is the same, but the directions of the communications are reversed. The $\oplus\langle\dots\rangle$ constructor is an internal choice, often called "select" or "choice": the client can decide which label to send.

Suppose that the server is upgraded by the addition of new services and an extension of an existing service. The equality test can now handle floating point numbers, and there is a multiplication service. The type of the new server is S'.

$$\begin{aligned}S' = \&\langle\ & \mathsf{add}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end},\\ & \mathsf{mul}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end},\\ & \mathsf{eq}:?[\mathsf{float}].?[\mathsf{float}].![\mathsf{bool}].\mathsf{end}\ \rangle\end{aligned}$$

The requirement for a theory of subtyping is to allow a client that has been typechecked with type $\overline{S}$ to interact with the new server. In Sects. 4 and 5 we will discuss the two approaches to subtyping that have appeared in the literature, and explain how they account for this example.

Another possibility is to upgrade the type of the addition service to

$$\mathsf{add}:?[\mathsf{float}].?[\mathsf{float}].![\mathsf{float}].\mathsf{end}.$$

This motivates the development of a theory of bounded polymorphism (Gay, 2008) that allows the addition service to have type

$$\mathsf{add}(X \leqslant \mathsf{float}):?[X].?[X].![X].\mathsf{end}$$

so that the client chooses an instantiation of the type variable X as either int or float and then proceeds appropriately. We do not consider bounded polymorphism further in the present paper.

4 Channel-Oriented Subtyping

In Sect. 3 we defined session types for the mathematical server and client, without choosing a language in which to implement them. We will now give pi calculus definitions, following the example of Gay and Hole (2005). The core of the server process is $\mathsf{serverbody}$, parameterised by its channel endpoint x.

$$\begin{aligned}\mathsf{serverbody}(x) = x\triangleright\{\ & \mathsf{add}:x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u+v].\mathbf{0},\\ & \mathsf{eq}:x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u=v].\mathbf{0}\ \}.\end{aligned}$$

The branching construct $x\triangleright\{\ldots\}$ corresponds to the type constructor $\&\langle\ldots\rangle$; it receives a label and executes the code corresponding to that label. The rest of the syntax is pi calculus extended with arithmetic operations. The process $x?[u{:}\mathsf{int}].P$ receives an integer value on channel x, binds it to the name u, and then executes P. The process $x![u+v].Q$ sends the value of the expression $u+v$ on channel x and then executes Q. Finally, $\mathbf{0}$ is the terminated process.

The type system derives judgements of the form $\Gamma \vdash P$, where Γ is an environment of typed channels and P is a process. A process does not have a type; it is either correctly typed or not. The typing of $\mathsf{serverbody}$ is

$$x:S \vdash \mathsf{serverbody}(x)$$

where S is the session type defined in Sect. 3. Although we are not showing the typing rules here, it should be clear that the process and the type have the same communication structure.

Similarly, the core of one possible client process, which uses the addition service, is $\mathsf{clientbody}$, again parameterised by its channel endpoint.

$$\mathsf{clientbody}(x) = x \triangleleft \mathsf{add}.x![2].x![3].x?[u{:}\mathsf{int}].\mathbf{0}$$

The typing is

$$x{:}\overline{S} \vdash \mathsf{clientbody}(x).$$

The client and server need to establish a connection so that they use opposite endpoints c^- and c^+ of channel c. The original formulation of session types (Takeuchi et al., 1994) used matching $\mathsf{request}$ and accept constructs, but here we follow Gay and Hole's approach in which the client uses standard pi calculus name restriction to create a fresh channel and then sends one endpoint to the server. This requires a globally known channel a of type $\widehat{\ }[S]$ (the standard pi calculus channel type constructor). The complete definitions and typings are

$$\begin{aligned} \mathsf{server}(a) &= a?[y{:}S].\mathsf{serverbody}(y) \\ \mathsf{client}(a) &= (\nu x : S)(a![x^+].\mathsf{clientbody}(x^-)) \\ a{:}\widehat{\ }[S] &\vdash \mathsf{server}(a) \\ a{:}\widehat{\ }[S] &\vdash \mathsf{client}(a) \end{aligned}$$

where $(\nu x : S)$ binds x^+ with type S and x^- with type $\overline{S}$.

In Sect. 3 we defined the session type S' for an upgraded server. The corresponding definition of $\mathsf{newserverbody}$ is

$$\begin{aligned} \mathsf{newserverbody}(x) = x \triangleright \{\ & \mathsf{add}{:}\,x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u+v].\mathbf{0}, \\ & \mathsf{mul}{:}\,x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u*v].\mathbf{0}, \\ & \mathsf{eq}{:}\,x?[u{:}\mathsf{float}].x?[v{:}\mathsf{float}].x![u=v].\mathbf{0}\ \}. \end{aligned}$$

and we have $x : S' \vdash \mathsf{newserverbody}(x)$. Now $\mathsf{newserver}$ is defined in terms of $\mathsf{newserverbody}$ in the same way as before.

Clearly client can interact safely with $\mathsf{newserver}$; the question is how to extend the type system so that

$$a{:}\widehat{\ }[S'] \vdash \mathsf{client}(a) \mid \mathsf{newserver}(a).$$

In Gay and Hole's theory, this is accounted for by considering safe substitution of channels. It is safe for $\mathsf{newserverbody}$ to interact on a channel of type S instead of the channel of type S' that it expects; the substitution means that $\mathsf{newserverbody}$ never receives label mul and always receives integers in the eq service, because the process at the other endpoint of the channel is using it with type $\overline{S}$. Subtyping is defined according to this concept of safe substitution, giving $S \leqslant S'$. We see that branch (external choice) is covariant in the set of labels, and input is covariant in the message type. If we imagine changing the definitions so that the server creates the channel and sends one endpoint to the client, we find that select (internal choice) is contravariant in the set of labels and output is contravariant in the message type. These are the variances established by Pierce

$$\frac{}{\mathsf{end} \leqslant \mathsf{end}}\ \textsc{S-End} \qquad \frac{T \leqslant U \qquad U \leqslant T}{\ulcorner[T] \leqslant \ulcorner[U]}\ \textsc{S-Chan}$$

$$\frac{T \leqslant U \qquad V \leqslant W}{?[T].V \leqslant ?[U].W}\ \textsc{S-InS} \qquad \frac{m \leqslant n \qquad \forall i \in \{1,\ldots,m\}.S_i \leqslant T_i}{\&\langle\, l_i : S_i \,\rangle_{1 \leqslant i \leqslant m} \leqslant \&\langle\, l_i : T_i \,\rangle_{1 \leqslant i \leqslant n}}\ \textsc{S-Branch}$$

$$\frac{U \leqslant T \qquad V \leqslant W}{![T].V \leqslant ![U].W}\ \textsc{S-OutS} \qquad \frac{m \leqslant n \qquad \forall i \in \{1,\ldots,m\}.S_i \leqslant T_i}{\oplus\langle\, l_i : S_i \,\rangle_{1 \leqslant i \leqslant n} \leqslant \oplus\langle\, l_i : T_i \,\rangle_{1 \leqslant i \leqslant m}}\ \textsc{S-Choice}$$

Fig. 1. Subtyping for non-recursive types.

and Sangiorgi (1996) in the first work on subtyping for (non-session-typed) pi calculus. Furthermore, it is clear that all of the session type constructors are covariant in the continuation type, simply because after the first communication we can again consider safe substitutability with the continuation types in the substituting and substituted positions. The definition of subtyping for non-recursive session types is summarised in Fig. 1, which includes the invariant rule for standard channel types. For recursive types, the same principles are used as the basis for a coinductive definition (Gay and Hole, 2005).

This definition of subtyping is justified by the proof of type safety in Gay and Hole's system. In the rest of this section, we give alternative technical justifications for the definition.

4.1 Safe Substitutability of Channels, Formally

Gay and Hole express safe substitutability of channels for channels by (a more general form of) the following result, in which z^p is either z^+ or z^-.

Lemma 1 (Substitution (Gay and Hole, 2005, Lemma 8)). *If* $\mathit{\Gamma}, w\!:\!W \vdash P$ *and* $Z \leqslant W$ *and* $z^p \notin dom(\mathit{\Gamma})$ *then* $\mathit{\Gamma}, z^p\!:\!Z \vdash P\{z^p/w\}$.

They define subtyping first, taking into account recursive types by giving a coinductive definition based on the principles of Fig. 1, and then prove type safety, for which Lemma 1 is a necessary step. Alternatively, we can take Lemma 1 as a statement of safe substitutability, i.e. a specification of the property that we want subtyping to have, and then derive the definition of subtyping by considering how to prove the lemma by induction on typing derivations. For example, in the inductive case for an input process, the relevant typing rule is T-InS. The occurrence of subtyping in the rule is based on safe substitutability of channels: the received value of type T must be substitutable for the bound variable y of type U.

$$\frac{\mathit{\Gamma}, x^p\!:\!S, y\!:\!U \vdash P \qquad T \leqslant U}{\mathit{\Gamma}, x^p\!:\!?[T].S \vdash x^p?[y\!:\!U].P}\ \textsc{T-InS}$$

The specific case in the proof of the lemma is that we substitute z^p for x in the process $x?[y{:}U].Q$. The typing derivation for $x?[y{:}U].Q$ concludes with an application of rule T-InS.

$$\frac{\Gamma, x{:}S, y{:}U \vdash Q \qquad T \leqslant U}{\Gamma, x{:}?[T].S \vdash x?[y{:}U].Q}\ \text{T-InS}$$

To prove this case of Lemma 1 we need to establish $\Gamma, z^p : Z \vdash z^p?[y{:}U].Q\{z^p/x\}$, which has to have a derivation ending with an application of rule T-InS:

$$\frac{\Gamma, z^p{:}S', y{:}U \vdash Q\{z^p/x\} \qquad A \leqslant U}{\Gamma, z^p{:}?[A].S' \vdash z^p?[y{:}U].Q\{z^p/x\}}\ \text{T-InS}$$

From the assumptions $Z \leqslant ?[T].S$ and $T \leqslant U$, we need to be able to conclude $Z = ?[A].S'$ and $S' \leqslant S$ (to use the induction hypothesis) and $A \leqslant U$. To go from $T \leqslant U$ to $A \leqslant U$, for arbitrary U, requires $A \leqslant T$. Overall, in order to make the proof of Lemma 1 work, we need:

if $Z \leqslant ?[T].S$ then $Z = ?[A].S'$ with $A \leqslant T$ and $S' \leqslant S$.

This is essentially one of the clauses in the coinductive definition of subtyping (Gay and Hole, 2005, Definition 3); the only difference is that the coinductive definition unfolds recursive types.

As another example we can consider the case of branch, with the typing rule T-Offer. It contains the raw material of subtyping in the form of the condition $m \leqslant n$, which guarantees that the received label is within the range of possibilities.

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.(\Gamma, x^p{:}S_i \vdash P_i)}{\Gamma, x^p{:}\&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m} \vdash x^p \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

The relevant case in the proof of Lemma 1 is that we substitute z^p for x in the process $x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}$. The typing derivation for $x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}$ concludes with an application of rule T-Offer.

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.(\Gamma, x{:}S_i \vdash P_i)}{\Gamma, x{:}\&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m} \vdash x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

We need to establish $\Gamma, z^p{:}Z \vdash z^p \triangleright \{\, l_i : P_i\{z^p/x\} \,\}_{1\leqslant i\leqslant n}$, whose derivation must end with an application of T-Offer:

$$\frac{r \leqslant n \quad \forall i \in \{1, \ldots, r\}.(\Gamma, z^p{:}Z_i \vdash P_i)}{\Gamma, z^p{:}\&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r} \vdash z^p \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

From the assumptions $Z \leqslant \&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m}$ and $m \leqslant n$ we need to be able to conclude $Z = \&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r}$ and $\forall i \in \{1 \ldots r\}.Z_i \leqslant S_i$ and $r \leqslant n$. Therefore we need:

$$\begin{aligned}
[\![\mathsf{end}]\!] &= \mathsf{cap}_\emptyset[\,] \\
[\![?[T].S]\!] &= \mathsf{cap}_\mathsf{i}[[\![T]\!], [\![S]\!]] \\
[\![![T].S]\!] &= \mathsf{cap}_\mathsf{o}[[\![T]\!], [\![\overline{S}]\!]] \\
[\![\&\langle l_1 : S_1, \ldots, l_n : S_n\rangle]\!] &= \mathsf{cap}_\mathsf{i}[\langle l_1 : [\![S_1]\!], \ldots, l_n : [\![S_n]\!]\rangle] \\
[\![\oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle]\!] &= \mathsf{cap}_\mathsf{o}[\langle l_1 : [\![\overline{S_1}]\!], \ldots, l_n : [\![\overline{S_n}]\!]\rangle]
\end{aligned}$$

Fig. 2. Translation of session types into linear and variant types (Dardha et al., 2012). Data types such as int and bool are translated into themselves.

if $Z \leqslant \&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m}$ then $Z = \&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r}$ with $r \leqslant m$ and $\forall i \in \{1 \ldots r\}.Z_i \leqslant S_i$.

Again this is essentially a clause in the coinductive definition of subtyping. The other clauses are obtained similarly by considering other possibilities for the structure of P, with the exception of the clause for end which requires proving (straightforwardly) that $\Gamma, x{:}\mathsf{end} \vdash P$ if and only if $\Gamma \vdash P$ and $x \notin \mathit{fn}(P)$. In order to type as many processes as possible we take the largest relation satisfying this clause, which leads to the coinductive definition.

4.2 Channel-Oriented Subtyping by Translation

Kobayashi (2003) observed informally that the branch and choice constructors of session types can be represented by variant types, which have been considered in pi calculus independently of session types (Sangiorgi and Walker, 2001). Specifically, making a choice corresponds to sending one label from a variant type, and offering a choice corresponds to a case-analysis on a received label. With this representation, Gay and Hole's definition of subtyping follows by combining the standard definition of subtyping for variants in λ-calculus (Pierce, 2002) with Pierce and Sangiorgi's (1996) definition of subtyping for input and output types in pi calculus. Dardha et al. (2012) developed this idea in detail[1]. They showed that not only subtyping, but also polymorphism, can be derived from a translation of session types into linear pi calculus (Kobayashi et al., 1999) with variants.

The translation combines three ideas. First, in the linear pi calculus, a linear channel can be used for only one communication. To allow a session channel to be used for a series of messages, a continuation-passing style is used, so that each message is accompanied by a fresh channel which is used for the next message. Second, the linear pi calculus separates the capabilities for input (cap_i) and output (cap_o) on a channel. A session type corresponding to an initial input or output is translated into an input-only or output-only capability. Polyadic channel types of the form $\mathsf{cap}_\mathsf{i}[T_1, \ldots, T_n]$ are used. Third, branch and select session types are both translated into variant types $\langle l_1 : T_1, \ldots, l_n : T_n\rangle$, with the input or output capability as appropriate. The translation of types is defined in Fig. 2; note the use of the empty capability $\mathsf{cap}_\emptyset$ in the translation of end.

[1] See also Dardha's (2014) PhD thesis.

To illustrate the translation, consider the session types S and $\overline{S}$ from Sect. 3.

$$\begin{aligned}[\![S]\!] = \mathsf{cap_i}[\langle\ &\mathsf{add} : \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_o}[\mathsf{int}, \mathsf{cap}_\emptyset[\,]\,]\,]\,]\\ &\mathsf{eq} : \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_o}[\mathsf{bool}, \mathsf{cap}_\emptyset[\,]\,]\,]\,]\,\rangle\,]\end{aligned}$$

$$\begin{aligned}[\![\overline{S}]\!] = \mathsf{cap_o}[\langle\ &\mathsf{add} : \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_o}[\mathsf{int}, \mathsf{cap}_\emptyset[\,]\,]\,]\,]\\ &\mathsf{eq} : \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_i}[\mathsf{int}, \mathsf{cap_o}[\mathsf{bool}, \mathsf{cap}_\emptyset[\,]\,]\,]\,]\,\rangle\,]\end{aligned}$$

Duality appears as a reversal of input/output capability at the top level only, so that the message types of $[\![S]\!]$ and $[\![\overline{S}]\!]$ are the same.

The standard definition of subtyping for variant types is covariant in the set of labels. With the definitions in Fig. 2, considering that $\mathsf{cap_i}$ is covariant and $\mathsf{cap_o}$ is contravariant, this gives the same variances for branch and select that Gay and Hole define.

4.3 Channel-Oriented Subtyping by Semantic Subtyping

Castagna et al. (2009) use the idea of semantic subtyping (Frisch et al., 2002) to develop a set-theoretic model of session types in which subtyping is simply set inclusion. The foundation for their model is a duality relation $\bowtie$ on session types, which characterises safe communication. Gay and Hole generalised the duality function $\overline{(\cdot)}$ to a coinductively-defined relation $\perp$ (in order to handle recursive types) which requires exact matching of label sets and message types. For example, with the definitions in Sect. 3, we have $S \perp \overline{S}$ but not $S' \perp \overline{S}$. In contrast, $S' \bowtie \overline{S}$ because every message sent on an endpoint of type $\overline{S}$ will be understood by the process at the opposite endpoint of type S', and vice versa. Deriving the definition of subtyping from this definition of duality is technically more complicated because of the circularity that the definition of duality itself involves subtyping of message types, which in general can be session types. When the theory is worked out, the session type constructors have the same variances as in Fig. 1. The paper does not discuss the distinction between channel-oriented and process-oriented subtyping, but the fact that the definition of duality includes subtyping on message types, which can be channel types, implies that substitutability of channels is being considered.

5 Process-Oriented Subtyping

We consider the paper by Demangeon and Honda (2011) as a typical example that defines subtyping in the Honda et al. style. Like Dardha et al., they study subtyping for session types by translating into a simpler language. Apart from the definition of subtyping, the main difference is that their linear pi calculus combines variant types and input/output types so that a label is accompanied by a message. The intuition behind the Honda et al. definition of subtyping is easiest to understand by considering the type environment to be the type of a process, which we emphasize here by writing judgements in the form $P \vdash \Gamma$. Their subsumption result has a clear analogy with subtyping for λ-calculus.

Here, the relation $\sqsubseteq$ is the subtyping relation on session types, lifted pointwise to environments.

Proposition 1 (Subsumption (Demangeon and Honda, 2011, Proposition 9)). *If $P \vdash \Gamma$ and $\Gamma \sqsubseteq \Gamma'$ then $P \vdash \Gamma'$.*

Considering the type environment as a specification of the interactions that process must allow, we can return to the maths server to see that all of the following typings make sense.

$$\begin{aligned} \mathsf{serverbody}(x) &\vdash x : S \\ \mathsf{newserverbody}(x) &\vdash x : S' \\ \mathsf{newserverbody}(x) &\vdash x : S \end{aligned}$$

In the first two judgements, the type and the process match exactly. The third judgement states that the process with more behaviour, newserverbody, satisfies the requirement expressed by the original type S, i.e. it provides the add and eq services. In relation to subtyping, and Proposition 1, this means $S' \sqsubseteq S$, i.e.

$$\left.\begin{array}{l} \&\langle\ \mathsf{add} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad\ \ \mathsf{mul} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad\ \ \mathsf{eq} : ?[\mathsf{float}].?[\mathsf{float}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{array}\right\} \sqsubseteq \left\{\begin{array}{l} \&\langle\ \mathsf{add} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad\ \ \mathsf{eq} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{array}\right.$$

This is the opposite of the subtyping relationship for these types in Gay and Hole's theory, which we described in Sect. 4. Indeed, Demangeon and Honda define subtyping for branch types to be contravariant in the set of labels, and subtyping for select types is covariant. The example above also shows that input needs to be contravariant in the message type and output needs to be covariant, so that the subtyping relation is exactly the converse of Gay and Hole's definition. We emphasise that branch and input both involve receiving a value, so it makes sense that they have the same variance; similarly select and output. Demangeon and Honda actually define subtyping for input, which in their presentation is unified with branch, to be *covariant* in the message type, and conversely subtyping for output, which they unify with select, is *contravariant.* It seems that this is just a typographical error, as other papers in the Honda et al. style, for example by Chen et al. (2014), use the expected variance in the message type. In some definitions of subtyping, for example by Mostrous and Yoshida (2015), the variance depends on whether data or channels are being communicated; the technical details need further investigation to clarify this point.

Demangeon and Honda give further justification for their definition of subtyping by relating it to duality with the following result, stated here in a form that combines their Definition 10 and Proposition 12.

Proposition 2. *$S_1 \sqsubseteq S_2$ if and only if for all S, $S_1 \bowtie S \Rightarrow S_2 \bowtie S$.*

In words: if a process satisfies type S_1, then it also satisfies type S_2 if and only if all safe interactions with S_1 are also safe interactions with S_2.

The view of the session type environment as the specification of a process, and consideration of when one process satisfies the specification originally given for another process, correspond to working with safe substitution of processes, instead of safe substitution of channels as in Sect. 4. Explicitly, consider the following typing derivation for a server and client in parallel; we ignore the step of establishing the connection, because in Demangeon and Honda's system this is achieved by a symmetrical request/accept construct rather than by sending a channel endpoint.

$$\frac{\mathsf{serverbody}(x^+) \vdash x^+ : S \qquad \mathsf{clientbody}(x^-) \vdash x^- : \overline{S}}{\mathsf{serverbody}(x^+) \mid \mathsf{clientbody}(x^-) \vdash x^+ : S, x^- : \overline{S}}$$

Because we also have $\mathsf{newserverbody}(x^+) \vdash x^+ : S$, the process $\mathsf{newserverbody}(x^+)$ can be safely substituted for $\mathsf{serverbody}(x^+)$ in the derivation, i.e. in the typed system.

It should be possible to prove a formal result about safe substitutability of processes, along the following lines, where $C[\cdot]$ is a process context.

Conjecture 1. If $P \vdash \Gamma_P$ and $Q \vdash \Gamma_Q$ and $\Gamma_Q \sqsubseteq \Gamma_P$ and $C[P] \vdash \Gamma$ then $C[Q] \vdash \Gamma$.

6 Unifying Channel- and Process-Oriented Subtyping

Sections 4 and 5 described two separate type systems, with two definitions of subtyping. To better understand the relationship between these definitions, we use a language and type system in which both channel substitution and process substitution can be expressed internally, by message-passing. This requires higher-order communication, i.e. the ability to send a process as a message.

A natural way to develop a higher-order process calculus is to require a process, before being sent as a message, to be made self-contained by being abstracted on all of its channels. When a process is received, it can be applied to the channels that it needs, and then will start executing. This approach is realistic for distributed systems, as it does not assume that channels are globally available. Therefore we need a system that combines pi calculus and λ-abstraction, with session types. This combination has been studied by Mostrous and Yoshida (2015). We will now informally discuss typings in such a system.

In our presentation of Honda et al. style subtyping (Sect. 5) we used typing judgements of the form $P \vdash \Gamma$. We will now write this as

$$P \vdash \mathsf{proc}(\Gamma)$$

to emphasise the view of Γ as the type of P. In our presentation of Gay and Hole subtyping (Sect. 4) we used typing judgements of the form $\Gamma \vdash P$. We will now write this as

$$\Gamma \vdash P : \mathsf{proc}$$

to emphasise that P is being given the type which says that it is a correct process.

In Mostrous and Yoshida's language of higher-order processes we can abstract a process on its channels to obtain an expression with a function type, for example

$$\vdash \lambda x.\mathsf{server}(x) : S \rightarrow \mathsf{proc}$$

and

$$\vdash \lambda x.\mathsf{newserver}(x) : S' \rightarrow \mathsf{proc}.$$

Now we can identify $\mathsf{proc}(x : S)$ with $S \rightarrow \mathsf{proc}$. If we use Gay and Hole's subtyping relation, so that $S \leqslant S'$, then the standard definition of subtyping for function types gives $\mathsf{proc}(x{:}S') \leqslant \mathsf{proc}(x : S)$, which is Honda et al.'s definition. In other words, by moving to a setting in which safe substitution of both channels and processes can be expressed in terms of message-passing, we can explain the difference between Honda et al. subtyping and Gay/Hole subtyping by the fact that subtyping on function types is contravariant in the parameter.

7 Conclusion

We have discussed two definitions of subtyping for session types, both justified in terms of safe substitutability. The first definition, due to Gay and Hole, is based on safe substitutability of channels and has been derived in several ways in the literature. The second definition, due to Honda et al., is based on safe substitutability of processes. We have shown that the two definitions can be reconciled in a session type system for higher-order processes, of the kind defined by Mostrous and Yoshida. This is achieved by identifying the type of a process (its session environment, à la Demangeon and Honda) with a function type that abstracts all of its channels. The Honda et al. definition, in which branch types are contravariant, therefore arises from the combination of the Gay and Hole definition, in which branch types are covariant, and the standard definition of subtyping for function types, which is contravariant in the parameter.

We have used an informal presentation in order to focus on intuition and reduce the number of definitions being quoted from the literature. A more technically detailed account, including a proof of Conjecture 1, will be left for a future longer paper.

Acknowledgements. I am grateful to Nobuko Yoshida for discussion of the origin of the Honda et al. definition of subtyping, and to the anonymous referees for their comments.

References

Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007). http://dx.doi.org/10.1007/978-3-540-71316-6_2

Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: Proceedings of the 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 219–230. ACM (2009). http://doi.acm.org/10.1145/1599410.1599437

Chen, T., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: Proceedings of the 16th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 135–146. ACM (2014). http://doi.acm.org/10.1145/2643135.2643138

Dardha, O.: Type Systems for Distributed Programs: Components and Sessions. Ph.D. thesis, University of Bologna (2014). https://tel.archives-ouvertes.fr/tel-01020998

Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 139–150. ACM (2012)

Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 280–296. Springer, Heidelberg (2011). http://dx.doi.org/10.1007/978-3-642-23217-6_19

Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS), pp. 137–146. IEEE (2002). http://dx.doi.org/10.1109/LICS.2002.1029823

Gay, S.J.: Bounded polymorphism in session types. Math. Struct. Comput. Sci. **18**(5), 895–930 (2008). http://dx.doi.org/10.1017/S0960129508006944

Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999)

Gay, S.J., Hole, M.J.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2/3), 191–225 (2005)

Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 273–284. ACM (2008)

Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)

Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K. (ed.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003). http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf

Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999)

Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)

Mostrous, D., Yoshida, N.: Two session typing systems for higher-order mobile processes. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 321–335. Springer, Heidelberg (2007). http://dx.doi.org/10.1007/978-3-540-73228-0_23

Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order π-calculus. Inf. Comput. **241**, 227–263 (2015). http://dx.doi.org/10.1016/j.ic.2015.02.002

Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Math. Struct. Comput. Sci. **6**(5), 409–454 (1996)

Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)

Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)

Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)

Proof-Relevant Parametricity

Neil Ghani, Fredrik Nordvall Forsberg, and Federico Orsanigo(✉)

University of Strathclyde, Glasgow, Scotland
{neil.ghani,fredrik.nordvall-forsberg,federico.orsanigo}@strath.ac.uk

Abstract. Parametricity is one of the foundational principles which underpin our understanding of modern programming languages. Roughly speaking, parametricity expresses the hidden invariants that programs satisfy by formalising the intuition that programs map related inputs to related outputs. Traditionally parametricity is formulated with proof-irrelevant relations but programming in Type Theory requires an extension to proof-relevant relations. But then one might ask: can our proofs that polymorphic functions are parametric be parametric themselves? This paper shows how this can be done and, excitingly, our answer requires a trip into the world of higher dimensional parametricity.

1 Introduction

According to Strachey (2000), a polymorphic program is *parametric* if it applies the same uniform algorithm at all instantiations of its type parameters. Reynolds (1983) proposed *relational parametricity* as a mathematical model of parametric polymorphism. Phil Wadler, with his characteristic ability to turn deep mathematical insight into practical gains for programmers, showed how Reynolds' relational parametricity has strong consequences (Wadler 1989, 2007): it implies equivalences of different encodings of type constructors, abstraction properties for datatypes, and famously, it allows properties of programs to be derived "for free" purely from their types.

Within relational parametricity, types containing free type variables map not only sets to sets, but also relations to relations. A relation R between sets A and B is a subset $R \subseteq A \times B$. We call these proof-irrelevant relations as, given $a \in A$ and $b \in B$, the only information R conveys is whether a is related to b and not, for example, *how* a is related to b. However, the development of dependently type programming languages, constructive logics and proof assistants means such relations are insufficient in a number of settings. For example, it is often natural to consider a relation R between sets A and B to be a function $R : A \times B \to \mathsf{Set}$, where we think of $R(a, b)$ as the set of proofs that R relates a and b. Such *proof-relevant* relations are needed if one wants to work in the pure Calculus of Constructions (Coquand and Huet 1988) without assuming additional axioms (in contrast, Atkey (2012) formalised (proof-irrelevant) relational parametricity in

This work is partially supported by SICSA, and EPSRC grants EP/K023837/1 and EP/M016951/1.

Coq, an implementation of the Calculus of Constructions, by assuming the axiom of Propositional Extensionality). This paper asks the fundamental question:

> *Does the relational model of parametric polymorphism extend from proof-irrelevant relations to proof-relevant relations?*

At first sight, one might hope for a straightforward answer. Many properties in the proof-irrelevant world have clear analogues as proof-relevant constructions. Indeed, as we shall see, this approach gives a satisfactory treatment of the function space. However, universally quantified types pose a much more significant challenge; it is insufficient to simply take the uniformity condition inherent within a proof-irrelevant parametric polymorphic function and replace it with a function acting on proofs; this causes the Identity Extension Lemma to fail. Instead, to prove this lemma in a proof-relevant setting, we need to strengthen the uniformity condition on parametric polymorphic functions by requiring it to itself be parametric.

Proof-relevant parametricity thus entails adding a second layer of parametricity to ensure that the proofs that functions are parametric are themselves parametric. This takes us into the world of 2-dimensional parametricity where type constructors now act upon sets, relations and 2-relations. But, there are actually surprisingly many choices as to what a 2-relation is! Further, at higher dimensions, there are a number of potential equality relations and it is not *a priori* clear which of these need to be preserved and which do not. Relations are naturally organised in a cubical or simplicial manner, and so this will not surprise those familiar with simplicial and cubical methods, where there is an analogous choice of which face maps and degeneracies to consider. For example, do connections (Brown et al. 2011) have a role to play in proof-relevant parametricity? These questions are not at all obvious — we went down many false routes before finding the right answer.

The paper is structured as follows: in Sect. 2, we introduce the preliminaries we need, while Sects. 3 and 4 introduce proof-relevant relations and 2-relations. Section 5 constructs a 2-dimensional model of System F, and proves it correct by establishing 2-dimensional analogues of the Identity Extension Lemma and the Abstraction Theorem. We present a proof-of-concept application in Sect. 6, where we generalise the usual proof that parametricity implies naturality to the 2-dimensional setting. Section 7 concludes, with plans for future work including higher-dimensional logical relations, and the relationship with the cubical sets model of HoTT.

2 Impredicative Type Theory and the Identity Type

In order to make proof-relevant relations precise, we work in the constructive framework of impredicative Martin-Löf Type Theory (Coquand and Huet 1988, Martin-Löf 1972). Impredicativity allows us to quantify over all types of sort Type in order to construct a new object of sort Type.[1] Following Atkey (2012), we will

[1] In Coq, this feature can be turned on by means of the command line option `-impredicative-set`.

use impredicative quantification in the meta-theory to interpret impredicative quantification in the object theory. This simplifies the presentation, and allows us to focus on the proof-relevant aspects of the logical relations.

Apart from impredicativity, the type theory we employ is standard; we make use of dependent function types $(\Pi x : A)B(x)$ and dependent pair types $(\Sigma x : A)B(x)$ with the usual introduction and elimination rules. We write $A \to B$ for $(\Pi x : A)B$ and $A \times B$ for $(\Sigma x : A)B$ when B does not depend on $x : A$. Crucial for our development will be Martin-Löf's *identity type*, given by the following rules:

$$\frac{A : \mathsf{Type} \qquad a, b : A}{\mathsf{Id}_A(a,b) : \mathsf{Type}} \qquad \frac{a : A}{\mathsf{refl}(a) : \mathsf{Id}_A(a,a)}$$

$$\frac{C : (\Pi x, y : A)(\mathsf{Id}_A(x,y) \to \mathsf{Type}) \qquad d : (\Pi x : A)C(x,x,\mathsf{refl}(x))}{J(C,d) : (\Pi x, y : A)(\Pi p : \mathsf{Id}_A(x,y))C(x,y,p)}$$

In the language of the HoTT book (The Univalent Foundations Program, 2013), the elimination rule J is called *path induction*. We stress that we are *not* assuming Uniqueness of Identity Proofs, as that would in effect result in proof-irrelevance once again. In this paper, we will however restrict attention to types where identity proofs of identity proofs are unique, i.e. to types A where $\mathsf{Id}_{\mathsf{Id}_A(x,y)}(p,q)$ is trivial. Garner (2009) has investigated the semantics of Type Theory where all types are of this form. For our purposes, it is enough to work with a *subuniverse* of such types. To make this precise, define

$$\begin{aligned}
\mathsf{isProp}(A) &:= (\Pi x, y : A)\mathsf{Id}_A(x,y) & \mathsf{Prop} &:= (\Sigma X : \mathsf{Type})(\mathsf{isProp}(X))\\
\mathsf{isSet}(A) &:= (\Pi x, y : A)\mathsf{isProp}(\mathsf{Id}_A(x,y)) & \mathsf{Set} &:= (\Sigma X : \mathsf{Type})(\mathsf{isSet}(X))\\
\mathsf{is\text{-}1\text{-}Type}(A) &:= (\Pi x, y : A)\mathsf{isSet}(\mathsf{Id}_A(x,y)) & \mathsf{1\text{-}Type} &:= (\Sigma X : \mathsf{Type})(\mathsf{is\text{-}1\text{-}Type}(X))
\end{aligned}$$

Here Prop is the subuniverse of *propositions*, i.e. types with at most one inhabitant up to identity, while Set is the subuniverse of *sets*, i.e. types whose identity types in turn are propositional. Finally, we are interested in the subuniverse $\mathsf{1\text{-}Type}$ of 1-types, i.e. types whose identity types are sets. Subuniverses of an impredicative universe are also impredicative. Furthermore, all three of Prop, Set and $\mathsf{1\text{-}Type}$ are closed under Π- and Σ-types. The witness that a type is in a subuniverse is itself a proposition, and so we will abuse notation and leave it implicit — if there is a proof, it is unique up to identity.

We denote by $a \equiv_A b$ the existence of a proof $p : \mathsf{Id}_A(a,b)$. We often leave out the subscript if it can be inferred from context. A function $f : A \to B$ is said to be an *equivalence* if it has a left and a right inverse, and if there exists an equivalence $A \to B$, we write $A \cong B$. If $P : A \to \mathsf{Prop}$, then we write $\{x : A \mid P(x)\}$ for $(\Sigma x : A)P(x)$. Since $P(x)$ is a proposition for each $x : A$, we have that $\mathsf{Id}_{\{x:A \mid P(x)\}}((a,p),(b,q)) \cong \mathsf{Id}_A(a,b)$. For this reason, we will often leave the proof $p : P(a)$ implicit when talking about an element (a,p) of $\{x : A \mid P(x)\}$. We also suggestively write $a \in P$ for $P(a)$. The identity type has a rich structure. In order to introduce notation, we list some basic facts here, and refer to the HoTT book (The Univalent Foundations Program, 2013) for more information.

Lemma 1 (Structure on $\mathsf{Id}_A(a,b)$).

(i) *For any $p : \mathsf{Id}_A(a,b)$ there is $p^{-1} : \mathsf{Id}_A(b,a)$.*
(ii) *For any $p : \mathsf{Id}_A(a,b)$ and $q : \mathsf{Id}_A(b,c)$ there is $p \cdot q : \mathsf{Id}_A(a,c)$, and $\mathsf{refl}(a)$, $-^{-1}$ and $- \cdot -$ satisfy the laws of a (higher) groupoid.*
(iii) *All functions $f : A \to B$ are functorial in Id_A, i.e. there is a term $\mathsf{ap}(f) : \mathsf{Id}_A(a,b) \to \mathsf{Id}_B(f(a), f(b))$.*
(iv) *All type families respect Id_A, i.e. there is a function*

$$\mathsf{tr} : (P : A \to \mathsf{Type}) \to \mathsf{Id}_A(a,b) \to P(a) \to P(b). \qquad \square$$

We frequently use the following characterisation of equality in Σ-types:

Lemma 2. $\mathsf{Id}_{(\Sigma x:A)B(x)}((x,y),(x',y')) \cong (\Sigma p : \mathsf{Id}_A(x,x'))\mathsf{Id}_{B(x')}(\mathsf{tr}(B,p,y),y')$.

For function types, the corresponding statement is not provable, so we rely on the following axiom:

Axiom 3 (Function Extensionality). *The function*

$$\mathsf{happly} : \mathsf{Id}_{(\Pi x:A)B(x)}(f,g) \to (\Pi x : A)\mathsf{Id}_{B(x)}(f(x),g(x))$$

defined using J in the obvious way is an equivalence. In particular, we have an inverse

$$\mathsf{ext} : (\Pi x : A)\mathsf{Id}_{B(x)}(f(x),g(x)) \to \mathsf{Id}_{(\Pi x:A)B(x)}(f,g).$$

This axiom is justified by models of impredicative Type Theory in intuitionistic set theory. It also follows from Voevodsky's Univalence Axiom (Voevodsky 2010), which we do not assume in this paper. We will use function extensionality in order to derive the Identity Extension Lemma for arrow types, as in e.g. Wadler (2007).

3 Proof-Relevant Relations

We now define proof-relevant relations:

Definition 4. *The collection of proof-relevant relations is denoted* Rel *and consists of triples (A,B,R), where A, B : 1-*Type *and $R : A \times B \to \mathsf{Set}$. The 1–type of morphisms from (A,B,R) to (A',B',R') is*

$$(\Sigma f : A \to A')(\Sigma g : B \to B')(\Pi x : A, y : B)R(a,b) \to R'(fa,gb).$$

In the rest of this paper we take relation to mean proof-relevant relation. The above definition means morphisms between relations have a proof-relevant equality and, thus, showing morphisms are equal involves constructing explicit proofs to that effect. Indeed, the equality of morphisms is given by

$$\mathsf{Id}((f,g,p),(f',g',p')) \cong (\Sigma\phi : \mathsf{Id}(f,f'), \psi : \mathsf{Id}(g,g'))\ \mathsf{Id}(\mathsf{tr}(\phi,\psi)p, p')$$

However, since $R : A \times B \to \mathsf{Set}$ has codomain Set, while A and B are 1–types, the complexity of R compared to A and B has decreased. This means relations between proof-relevant relations are in fact proof-irrelevant (see Sect. 4).

Given a relation (A, B, R), we often denote A by R_0 and B by R_1, write $R : \mathsf{Rel}(R_0, R_1)$, or $R : R_0 \leftrightarrow R_1$, and call R a relation between A and B. Similarly, given a morphism (f, g, p), we denote f by p_0, g by p_1 and write $p : (R_0 \to R_1)(p_0, p_1)$. If $R : \mathsf{Rel}(R_0, R_1)$ and $P : \mathsf{Rel}(P_0, P_1)$, then we have $\mathbf{1} : \mathsf{Rel}(\mathbf{1}, \mathbf{1})$, $P \times R : \mathsf{Rel}(P_0 \times R_0, P_1 \times R_1)$ and $R \Rightarrow P : \mathsf{Rel}(R_0 \to P_0, R_1 \to P_1)$ defined by

$$\begin{aligned} \mathbf{1}(x,y) &:= \mathbf{1} \\ (R \times P)((x,y),(x',y')) &:= R(x,x') \times P(y,y') \\ (R \Rightarrow P)(f,g) &:= (\Pi x : R_0, y : R_1)(R(x,y) \to P(fx, gy)) \end{aligned}$$

Interpreting abstraction and application requires the following functions:

Lemma 5. *Let $R : \mathsf{Rel}(A, B)$, $R' : \mathsf{Rel}(A', B')$, and $R'' : \mathsf{Rel}(A'', B'')$. There is an equivalence $\mathsf{abs} : (R \times R' \to R'') \to (R \to (R' \Rightarrow R''))$ with inverse $\mathsf{app} : (R \to (R' \Rightarrow R'')) \to (R \times R' \to R'')$.* □

We will also make use of the *equality relation* $\mathsf{Eq}(A)$ for each 1-type A:

Definition 6. *Equality $\mathsf{Eq} : 1\text{-}\mathsf{Type} \to \mathsf{Rel}$ is defined by $\mathsf{Eq}(A) = (A, A, \mathsf{Id}_A)$ on objects and $\mathsf{Eq}(f) = (f, f, \mathsf{ap}(f))$ on morphisms.*

Proposition 7. *Eq is full and faithful in that $(\mathsf{Eq}X \to \mathsf{Eq}Y) \cong X \to Y$.*

Proof. By function extensionality and contractability of singletons, we have

$$\begin{aligned} (\mathsf{Eq}X \to \mathsf{Eq}Y) &= (\Sigma f : X \to Y)(\Sigma g : X \to Y)(\Pi x x')\mathsf{Id}_X(x,x') \to \mathsf{Id}_Y(fx, gx') \\ &\cong (\Sigma f : X \to Y)(\Sigma g : X \to Y)\mathsf{Id}_{X \to Y}(f,g) \\ &\cong (\Sigma f : X \to Y)\mathbf{1} \cong X \to Y. \end{aligned}$$ □

Similarly, the exponential of equality relations is an equality relation. Here, we abuse notation and use the same symbol for equivalence of types and isomorphisms of relations:

Proposition 8. *For all $X, Y : 1\text{-}\mathsf{Type}$, we have $(\mathsf{Eq}X \Rightarrow \mathsf{Eq}Y) \cong \mathsf{Eq}(X \to Y)$.*

Proof. By extensionality it is enough to show

$$((\Pi x, x' : X)\mathsf{Id}(x,x') \to \mathsf{Id}(fx, gx')) \cong (\Pi x : X)\mathsf{Id}(fx, gx)$$

for every $f, g : X \to Y$. Functions can easily be constructed in both directions and proved inverse using extensionality and path induction. □

4 Relations Between Relations

Intuitively, 2-relations should relate proofs of relatedness in proof-relevant relations. Although conceptually simple, formalising 2-relations is non-trivial as various choices arise. For instance, if R and R' are proof-relevant relations, one may consider 2-relations between them as being given by functions

$$Q : (\Pi a : R_0, a' : R'_0, b : R_1, b' : R'_1)\ (R(a,b) \times R'(a',b')) \to \mathsf{Prop}$$

with the intuition of $(p, p') \in Q(a, a', b, b')$ being that Q relates the proof p to the proof p'. However, the natural arrow type of such 2-relations does not preserve equality. The problem is that, while a is related to b, and a' is related to b', there is no relationship between a and a' and b and b'. Thus, we were led to the following definition which seems to originate with Grandis (see e.g. Grandis (2009)).

Definition 9. *A 2-relation consists of the following 1-types and proof-relevant relations between them*

$$\begin{tikzcd}
Q_{00} \arrow[r, leftrightarrow, "Q_{r0}"] \arrow[d, leftrightarrow, "Q_{0r}"'] & Q_{10} \arrow[d, leftrightarrow, "Q_{1r}"] \\
Q_{01} \arrow[r, leftrightarrow, "Q_{r1}"'] & Q_{11}
\end{tikzcd}$$

together with a predicate

$$Q : (\Pi a : Q_{00}, b : Q_{10}, c : Q_{01}, d : Q_{11})$$
$$Q_{r0}(a,b) \times Q_{0r}(a,c) \times Q_{r1}(c,d) \times Q_{1r}(b,d) \to \mathsf{Prop}$$

A morphism of 2-relations consists of 4 functions between each corresponding node, 4 maps of relations such that each is over the appropriate pair of morphisms of 1-types, and a predicate stating that proofs related in one 2-relation are mapped to proofs which are related in the other 2-relation.

Thus a 2-relation is a 9-tuple and, even worse, a morphism of 2-relations is a 27-tuple! This combinatorial complexity is enough to scupper any noble mathematical intentions. We therefore develop a more abstract treatment beginning with the indices in a 2-relation. This extends the notion of reflexive graphs (Robinson and Rosolini 1994; O'Hearn and Tennent 1995; Dunphy and Reddy 2004) to a second level of 2-relations; this notion, in turn, is just the first few levels of the notion of a cubical set (Brown and Higgins 1981).

Definition 10. *Let I_0 be the type with elements $\{00, 01, 10, 11\}$ of indices for 1-types, and I_1 the type with elements $\{0r, r0, 1r, r1\}$ of indices for proof-relevant relations. Define the source and target function $@ : I_1 \times \mathsf{Bool} \to I_0$ where $w@i$ replaces the occurrence of r in w by i. We write $w@i$ as wi.*

I_0**-types:** Next we develop algebra for the types contained in 2-relations.

Definition 11. *An I_0-type is a function $X \colon I_0 \to 1\text{-}\mathsf{Type}$. To increase legibility we write X_w for Xw. The collection of maps between two I_0-types is defined by*

$$X \to X' := (\Pi w : I_0) X_w \to X'_w$$

We define the following operations on I_0-types:

$$\begin{aligned} \mathbf{1} &:= \lambda w.\mathbf{1} \\ X \times X' &:= \lambda w.X_w \times X'_w \\ X \Rightarrow X' &:= \lambda w.X_w \to X'_w \end{aligned}$$

If X is an I_0-type, define its elements $\mathsf{El}X = (\Pi w : I_0)X_w$. The natural extension of this action to morphisms $f : X \to X'$ is denoted $\mathsf{El}\, f : \mathsf{El}X \to \mathsf{El}X'$.

Note that elements deserve that name as $\mathsf{El}X \cong \mathbf{1} \to X$. The construction of elements preserves structure as the following lemma shows:

Lemma 12. *Let X and X' be I_0-types. Then*

$$\begin{aligned} \mathsf{El}\,\mathbf{1} &\cong \mathbf{1} \\ \mathsf{El}(X \times X') &\cong \mathsf{El}X \times \mathsf{El}X' \\ \mathsf{El}(X \Rightarrow X') &= (\Pi w : I_0)X_w \to X'_w. \end{aligned} \qquad \square$$

Finally, we show how to interpret abstraction and application over I_0-types:

Lemma 13. *Let X, X' and X'' be I_0-types. The function*

$$\mathsf{abs} = \lambda w.\, \lambda f.\, \lambda x.\, \lambda x'.\, f(x, x') : (X \times X' \to X'') \to (X \to (X' \Rightarrow X''))$$

is an equivalence with inverse $\mathsf{app} = \lambda w.\, \lambda f.\, \lambda y.\, f\,(\pi_0 y)\,(\pi_1 y)$. $\square$

I_1**-Relations:** Next we develop algebra for the relations contained in 2-relations.

Definition 14. *An I_1-relation is a pair (X, R) of an I_0-type X and a function $R : (\Pi w : I_1)\mathsf{Rel}(X_{w0}, X_{w1})$. The collection of maps between two I_1-relations is defined by*

$$(X, R) \to (X', R') := (\Sigma f : X \to X')(\Pi w : I_1)(R_w \to R'_w)(f_{w0}, f_{w1})$$

We define the following operations on I_1-relations:

$$\begin{aligned} \mathbf{1} &:= (\mathbf{1}, \lambda w.\mathbf{1}) \\ (X, R) \times (X', R') &:= (X \times X', \lambda w.R_w \times R'_w) \\ (X, R) \Rightarrow (X', R') &:= (X \Rightarrow X', \lambda w.R_w \Rightarrow R'_w) \end{aligned}$$

If (X, R) is an I_1-relation, define its elements

$$\mathsf{El}(X, R) = (\Sigma x : \mathsf{El}X)(\Pi w : I_1)R_w(x_{w0}, x_{w1})$$

The natural extension of El to morphisms $(f, g) : (X, R) \to (X', R')$ is denoted $\mathsf{El}(f, g) : \mathsf{El}(X, R) \to \mathsf{El}(X', R')$.

Note that elements deserve that name as $\mathsf{El}(X,R) \cong \mathbf{1} \to (X,R)$. The construction of elements preserves structure as the following lemma shows:

Lemma 15. *Let (X,R) and (X',R') be I_1-relations. Then*

$$\begin{aligned} \mathsf{El}\,\mathbf{1} &\cong \mathbf{1} \\ \mathsf{El}((X,R)\times(X',R')) &\cong \mathsf{El}(X,R)\times\mathsf{El}(X',R') \\ \mathsf{El}((X,R)\Rightarrow(X',R')) &= (\Sigma f : \mathsf{El}(X\Rightarrow X'))(\Pi w : I_1)(R_w \Rightarrow R'_w)(f_{w0}, f_{w1}). \square \end{aligned}$$

Finally, we show how to interpret abstraction and application over I_0-types:

Lemma 16. *Let $(X,R), (X',R')$ and (X'',R'') be I_1-relations. There is an equivalence* $\mathsf{abs} : ((X,R)\times(X',R') \to (X'',R'')) \to ((X,R) \to ((X',R') \Rightarrow (X'',R'')))$ *with inverse*

$$\mathsf{app} : ((X,R) \to ((X',R') \Rightarrow (X'',R''))) \to ((X,R)\times(X',R') \to (X'',R'')).$$

Proof. The proof is similar to the proof of Lemma 5, but rests crucially on the fact that $R \Rightarrow P : \mathsf{Rel}(R_0 \to P_0, R_1 \to P_1)$. □

2-Relations: Finally, we develop the same algebra for 2-relations.

Definition 17. *An 2-relation is a pair consisting of an I_1-relation (X,R) and a function $Q : \mathsf{El}(X,R) \to \mathsf{Prop}$. The collection of maps between two 2-relations is defined by*

$$\begin{aligned} ((X,R),Q) \to ((X',R'),Q') := (\Sigma(f,g) : (X,R) \to (X',R')) \\ (\Pi(x,p) : \mathsf{El}(X,R))p \in Q(x) \Rightarrow (\mathsf{El}\ g\ p) \in Q'(\mathsf{El}\ f\ x) \end{aligned}$$

We define the following operations on 2-relations

$$\begin{aligned} \mathbf{1} &= (\mathbf{1}, \lambda\_.\mathbf{1}) \\ ((X,R),Q)\times((X',R)',Q') &= ((X,R)\times(X',R'), \\ &\qquad \lambda(x,y)\lambda(p,q).p\in Q(x)\wedge q\in Q'(y)) \\ ((X,R),Q)\Rightarrow((X',R'),Q') &= ((X,R)\Rightarrow(X',R'), \\ &\qquad \lambda(f,g).(\Pi(x,p):\mathsf{El}(X,R))p\in Q(x)\Rightarrow(\mathsf{El}\ g\ p)\in Q'(\mathsf{El}\ f\ x)). \end{aligned}$$

Lemma 18. *Let $((X,R),Q), ((X',R'),Q')$ and $((X'',R''),Q'')$ be 2-relations. There is an equivalence*

$$\begin{aligned} \mathsf{abs} : (((X,R),Q)\times((X',R'),Q') \to ((X'',R''),Q'')) \cong \\ (((X,R),Q) \to (((X',R'),Q') \Rightarrow ((X'',R''),Q''))) \end{aligned}$$

with inverse app.

Proof. Note that if X, X' and X'' are I_0-types, and if $f : X \times X' \to X''$ and $\mathsf{abs}f : X \to (X' \Rightarrow X'')$, then for any $x : \mathsf{El}X, x' : \mathsf{El}X'$, and $w : I_0$

$$(\mathsf{El}f)\ (x, x')\ w \equiv (\mathsf{El}\ (\mathsf{abs}f)\ x\ w)(x'\ w)$$

Similar results hold for app and for the analogous lemmas for I_1-sets. This, together with Lemma 16, extensionality and direct calculation gives the result. □

As in cubical and simplicial settings, there is more than one "degenerate" relation in higher dimensional relations. For example, we can duplicate a relation vertically or horizontally giving two functors $\mathsf{Eq}_{||}, \mathsf{Eq}_{=} \colon \mathsf{Rel} \to \mathsf{2Rel}$ sending a relation R to the 2-relation indexed, respectively, by

$$\begin{tikzcd} R_0 \arrow[r, leftrightarrow, "\mathsf{Eq}(R_0)"] \arrow[d, leftrightarrow, "R"'] \arrow[dr, phantom, "\mathsf{Eq}_{||}(R)"] & R_0 \arrow[d, leftrightarrow, "R"] \\ R_1 \arrow[r, leftrightarrow, "\mathsf{Eq}(R_1)"'] & R_1 \end{tikzcd} \qquad \begin{tikzcd} R_0 \arrow[r, leftrightarrow, "R"] \arrow[d, leftrightarrow, "\mathsf{Eq}(R_0)"'] \arrow[dr, phantom, "\mathsf{Eq}_{=}(R)"] & R_1 \arrow[d, leftrightarrow, "\mathsf{Eq}(R_0)"] \\ R_0 \arrow[r, leftrightarrow, "R"'] & R_1 \end{tikzcd}$$

where $(p, q, p', q') \in \mathsf{Eq}_{||}(R)(a, b, c, d)$ if and only if $\mathsf{tr}(p, p')q \equiv_{R(b,d)} q'$, while $(p, q, p', q') \in \mathsf{Eq}_{=}(R)(a, b, c, d)$ if and only if $\mathsf{tr}(q, q')p \equiv_{R(c,d)} p'$. Note that both the compositions $\mathsf{Eq}_{||} \circ \mathsf{Eq}$ and $\mathsf{Eq}_{=} \circ \mathsf{Eq}$ define the same functor which we denote Eq_2. Another degeneracy, called a connection (Brown et al. 2011), is defined by a functor $\mathbf{C} \colon \mathsf{Rel} \to \mathsf{2Rel}$ which maps a relation R to the 2-relation indexed by

$$\begin{tikzcd} R_0 \arrow[r, leftrightarrow, "\mathsf{Eq}(R_0)"] \arrow[d, leftrightarrow, "\mathsf{Eq}(R_0)"'] \arrow[dr, phantom, "\mathbf{C}R"] & R_0 \arrow[d, leftrightarrow, "R"] \\ R_0 \arrow[r, leftrightarrow, "R"'] & R_1 \end{tikzcd}$$

and with $(p, q, p', q') \in \mathbf{C}(R)(a, b, c, d)$ if and only if $\mathsf{tr}(q^{-1} \cdot p)p' \equiv_{R(b,d)} q'$ (there is of course also a symmetric version which swaps the role of $\mathsf{Eq}(R_0)$ and R, but we will not make us of this in the current paper). Again $\mathbf{C} \circ \mathsf{Eq}$ gives Eq_2.

Proposition 19. *The functor* $\mathsf{Eq}_{||}$ *is full and faithful.*

Proof. Similar to the proof of Proposition 7. □

Again, we can prove that exponentiation preserves all the degeneracies and the connection:

Proposition 20. *For all* $R, R' : \mathsf{Rel}$*, we have*

(i) an equivalence $\mathsf{Eq}_{||}R \Rightarrow \mathsf{Eq}_{||}R' \cong \mathsf{Eq}_{||}(R \to R')$
(ii) an equivalence $\mathsf{Eq}_{=}R \Rightarrow \mathsf{Eq}_{=}R' \cong \mathsf{Eq}_{=}(R \to R')$
(iii) an equivalence $\mathbf{C}R \Rightarrow \mathbf{C}R' \cong \mathbf{C}(R \to R')$. □

5 Proof-Relevant Two-Dimensional Parametricity

We now have the structure needed to define a 2-dimensional, proof-relevant model of System F. We recall the rules of System F in Fig. 1. Each type judgement $\Gamma \vdash T$ type, with $|\Gamma| = n$, will be interpreted in the semantics as

$$\begin{aligned}
[\![T]\!]_0 &: |\mathsf{1\text{-}Type}|^n \to \mathsf{1\text{-}Type}\\
[\![T]\!]_1 &: |\mathsf{Rel}|^n \to \mathsf{Rel}\\
[\![T]\!]_2 &: |\mathsf{2Rel}|^n \to \mathsf{2Rel}
\end{aligned}$$

by induction on type judgements with $[\![T]\!]_1$ over $[\![T]\!]_0 \times [\![T]\!]_0$, and $[\![T]\!]_2$ over $[\![T]\!]_1 \times [\![T]\!]_1 \times [\![T]\!]_1 \times [\![T]\!]_1$. This is similar to our previous work on bifibrational functorial models of (proof-irrelevant) parametricity (Ghani et al. 2015a, 2015b, but with an additional 2-relational level.

Type formation rules:

$$\frac{}{\Gamma \vdash X \text{ type}}\ (X \in \Gamma) \qquad \frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}} \qquad \frac{\Gamma, X \vdash A \text{ type}}{\Gamma \vdash \forall X.A \text{ type}}$$

Term typing rules:

$$\frac{}{\Gamma; \Delta \vdash x : A}\ (x : A \in \Delta) \qquad \frac{\Gamma; \Delta, x : A \vdash t : B}{\Gamma; \Delta \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma; \Delta \vdash s : A \to B \qquad \Gamma; \Delta \vdash t : A}{\Gamma; \Delta \vdash s\, t : B}$$

$$\frac{\Gamma, X; \Delta \vdash t : A}{\Gamma; \Delta \vdash \Lambda X.t : \forall X.A}\ (X \notin FV(\Delta)) \qquad \frac{\Gamma; \Delta \vdash t : \forall X.A \qquad \Gamma; \Delta \vdash B \text{ type}}{\Gamma; \Delta \vdash t[B] : A[X \mapsto B]}$$

Judgemental equality:

$$\frac{}{\Gamma; \Delta \vdash (\lambda x.\, t)u = t[x \mapsto u] : B} \qquad \frac{}{\Gamma; \Delta \vdash t = \lambda x.\, tx : A \to B}\ (x : A \notin \Delta)$$

$$\frac{}{\Gamma; \Delta \vdash (\Lambda X.t)[B] = t[X \mapsto B] : A[X \mapsto B]} \qquad \frac{}{\Gamma; \Delta \vdash t = \Lambda X.t[X] : \forall X.A}\ (X \notin \Gamma)$$

Fig. 1. Typing rules for System F

5.1 Interpretation of Types

The full interpretation can be found in Fig. 2. For type variables and arrow types, we just use projections and exponentials at each level. Elements of $[\![\forall X.T]\!]_0 \bar{A}$ consist of an ad-hoc polymorphic function f_0, a proof f_1 that f_0 is suitably uniform, and finally a (unique) proof (A0) that also the proof f_1 is parametric.

Similarly, elements of $(\llbracket\forall X.T\rrbracket_1\bar{R})(f,g)$ are proofs ϕ that are suitably parametric in relation to f and g, both with respect to equalities (conditions (A1.1) and (A1.2)) and connections (condition (A1.3)). We have not included uniformity also with respect to the "symmetric" connection since it is not needed for our applications, and we wish to keep the logical relation minimal.

Using Lemma 2 and function extensionality, we can characterise equality in the interpretation of $\forall$-types in the following way (note that $\mathsf{Id}_{(\llbracket\forall X.T\rrbracket_2\bar{Q})\vec{f}}(\vec{\phi},\vec{\psi})$ is trivial by assumption, since $(\llbracket\forall X.T\rrbracket_2\bar{Q})\vec{f}$ is a proposition):

Lemma 21. *For all $f,g:\llbracket\forall X.T\rrbracket_0\bar{A}$,*

$$\begin{aligned}\mathsf{Id}_{\llbracket\forall X.T\rrbracket_0\bar{A}}(f,g)\cong\{\,\tau:(\Pi A:1\text{-}\mathsf{Type})\mathsf{Id}_{\llbracket T\rrbracket_0(\bar{A},A)}(f_0A,g_0A)\mid\\ (\forall R:\mathsf{Rel})\quad(f_1R,\tau R_0,g_1R,\tau R_1)\in\\ \mathsf{Eq}_=(\llbracket T\rrbracket_1(\mathsf{Eq}(\bar{A}),R))(f_0R_0,f_0R_1,g_0R_0,g_0R_1)\}\end{aligned}$$

$$\llbracket X_0,\ldots,X_n\vdash X_k\ \mathsf{type}\rrbracket_i\vec{Y}=Y_k\qquad\llbracket S\to T\rrbracket_i\vec{Y}=\llbracket S\rrbracket_i\vec{Y}\Rightarrow_i\llbracket T\rrbracket_i\vec{Y}$$

$$\begin{aligned}\llbracket\forall X.T\rrbracket_0\bar{A}=\{\,&f_0:(\Pi A:1\text{-}\mathsf{Type})\llbracket T\rrbracket_0(\bar{A},A),\\ &f_1:(\Pi R:\mathsf{Rel})\llbracket T\rrbracket_1(\mathbf{Eq}(\bar{A}),R)(f_0R_0,f_0R_1)\mid\\ &(\forall Q:\mathsf{2Rel})\ (f_1Q_{r0},f_1Q_{0r},f_1Q_{r1},f_1Q_{1r})\in\\ &\llbracket T\rrbracket_2(\mathsf{Eq}_2(\bar{A}),Q)(f_0Q_{00},f_0Q_{10},f_0Q_{01},f_0Q_{11})\}\end{aligned}\qquad\text{(A0)}$$

$$\begin{aligned}(\llbracket\forall X.T\rrbracket_1\bar{R})((f_0,f_1),(g_0,g_1))=\{\,&\phi:(\Pi R:\mathsf{Rel})\llbracket T\rrbracket_1(\bar{R},R)(f_0R_0,g_0R_1)\mid\\ &(\forall Q:\mathsf{2Rel})\\ &\big((f_1Q_{r0},\phi Q_{0r},g_1Q_{r1},\phi Q_{1r})\in\\ &\quad\llbracket T\rrbracket_2(\mathsf{Eq}_{\|}(\bar{R}),Q)(f_0Q_{00},f_0Q_{10},g_0Q_{01},g_0Q_{11})\qquad\text{(A1.1)}\\ &\wedge(\phi Q_{r0},f_1Q_{0r},\phi Q_{r1},g_1Q_{1r})\in\\ &\quad\llbracket T\rrbracket_2(\mathsf{Eq}_=(\bar{R}),Q)(f_0Q_{00},f_0Q_{10},g_0Q_{01},g_0Q_{11})\qquad\text{(A1.2)}\\ &\wedge(f_1Q_{r0},f_1Q_{0r},\phi Q_{r1},\phi Q_{1r})\in\\ &\quad\llbracket T\rrbracket_2(\mathbf{C}\bar{R},Q)(f_0Q_{00},f_0Q_{10},f_0Q_{01},g_0Q_{11})\big)\}\qquad\text{(A1.3)}\end{aligned}$$

$$\begin{aligned}(\phi_0,\phi_1,\phi_2,\phi_3)\in(\llbracket\forall X.T\rrbracket_2\bar{Q})(f,g,h,l)\ \text{iff}\\ (\forall Q:\mathsf{2Rel})\ (\phi_0Q_{r0},\phi_1Q_{0r},\phi_2Q_{r1},\phi_3Q_{1r})\in\\ \llbracket T\rrbracket_2(\bar{Q},Q)(f_0Q_{00},g_0Q_{10},h_0Q_{01},l_0Q_{11})\end{aligned}$$

Fig. 2. Interpretation of types

This can be used to prove a generalised version of the Identity Extension Lemma:

Theorem 22 (IEL). *For every type judgement $\Gamma \vdash T$ type, we have*

(i) an equivalence $\Theta_{T,0} : [\![T]\!]_1 \circ \mathsf{Eq} \cong \mathsf{Eq} \circ [\![T]\!]_0$,
(ii) an equivalence $\Theta_{T,||} : [\![T]\!]_2 \circ \mathsf{Eq}_{||} \cong \mathsf{Eq}_{||} \circ [\![T]\!]_1$ over $\Theta_{T,0}$,
(iii) an equivalence $\Theta_{T,=} : [\![T]\!]_2 \circ \mathsf{Eq}_{=} \cong \mathsf{Eq}_{=} \circ [\![T]\!]_1$ over $\Theta_{T,0}$, and
(iv) an equivalence $\Theta_{T,\mathbf{C}} : [\![T]\!]_2 \circ \mathbf{C} \cong \mathbf{C} \circ [\![T]\!]_1$ over $\Theta_{T,0}$. □

Proof. We prove (i) for ∀-types, since it is useful in order to understand the logical relations in Fig. 2. We refer to the appendix for the rest of the proof.

We define maps

$$\Theta_{\forall X.T,0} : [\![\forall X.T]\!]_1 \mathsf{Eq}(\bar{A})(f,g) \to \mathsf{Eq}([\![\forall X.T]\!]_0 \bar{A})(f,g)$$
$$\Theta^{-1}_{\forall X.T,0} : \mathsf{Eq}([\![\forall X.T]\!]_0 \bar{A})(f,g) \to [\![\forall X.T]\!]_1 \mathsf{Eq}(\bar{A})(f,g)$$

for all f, g and show that they are inverses — this does not come for free, as in the proof-irrelevant setting, but will be considerably easier since we are considering 1-types only, and not arbitrary types. We first define $\Theta_{\forall X.T,0}(\rho) := \varphi(\lambda(A : \text{1-}\mathsf{Type}).\, \rho\, \mathsf{Eq}(A))$, where φ is part of the equivalence given by Lemma 21. The condition from Lemma 21 is satisfied by (A1.1) together with the induction hypothesis.

For $\Theta^{-1}_{\forall X.T,0}$, we define $\Theta^{-1}_{\forall X.T,0}(\tau) := \lambda R\colon \mathsf{Rel}.\, \mathsf{tr}(f_1 \mathsf{Eq}(R_0), \tau R_1) f_1 R$. We need to check that conditions (A1.1), (A1.2) and (A1.3) are satisfied — we verify (A1.1) in detail here, (A1.2) and (A1.3) follow analogously. Let $Q : \mathsf{2Rel}$. By (A0), we have

$$(f_1 Q_{r0}, f_1 Q_{0r}, f_1 Q_{r1}, f_1 Q_{1r}) \in [\![T]\!]_2(\mathsf{Eq}_2(\bar{A}), Q)(f_0 Q_{00}, f_0 Q_{10}, f_0 Q_{01}, f_0 Q_{11})$$

while we want to prove

$$\begin{aligned}(f_1 Q_{r0}, \mathsf{tr}(f_1 \mathsf{Eq}(Q_{00}), \tau Q_{01}) f_1 Q_{0r}, g_1 Q_{r1}, \mathsf{tr}(f_1 \mathsf{Eq}(Q_{10}), \tau Q_{11}) f_1 Q_{1r})\\ \in [\![T]\!]_2(\mathsf{Eq}_{||}(\mathsf{Eq}(\bar{A})), Q)(f_0 Q_{00}, f_0 Q_{10}, g_0 Q_{01}, g_0 Q_{11}).\end{aligned}$$

Since $\mathsf{Eq}_{||}(\mathsf{Eq}(\bar{A})) \cong \mathsf{Eq}_2(\bar{A})$, we only need to prove

$$p : (f_0 Q_{00}, f_0 Q_{10}, f_0 Q_{01}, f_0 Q_{11}) \equiv (f_0 Q_{00}, f_0 Q_{10}, g_0 Q_{01}, g_0 Q_{11})$$

and

$$\begin{aligned}q : \mathsf{tr}(p)(f_1 Q_{r0}, f_1 Q_{0r}, f_1 Q_{r1}, f_1 Q_{1r}) \equiv\\ (f_1 Q_{r0}, \mathsf{tr}(f_1 \mathsf{Eq}(Q_{00}), \tau Q_{01}) f_1 Q_{0r}, g_1 Q_{r1}, \mathsf{tr}(f_1 \mathsf{Eq}(Q_{10}), \tau Q_{11}) f_1 Q_{1r})\end{aligned}$$

and transport along $\mathsf{pair}_{=}(p,q)$. We use $p = \mathsf{pair}_{=}(f_1 \mathsf{Eq}(Q_{00}), f_1 \mathsf{Eq}(Q_{10}), \tau Q_{01}, \tau Q_{11})$ and q given by conditions (A1.1), (A1.3), (A0) for f_1 and the condition from Lemma 21.

We now check that $\Theta_{\forall X.T,0} \circ \Theta^{-1}_{\forall X.T,0} = \mathsf{id}$ and $\Theta^{-1}_{\forall X.T,0} \circ \Theta_{\forall X.T,0} = \mathsf{id}$. One way round

$$\Theta_{\forall X.T,0}(\Theta^{-1}_{\forall X.T,0}(\tau))(A) = \mathsf{tr}(f_1 \mathsf{Eq}(A), \tau A) f_1 \mathsf{Eq}(A) \equiv (f_1 \mathsf{Eq}(A))^{-1} \bullet f_1 \mathsf{Eq}(A) \bullet \tau A \equiv \tau A$$

by Lemma 1 as required. By definition we have $\Theta^{-1}_{\forall X.T,0}(\Theta_{\forall X.T,0}(\rho))(R) = \mathsf{tr}(f_1\mathsf{Eq}(A), \rho\mathsf{Eq}(B))f_1 R$. Condition A1.3 implies that

$$(f_1\mathsf{Eq}(A), \rho R, \rho\mathsf{Eq}(B), f_1 R) \in [\![T]\!]_2(\mathbf{C}(\mathsf{Eq}(\bar{A})), \mathsf{Eq}_{||}(R))(f_0 A, f_0 A, f_0 B, f_B)$$

and since $\mathbf{C}(\mathsf{Eq}(\bar{A})) \cong \mathsf{Eq}_{||}(\mathsf{Eq}(\bar{A}))$, by the induction hypothesis $(f_1\mathsf{Eq}(A), \rho R, \rho\mathsf{Eq}(B), f_1 R)$ are related in $\mathsf{Eq}_{||}[\![T]\!]_2(\mathsf{Eq}(\bar{A}), R)$, i.e. $\mathsf{tr}(f_1\mathsf{Eq}(A), \rho\mathsf{Eq}(B))f_1 R = \rho(R)$ as required. □

The proof critically uses of the uniformity condition (A1.3) for connections. In the interpretation of ∀-types in Fig. 2, and in the proof of Theorem 22, we made some seemingly arbitrary choices: we choose to only be uniform with respect to one connection, and we used the given f_1, not the given g_1, in order to construct the isomorphism $\Theta^{-1}_{\forall X.T,0}$. The following lemma shows that these choices are actually irrelevant:

Lemma 23. *For every type judgement $\Gamma, X \vdash T$ type and $(f_0, f_1) \in [\![\forall X.T]\!]_0\vec{A}$, $\phi \in [\![\forall X.T]\!]_1(\mathsf{Eq}\vec{A})(f, g)$, we have:*

(i) For every relation R, $\mathsf{tr}(f_1\mathsf{Eq}R_0, \phi\mathsf{Eq}R_1)f_1 R = \phi R$.
(ii) For every relation R,

$$\mathsf{tr}(f_1\mathsf{Eq}R_0, \phi\mathsf{Eq}R_1)f_1 R = \mathsf{tr}((\phi\mathsf{Eq}R_0)^{-1}, (g_1\mathsf{Eq}R_1)^{-1})g_1 R.$$

(iii) For every 2-relation Q,

$$(\phi Q_{r0}, \phi Q_{0r}, g_1 Q_{r1}, g_1 Q_{1r}) \in [\![T]\!]_2(\mathbf{C} \circ \mathsf{Eq}\vec{A}, Q)(f_0 Q_{00}, g_0 Q_{10}, g_0 Q_{01}, g_0 Q_{11}).$$

□

Here, item (i) is a technical lemma, while item (ii) says that one can equally well use g_1 as f_1 in the proof of Theorem 22. Finally item (iii) shows that in certain cases, the interpretation of terms of ∀-type are uniform also with respect to the other connection which is not explicitly mentioned in the logical relation for ∀.

5.2 Interpretation of Terms

We next show how to interpret terms. A term $\Gamma; \Delta \vdash t : T$, with $|\Gamma| = n$, will be given a "standard" interpretation

$$[\![t]\!]_0\vec{A} : [\![\Delta]\!]_0\vec{A} \to [\![T]\!]_0\vec{A},$$

for every $\vec{A} : 1\text{-}\mathsf{Type}^n$, a relational interpretation

$$([\![t]\!]_0\vec{R}_0, [\![t]\!]_0\vec{R}_1, [\![t]\!]_1\vec{R}) : [\![\Delta]\!]_1\vec{R} \to [\![T]\!]_1\vec{R},$$

for every $\vec{R} : \mathsf{Rel}^n$, and finally a 2-relational interpretation

$$(([\![t]\!]_0\vec{Q}_-, [\![t]\!]_1\vec{Q}_-), [\![t]\!]_2\vec{Q}) : [\![\Delta]\!]_2\vec{Q} \to [\![T]\!]_2\vec{Q}$$

for every $\vec{Q} : \mathsf{2Rel}^n$, where we have written e.g. $[\![t]\!]_0\vec{Q}_-$ for the map of I_0-types with components $([\![t]\!]_0\vec{Q}_-)w = [\![t]\!]_0\vec{Q}_w : [\![\Delta]\!]_0\vec{Q}_w \to [\![T]\!]_0\vec{Q}_w$ and similarly for $[\![t]\!]_1\vec{Q}_-$. At each level, $\Delta = x_1 : T_1, \ldots, x_m : T_m$ is interpreted as the product

$$[\![x_1 : T_1, \ldots, x_m : T_m]\!]_i = [\![T_1]\!]_i \times \ldots \times [\![T_m]\!]_i.$$

The full interpretation is given in Fig. 3. Variables, term abstraction and term application are again given by projections and the exponential structure at each level. For type abstraction and type application, we use the same concepts at the meta-level, but we also have to prove that the resulting term satisfies the uniformity conditions (A0), (A1.1), (A1.2) and (A1.3). In addition, we have to put in a twist for the relational interpretation in order to validate the β- and η-rules.

Lemma 24. *The interpretation in Fig. 3 is well-defined.*

Proof. The interpretation of $\Gamma; \Delta \vdash \Lambda X.t : \forall X.T$ is type-correct, since Δ is weakened with respect to X in $\Gamma, X; \Delta \vdash t : T$. The uniformity conditions (A0), (A1.1), (A1.2) and (A1.3) can all be proven using $[\![t]\!]_2$. $\square$

$$[\![x_0 : T_0, \ldots, x_n : T_n \vdash x_k : T_k]\!]_i\vec{X} \coloneqq \pi_k \qquad [\![\Delta, x : S \vdash t : T]\!]_i = [\![\Delta \vdash t : T]\!]_i \circ \pi_0$$

$$\begin{gathered}
[\![\Delta \vdash \lambda x.\, t : S \to T]\!]_0\vec{A}(\gamma) = \lambda s.\, [\![\Delta, x : S \vdash t : T]\!]_0\vec{A}(\gamma, s) \\
[\![\Delta \vdash \lambda x.\, t : S \to T]\!]_1\vec{R}(\overline{\gamma}) = \lambda s_0.\, \lambda s_1.\, \lambda s.\, [\![\Delta, x : S \vdash t : T]\!]_1\vec{R}((\gamma_0, s_0), (\gamma_1, s_1), (\gamma, s)) \\
[\![\Delta \vdash \lambda x.\, t : S \to T]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}) = \lambda((x, p), \gamma).\, [\![\Delta, x : S \vdash t : T]\!]_2\vec{Q}((\overline{x}, x), (\overline{p}, p))(\overline{\gamma}, \gamma)
\end{gathered}$$

$$\begin{gathered}
[\![f\; t]\!]_0\vec{A}(\gamma) = [\![f]\!]_0\vec{A}(\gamma)\,([\![t]\!]_0\vec{A}(\gamma)) \\
[\![f\; t]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma) = [\![f]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma, [\![t]\!]_0\vec{R}_0(\gamma_0), [\![t]\!]_0\vec{R}_1(\gamma_1), [\![t]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma)) \\
[\![f\; t]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}) = [\![f]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}, [\![t]\!]_0\vec{Q}_{\overline{i}}(\overline{x}), [\![t]\!]_1\vec{Q}_{\overline{j}}(\overline{p}), [\![t]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}))
\end{gathered}$$

$$[\![\Lambda X.t]\!]_0\vec{A}(\gamma) = (\lambda A.\, [\![\Delta, X; \Delta \vdash t : T]\!]_0(\vec{A}, A)\gamma, \lambda R.\, [\![t]\!]_1(\mathsf{Eq}(\vec{A}), R)\Theta_{\Delta,0}(\mathsf{refl}(\gamma)))$$

$$\begin{gathered}
[\![\Lambda X.t]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma) = \lambda R.\, ([\![t]\!]_1(\vec{R}, R))(\gamma_0, \gamma_1, \gamma) \\
[\![\Delta \vdash \Lambda X.t : \forall X.T]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}) = \lambda Q.\, [\![t]\!]_2(\vec{Q}, Q)((\overline{x}, \overline{p}), \overline{\gamma})
\end{gathered}$$

$$\begin{gathered}
[\![\Delta \vdash t[S] : T[S \mapsto X]]\!]_0\vec{A}(\gamma) = \mathsf{fst}([\![t]\!]_0\vec{A}(\gamma))([\![S]\!]_0\vec{A}) \\
[\![t[S]]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma) = \mathsf{tr}(\Theta_{T,0}(\mathsf{snd}([\![t]\!]_0\vec{R}_0\gamma_0)\mathsf{Eq}([\![S]\!]_0\vec{R}_0)))^{-1}([\![t]\!]_1\vec{R}(\gamma_0, \gamma_1, \gamma)([\![S]\!]_1\vec{R})) \\
[\![t[S]]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma}) = [\![t]\!]_2\vec{Q}((\overline{x}, \overline{p}), \overline{\gamma})([\![S]\!]_2\vec{Q})
\end{gathered}$$

Fig. 3. Interpretation of terms

Theorem 25. *The interpretation defined in Fig. 3 is sound, i.e. if* $\Gamma; \Delta \vdash s = t : T$*, then there is* $p_{\bar{A}} : \mathsf{Id}_{[\![T]\!]_0 \bar{A}}([\![s]\!]_0, [\![t]\!]_0)$ *and* $q_{\bar{R}} : \mathsf{Id}_{[\![T]\!]_1 \bar{R}}(\mathsf{tr}(p_{\bar{R}_0})([\![s]\!]_1), [\![t]\!]_1)$*. (We automatically have* $\mathsf{tr}(p, q)[\![s]\!]_2 \equiv [\![t]\!]_2$ *by proof-irrelevance of 2-relations).* □

This model reveals hidden uniformity not only in the "standard" interpretation of terms as functions, but also in the canonical proofs of this uniformity via the Reynolds relational interpretation of terms. In more detail: consider a term $\Gamma; \Delta \vdash t : T$ with $|\Gamma| = n$. By construction, our model shows that if $\vec{R} : \mathsf{Rel}^n$, $a : [\![\Delta]\!]_0 \vec{R}_0$, $b : [\![\Delta]\!]_0 \vec{R}_1$ and $p : [\![\Delta]\!]_1 \vec{R}(a, b)$, then $[\![t]\!]_1 \vec{R} p : [\![T]\!]_1 \vec{R}([\![t]\!]_0 \vec{R}_0\ a, [\![t]\!]_0 \vec{R}_1\ b)$, i.e. $[\![t]\!]_1 \vec{R}\ p$ is a proof that $[\![t]\!]_0 \vec{R}_0\ a$ and $[\![t]\!]_0 \vec{R}_1\ b$ are related at $[\![T]\!]_1 \vec{R}$. This is a proof-relevant version of Reynolds' Abstraction Theorem. Furthermore, if $\vec{Q} : 2\mathsf{Rel}^n$, $(a, b, c, d) : [\![\Delta]\!]_0 \vec{Q}_{00} \times [\![\Delta]\!]_0 \vec{Q}_{10} \times [\![\Delta]\!]_0 \vec{Q}_{01} \times [\![\Delta]\!]_0 \vec{Q}_{11}$ and $(p, q, r, s) \in [\![\Delta]\!]_2 \vec{Q}(a, b, c, d)$, then

$$\begin{aligned}([\![t]\!]_1 \vec{Q}_{r0}\ p, [\![t]\!]_1 \vec{Q}_{0r}\ q, [\![t]\!]_1 \vec{Q}_{r1}\ r, [\![t]\!]_1 \vec{Q}_{1r}\ s) \in \\ [\![T]\!]_2 \vec{Q}([\![t]\!]_0 \vec{Q}_{00}\ a, [\![t]\!]_0 \vec{Q}_{10}\ b, [\![t]\!]_0 \vec{Q}_{10}\ c, [\![t]\!]_0 \vec{Q}_{11}\ d)\end{aligned}$$

This is the Abstraction Theorem "one level up" for the proofs $[\![t]\!]_1$, which we will put to use in the next section.

6 Theorems About Proofs for Free

In Phil Wadler's famous 'Theorems for free!' (Wadler 1989), the fact that parametric transformations are always natural in the categorical sense is shown to have many useful and fascinating consequences. Among other things, it is shown that

$$[\![A]\!] \cong [\![\forall X.(A \to X) \to X]\!]$$

for all types A — the categorically inclined reader will recognise this as an instance of the Yoneda Lemma (see e.g. Mac Lane (1998)) for the identity functor, if only we dared to consider the right hand side of the equation to consist of *natural* transformations only. And indeed, as Wadler shows (and Reynolds already knew (1983)), all System F terms $[\![t]\!] : [\![\forall X.(A \to X) \to X]\!]$ are natural by parametricity. Hence, in proof-irrelevant parametric models of System F, indeed $[\![A]\!] \cong [\![\forall X.(A \to X) \to X]\!]$.

In a more expressive theory such as (impredicative) Martin-Löf Type Theory with proof-irrelevant identity types and function extensionality, we can go further even without a relational interpretation, as pointed out by Steve Awodey (personal communication). Taking inspiration from the Yoneda Lemma once again, we can show

$$A \cong (\Sigma t : (\Pi X : \mathsf{Set})(A \to X) \to X)\, \mathsf{isNat}(t) \tag{1}$$

where

$$\mathsf{isNat}(t) := (\Pi X, Y : \mathsf{Set})(\Pi f : X \to Y)\mathsf{Id}_{(A \to X) \to Y}(f \circ t_X, t_Y \circ (f \circ \_))$$

expresses that t is a natural transformation (note that we need the identity type in order to state this). If we start with the interpretation of a System F term in a proof-irrelevant model of parametricity, we can automatically derive this naturality proof using Wadler's argument.

The above isomorphism (1) relied on A being a set, i.e. that A has no non-trivial higher structure. If we instead consider $A : \mathsf{1\text{-}Type}$, the isomorphism (1) fails; instead we have

$$A \cong (\Sigma t : (\Pi X : \mathsf{Set})(A \to X) \to X)(\Sigma p : \mathsf{isNat}(t))\, \mathsf{isCoh}(p) \tag{2}$$

where

$$\begin{aligned}\mathsf{isCoh}(p) := (\Pi X, Y, Z : \mathsf{1\text{-}Type})(\Pi f : X \to Y)(\Pi g : Y \to Z)&\\ (p\, X\, Z\, (g \circ f)) \equiv (p\, Y\, Z\, g) \star (p\, X\, Y\, f)&\end{aligned}$$

expresses that the proof p is suitably coherent. Here $(p\, Y\, Z\, g) \star (p\, X\, Y\, f)$ is the operation that pastes the two proofs $p\, X\, Y\, f$ and $p\, Y\, Z\, g$ of diagrams commuting into a proof that the composite diagram commutes. Proof-irrelevant parametricity can not ensure this coherence condition, but as we will see, an extension of the usual naturality argument to proof-relevant parametricity will guarantee this extra uniformity of the proof as well.

6.1 Graph Relations and Graph 2-Relations

Relations representing graphs of functions are key to many applications of parametricity.

Definition 26. *Let $f : A \to B$ in 1-Type. We define the* graph $\langle f \rangle$ *of f as $\langle f \rangle := (A, B, \lambda a.\, \lambda b.\, \mathsf{Id}_B(f a, b)) : \mathsf{Rel}$. This extends to an action on commuting squares: if $g : A' \to B'$, $\alpha : A \to A'$, $\beta : B \to B'$ and $p : (\Pi x : A)\, \mathsf{Id}_{B'}(g(\alpha(a)), \beta(f(a)))$, then we define $\langle \alpha, \beta \rangle = (\alpha, \beta, \lambda a.\, \lambda b.\, \lambda(r : f a \equiv b).\, p(a) \centerdot \mathsf{ap}(\beta)(r)) : \langle f \rangle \to \langle g \rangle$.*

Abstractly, we see that $\langle f \rangle$ is obtained from $\mathsf{Eq}(B)$ by "reindexing" along (f, id) and there is a morphism $\langle f, \mathsf{id} \rangle : \langle f \rangle \to \mathsf{Eq}(B)$; in particular, we recover $\mathsf{Eq}(B)$ as $\langle \mathsf{id}_B \rangle$. Just like Eq is full and faithful, so is $\langle - \rangle : \mathsf{1\text{-}Type}^{\to} \to \mathsf{Rel}$:

Lemma 27. *For all $f : A \to B$, $g : A' \to B'$,*

$$(\langle f \rangle \Rightarrow \langle g \rangle) \cong (\Sigma \alpha : A \to A')(\Sigma \beta : B \to B')\mathsf{Id}_{A \to B'}(g \circ \alpha, \beta \circ f). \qquad \square$$

The main tool for deriving consequences of parametricity is the Graph Lemma, which relates the graph of the action of a functor on a morphism with its relational action on the graph of the morphism.

Theorem 28. *Let $F_0 : 1\text{-}\mathsf{Type} \to 1\text{-}\mathsf{Type}$ and $F_1 : \mathsf{Rel} \to \mathsf{Rel}$ over F_0 be functorial. If $F_1(\mathsf{Eq}(A)) \cong \mathsf{Eq}(F_0 A)$ for all A, then for any $f : A \to B$, there are morphisms $(\mathsf{id}, \mathsf{id}, \phi_{F,f}) : \langle F_0 f \rangle \to F_1 \langle f \rangle$ and $(\mathsf{id}, \mathsf{id}, \psi_{F,f}) : F_1 \langle f \rangle \to \langle F_0 f \rangle$.* $\square$

Note that in our proof-relevant setting, this theorem does *not* construct an equivalence $\langle F_0 f\rangle \cong F_1\langle f\rangle$. Instead, we only have a logical equivalence, i.e. maps in both directions, and that seems to be enough for all known consequences of parametricity. (In a proof-irrelevant setting, the constructed logical equivalence would automatically be an equivalence.)

Next, we consider also graph 2-relations. Since we have multiple "equality 2-relations", one could expect also multiple graph 2-relations, but for the application we have in mind, one suffices. Given functions f, g, l and h, we write $\Box(f, g, l, h)$ for the 1-type of proofs that the square

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & B \arrow[d, "g"] \\
C \arrow[r, "l"'] & D
\end{tikzcd}$$

commutes, i.e. $\Box(f, g, l, h) = (\Pi x : A)\mathsf{Id}_D(g(fx), l(hx))$. We define the 1-type of commuting squares by

$$(\text{1-Type}^{\rightarrow})^{\rightarrow} := (\Sigma f : A \rightarrow B)(\Sigma g : B \rightarrow D)(\Sigma l : C \rightarrow D)(\Sigma h : A \rightarrow C)\Box(f, g, l, h)$$

A morphism $(f, g, l, h, p) \rightarrow (f', g', l', h', p')$ in $(\text{1-Type}^{\rightarrow})^{\rightarrow}$ consists of four morphisms $\alpha : A \rightarrow A'$, $\beta : B \rightarrow B'$, $\gamma : C \rightarrow C'$ and $\delta : D \rightarrow D'$, and four proofs $q : \Box(\alpha, h', \gamma, h)$, $q' : \Box(\beta, f', \delta, g)$, $r : \Box(\gamma, l', \delta, l)$ and $r' : \Box(\alpha, f', \beta, f)$ such that they form a "commuting cube"

$$\begin{tikzcd}
& B \arrow[rr, "\beta"] \arrow[dd, "g"] & & B' \arrow[dd, "g'"] \\
A \arrow[ur, "f"] \arrow[rr, "\alpha"'] \arrow[dd, "h"'] & & A' \arrow[ur, "f'"] \arrow[dd, "h'"] & \\
& D \arrow[rr, "\delta"] & & D' \\
C \arrow[ur, "l"] \arrow[rr, "\gamma"'] & & C' \arrow[ur, "l'"'] &
\end{tikzcd}$$

i.e. such that $p \star q \star r \equiv p' \star q' \star r'$, where $p \star q \star r$ and $p' \star q' \star r'$ are pastings of the squares that proves that both ways from one corner of the cube to the opposite one commutes. The 2-*graph* $\langle \_ \rangle_2 : (\text{1-Type}^{\rightarrow})^{\rightarrow} \rightarrow \mathsf{2Rel}$ is defined by

$$\langle f, g, h, l, p\rangle_2 := (\langle f\rangle, \langle g\rangle, \langle h\rangle, \langle l\rangle, \lambda(a, b, c, d).\, \lambda(q, r, s, t).\, w(a) \cdot \mathsf{ap}(g)p \cdot q \equiv \mathsf{ap}(l)s \cdot r)$$

It also has an action on morphisms, which we omit here. The 2-relation says that the two ways to prove $h(l(a)) = d$ using p, q, r, s, t are in fact equal. Again, more abstractly, this is a "reindexing" of $\mathsf{Eq}_{||}(\langle g\rangle)$ along the morphism $((\langle f, l\rangle, \langle f, \mathsf{id}\rangle, \mathsf{id}_{\langle g\rangle}), \langle l, \mathsf{id}\rangle) : (\langle h\rangle, \langle f\rangle, \langle g\rangle, \langle l\rangle) \rightarrow (\langle g\rangle, \mathsf{Eq}(B), \langle g\rangle, \mathsf{Eq}(D))$ in Rel^4.

Lemma 29. *$\langle - \rangle_2$ is full and faithful in the sense that*

$$(\langle f, g, h, l, p\rangle_2 \rightarrow_{\mathsf{2Rel}} \langle f', g', h', l', p'\rangle_2) \cong (f, g, h, l, p) \rightarrow_{(\text{1-Type}^{\rightarrow})^{\rightarrow}} (f', g', h', l', p'). \qquad \Box$$

This lemma can be used to prove a 2-relational version of the Graph Lemma.

Theorem 30 (2-relational Graph Lemma). *Let* $F_2 : \mathsf{2Rel} \to \mathsf{2Rel}$ *be functorial, and over* (F_0, F_1) *where* F_0 *and* F_1 *are as in Theorem 28. If* $F_2(\mathsf{Eq}R) \cong \mathsf{Eq}(F_1 R)$ *for all* R*, then for any* (f, g, h, l, p) *in* $(\mathit{1}\text{-}\mathsf{Type}^{\to})^{\to}$*, there are morphisms* $\phi_2 : \langle F_0 f, F_0 g, F_0 h, F_0 l, \mathsf{ap}(F_0)p\rangle_2 \to F_2\langle f, g, h, l, p\rangle_2$ *and* $\psi_2 : F_2\langle f, g, h, l, p\rangle_2 \to \langle F_0 f, F_0 g, F_0 h, F_0 l, \mathsf{ap}(F_0)p\rangle$ *in* $\mathsf{2Rel}$ *over* (ϕ, ϕ) *and* (ψ, ψ) *from Theorem 28.* □

6.2 Coherent Proofs of Naturality

Let us now apply the tools we have developed to the question of the coherence of the naturality proofs from parametricity. We first recall the standard theorem that holds also with proof-irrelevant parametricity:

Theorem 31 (Parametric Terms are Natural). *Let* $F(X)$ *and* $G(X)$ *be functorial type expressions in the free type variable* X *in some type context* Γ*. Every term* $\Gamma; - \vdash t : \forall X. F(X) \to G(X)$ *gives rise to a natural transformation* $[\![F]\!]_0 \to [\![G]\!]_0$*, i.e. if* $f : A \to B$ *then there is* $\mathsf{nat}(f) : \mathsf{Id}([\![G]\!]_0(f) \circ [\![t]\!]_0\, A, [\![t]\!]_0\, B \circ [\![F]\!]_0(f))$.

Proof. We construct $\mathsf{nat}(f)$ using the relational interpretation of t: By construction, $[\![t]\!]_1 \langle f\rangle : [\![F]\!]_1(\langle f\rangle) \to [\![G]\!]_1(\langle f\rangle)$, hence using Theorem 28,

$$\psi_{G,f} \circ [\![t]\!]_1 \langle f\rangle \circ \phi_{F,f} : (\Pi xy)\, \langle [\![F]\!]_0 f\rangle(x, y) \to \langle [\![G]\!]_0 f\rangle([\![t]\!]_0\, A x, [\![t]\!]_0\, B\, y)$$

and since $\mathsf{refl} : \langle [\![F]\!]_0 f\rangle(a, ([\![F]\!]_0 f)a)$ for each $a : [\![F]\!]_0 A$, we can define $\mathsf{nat}(f) := \mathsf{ext}(\lambda a.\, (\psi_{G,f} \circ [\![t]\!]_1 \langle f\rangle \circ \phi_{F,f})\, a\, (([\![F]\!]_0 f)a)\, \mathsf{refl})$. □

In order for $([\![t]\!]_0, \mathsf{nat})$ to lie in the image of the isomorphism (2), we also need the naturality proofs to be coherent. But thanks to the 2-relational interpretation, we can show that they are:

Theorem 32. (Naturality Proofs are Coherent). *Let* F*,* G *and* t *be as in Theorem 31. The proof* $\mathsf{nat} : \mathsf{isNat}([\![t]\!]_0)$ *is coherent, i.e. for all* $f : A \to B$ *and* $g : B \to C$*, there is a proof* $\mathsf{coh}(f, g) : \mathsf{Id}(\mathsf{nat}(g \circ f), \mathsf{nat}(g) \star \mathsf{nat}(f))$.

Proof. We construct $\mathsf{coh}(f, g)$ using the 2-relational interpretation of t. By construction, $[\![t]\!]_2\langle f, g, g \circ f, \mathsf{id}, \mathsf{refl}\rangle_2 : [\![F]\!]_2\langle f, g, g \circ f, \mathsf{id}, \mathsf{refl}\rangle_2 \to [\![G]\!]_2\langle f, g, g \circ f, \mathsf{id}, \mathsf{refl}\rangle_2$, hence using Theorem 30,

$$\begin{aligned}&\phi_2 \circ [\![t]\!]_2\langle f, g, g \circ f, \mathsf{id}, \mathsf{refl}\rangle_2 \circ \psi_2 :\\ &\qquad (\Pi(\bar{x}, \bar{r}))\big(\bar{r} \in \langle F_0 f, F_0 g, F_0(g \circ f), \mathsf{id}, \mathsf{ap}(F_0)p\rangle_2 \bar{x}\\ &\qquad\qquad \to ([\![t]\!]_1 \bar{r}) \in \langle G_0 f, G_0 g, G_0(g \circ f), \mathsf{id}, \mathsf{ap}(G_0)p\rangle_2([\![t]\!]_0 \bar{x})\big)\end{aligned}$$

We define

$$\mathsf{coh}(f, g) := \mathsf{ext}(\lambda a.\, (\phi_2 \circ [\![t]\!]_2\langle f, g, g \circ f, \mathsf{id}, \mathsf{refl}\rangle_2 \circ \psi_2)\, (a, (F_0 f)a, F_0(g \circ f)a, a)\, \vec{\mathsf{refl}})$$

— this works, since ϕ_2 and ψ_2 are over (ϕ, ϕ) and (ψ, ψ) respectively, since $\mathsf{nat}(h)$ is defined to be $(\phi \circ [\![t]\!]_1 \circ \psi)\mathsf{refl}$, and since the 2-relation $\langle G_0 f, G_0 g, G_0 (g \circ f), \mathsf{id}, \mathsf{ap}(G_0)p\rangle_2$ exactly says that pasting the two diagrams produces the third in this case. □

7 Conclusions and Future Work

In this paper, we tackled the concrete problem of transporting Reynolds' theory of relational parametricity to a proof-relevant setting. This is non-trivial as one must modify Reynolds' uniformity predicate on polymorphic functions so that it itself becomes parametric. Implementing this intuition has significant mathematical ramifications: an extra layer of 2-dimensional relations is needed to formalise the idea of two proofs being related to each other. Further, there are a variety of choices to be made as to what face maps and degeneracies to use between proof-relevant relations and 2-relations. Having made these choices, we showed that the key theorems of parametricity, namely the identity extension lemma and the fundamental theorem of logical relations hold. Finally, we explored how a standard consequence of relational parametricty — namely the fact that parametricity implies naturality — also holds in the proof-relevant setting. This work complements the more proof-theoretic work on *internal* parametricity in proof-relevant frameworks (Bernardy et al. 2015, 2012; Polonsky 2015). Relevant is also the work on parametricity for dependent types in general (Atkey et al. 2014; Krishnaswami and Dreyer 2013), assuming proof-irrelevance.

In terms of future work, we are extending the results of this paper to arbitrary dimensions. We have a candidate definition of higher-dimensional relations, the requisite face maps and degeneracies and we have proven the Identity Extension Lemma. What remains to do is to fully investigate the consequences. For instance, what form of higher dimensional initial algebra theorem can be proved with higher dimensional parametricity? More generally, we need to compare the methods, structures and results of higher dimensional parametricity with (where possible) Homotopy Type Theory and in particular its cubical sets model (Bezem et al. 2014), which shares many striking similarities. Finally, once the theoretical framework is settled, we will want to implement it and then use that implementation in formal proof.

Acknowledgements. We thank Bob Atkey, Peter Hancock and the anonymous reviewers for helpful discussions and comments.

A Proofs from Section 5

Proof (of Theorem 22). The proof is done by induction on type judgements. For type variables, all statements are trivial. For arrow types, this is Propositions 8 and 20.

It remains to prove (ii), (iii) and (iv) for $\forall$-types. In this case we only need to produce maps in both directions — they will automatically compose to the identity by proof-irrelevance of 2-relations. The structure of the proof is the same for all of the three points.

For (ii) consider $(\tau_0, \rho_0, \tau_1, \rho_1) \in \mathsf{Eq}_{||}([\![\forall X.T]\!]_1 \bar{R})(f, g, h, l)$. We want to show that $(\Psi(\tau_0), \rho_0, \Psi(\tau_1), \rho_1) \in [\![\forall X.T]\!]_2 \mathsf{Eq}_{||}(\bar{R})(f, g, h, l)$, i.e. that for every 2-relation Q,

$$(\Psi(\tau_0)Q_{r0}, \rho_0 Q_{0r}, \Psi(\tau_1)Q_{r1}, \rho_1 Q_{1r}) \in [\![T]\!]_2(\mathsf{Eq}_{||}(\bar{R}), Q)(f_0 Q_{00}, g_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11}).$$

By condition A1.1, we have

$$(f_1 Q_{r0}, \rho_0 Q_{1r}, f_1 Q_{r1}, \rho_0 Q_{0r}) \in [\![T]\!]_2(\mathsf{Eq}_{||}(\bar{R}), Q)(f_0 Q_{00}, f_0 Q_{10}, h_0 Q_{01}, h_0 Q_{11})$$

and using the equalities $(f_1 \mathsf{Eq} Q_{00}, \tau_0 \mathsf{Eq} Q_{10}, h_1 \mathsf{Eq} Q_{01}, \tau_1 \mathsf{Eq} Q_{11})$, we can show

$$\begin{aligned}(f_0 Q_{00}, g_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11}, \Psi(\tau_0)Q_{r0}, \rho_0 Q_{0r}, \Psi(\tau_1)Q_{r1}, \rho_1 Q_{1r}) \\ \equiv (f_0 Q_{00}, f_0 Q_{10}, h_0 Q_{01}, h_0 Q_{11}, f_1 Q_{r0}, \rho_0 Q_{1r}, f_1 Q_{r1}, \rho_0 Q_{0r})\end{aligned}$$

We now transport across this equality to finish the argument.

Finally, in the other direction, if $(\rho_0, \rho_1, \rho_2, \rho_3) \in [\![\forall X.T]\!]_2 \mathsf{Eq}_{||}(\bar{R})(f, g, h, l)$, then $(\Theta(\rho_0), \rho_1, \Theta(\rho_2), \rho_3) \in \mathsf{Eq}_{||}([\![\forall X.T]\!]_1 \bar{R})(f, g, h, l)$ by straightforward calculation and the definition of $[\![\forall X.T]\!]_2$.

The case (iii) is just the same as the previous case, the only difference is that we now transport starting from condition (A1.2) and adjust the equalities along which we transport.

The last case (iv) is more complicated. Consider $(\tau_0, \tau_1, \rho_0, \rho_1) \in \mathbf{C}([\![\forall X.T]\!]_1 \bar{R})$ (f, g, h, l). We want to show that $(\Psi(\tau_0), \Psi(\tau_1), \rho_0, \rho_1) \in [\![\forall X.T]\!]_2 \mathbf{C}(\bar{R})$ (f, g, h, l), i.e. that for every 2-relation Q,

$$(\Psi(\tau_0)Q_{r0}, \Psi(\tau_1)Q_{0r}, \rho_0 Q_{r1}, \rho_1 Q_{1r}) \in [\![T]\!]_2(\mathbf{C}(\bar{R}), Q)(f_0 Q_{00}, g_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11}).$$

By condition A1.3, we have

$$(h_1 Q_{r0}, h_1 Q_{1r}, \rho_0 Q_{r1}, \rho_0 Q_{0r}) \in [\![T]\!]_2(\mathbf{C}\bar{R}, Q)(h_0 Q_{00}, h_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11})$$

and using the equalities

$$((\tau_1 Q_{00})^{-1}) \cdot f_1 \mathsf{Eq} Q_{00}, (\tau_1 \mathsf{Eq} Q_{10})^{-1} \cdot \tau_0 Q_{10}, (h_1 \mathsf{Eq} Q_{01})^{-1} \cdot h_1 \mathsf{Eq} Q_{01}, \mathsf{refl}),$$

we can show

$$\begin{aligned}(f_0 Q_{00}, g_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11}, \Psi(\tau_0)Q_{r0}, \Psi(\tau_1)Q_{0r}, \rho_0 Q_{r1}, \rho_1 Q_{1r}) \\ \equiv (h_0 Q_{00}, h_0 Q_{10}, h_0 Q_{01}, l_0 Q_{11}, h_1 Q_{r0}, h_1 Q_{1r}, \rho_0 Q_{r1}, \rho_0 Q_{0r})\end{aligned}$$

We can now transport across this equality to finish the argument. This requires the use of Lemma 23 (i) and (ii), and the fact that $(\tau_0, \tau_1, \rho_0, \rho_1) \in \mathbf{C}([\![\forall X.T]\!]_1 \bar{R})(f, g, h, l)$.

Finally, In the other direction, if $(\rho_0, \rho_1, \rho_2, \rho_3) \in [\![\forall X.T]\!]_2 \mathbf{C}(\bar{R})(f, g, h, l)$, then $(\Theta(\rho_0), \Theta(\rho_1), \rho_2, \rho_3) \in \mathbf{C}([\![\forall X.T]\!]_1 \bar{R})(f, g, h, l)$ by straightforward calculation and the definition of $[\![\forall X.T]\!]_2$. $\square$

Proof (of Lemma 23).

(i) Since

$$(f_1 R, f_1 \mathsf{Eq} R_0, f_1 R, f_1 \mathsf{Eq} R_1) \in [\![T]\!]_2(\mathbf{C} \circ \mathsf{Eq}(\vec{A}), \mathsf{Eq}_=(R))(f_0 R_0, f_0 R_1, g_0 R_0, g_0 R_1)$$

and $[\![T]\!]_2(\mathbf{C} \circ \mathsf{Eq}(\vec{A}), \mathsf{Eq}_=(R)) = [\![T]\!]_2(\mathsf{Eq}_= \circ \mathsf{Eq}(\vec{A}), \mathsf{Eq}_=(R)) \cong \mathsf{Eq}_=([\![T]\!]_1 (\mathsf{Eq}\vec{A}, R))$, by Theorem 22 (iii), the thesis follows.

(ii) By assumption,

$$(f_1 R, \phi \mathsf{Eq} R_0, g_1 R, \phi \mathsf{Eq} R_1) \in [\![T]\!]_2(\mathsf{Eq}_= \mathsf{Eq}\vec{A}, \mathsf{Eq}_= R)(f_0 R_0, f_1 R_1, g_0 R_0, g_0 R_1).$$

By Theorem 22 (iii), $[\![T]\!]_2(\mathsf{Eq}_= \mathsf{Eq}\vec{A}, \mathsf{Eq}_= R) \cong \mathsf{Eq}_=([\![T]\!]_1(\mathsf{Eq}\vec{A}, R))$, hence we have $\mathsf{tr}((\phi \mathsf{Eq} R_0)^{-1}, (g_1 \mathsf{Eq} R_1)^{-1}) g_1 R = \mathsf{tr}(f_1 \mathsf{Eq} R_0, \phi \mathsf{Eq} R_1) f_1 R$. If we now transport $(f_1 R, \phi \mathsf{Eq} R_0, g_1 R, \phi \mathsf{Eq} R_1)$ along the equality proof $((f_1 \mathsf{Eq} R_0)^{-1}$, $(f_1 \mathsf{Eq} R_1)^{-1}, (\phi \mathsf{Eq} R_0)^{-1}, (g_0 \mathsf{Eq} R_1)^{-1})$, the result follows.

(iii) By assumption,

$$(g_1 Q_{r0}, g_1 Q_{0r}, g_1 Q_{r1}, g_1 Q_{1r}) \in [\![T]\!]_2(\mathsf{Eq}_2 \vec{A}, Q)(g_0 Q_{00}, g_0 Q_{10}, g_0 Q_{01}, g_0 Q_{11})$$

We can transport $(g_1 Q_{r0}, g_1 Q_{0r}, g_1 Q_{r1}, g_1 Q_{1r})$ along the equality $((\phi \mathsf{Eq} Q_{00})^{-1}$, $(g_1 \mathsf{Eq} Q_{10})^{-1}, (g_1 \mathsf{Eq} Q_{01})^{-1}, (g_1 \mathsf{Eq} Q_{11})^{-1})$. By (i) and (ii), condition (A0), and $[\![T]\!]_2(\mathsf{Eq}_2 \vec{A}, Q) = [\![T]\!]_2(\mathbf{C} \circ \mathsf{Eq}\vec{A}, Q)$, the thesis follows. □

Proof (of Theorem 25). We need to check that the β- and η-rules for both term and type abstraction are respected. For term abstraction, this follows from Lemmas 13 and 16.

We next consider the η-rule for type abstraction. Let $\Gamma; \Delta \vdash t : \forall X.T$ be given. Let $[\![t]\!]_0 \vec{A}\gamma = (f_0, f_1)$. Showing $[\![\Lambda X.t[X]]\!]_0 \equiv [\![t]\!]_0$ means giving $p_0 : \mathsf{Id}(\lambda A.f_0 A, f_0)$ and $p_1 : \mathsf{Id}(\lambda R.(\mathsf{tr}(p_0 \bullet (\mathsf{snd}(([\![t]\!]_0 \bar{A}\gamma)\mathsf{Eq}(R_0))))^{-1}([\![t]\!]_1 \mathsf{Eq}(\bar{A})\Theta_{\Delta,0} (\mathsf{refl}(\gamma))\, R)), \mathsf{snd}([\![t]\!]_0 \vec{A}\gamma))$. For p_0, we choose $p_0 = \mathsf{refl}$. Note that

$$([\![t]\!]_1 \mathsf{Eq}(\bar{A})\Theta_{\Delta,0}(\mathsf{refl}(\gamma))\, R)) = \Theta_{\Delta,0}(\mathsf{refl}([\![t]\!]_0 \vec{A}\gamma))\, R =$$

$$\mathsf{tr}(f_1 \mathsf{Eq}(R_0), \mathsf{refl}) f_1 R$$

under the equivalence with respect to $\tau = \mathsf{refl}$, and

$$\mathsf{tr}(\mathsf{refl} \bullet (\mathsf{snd}(([\![t]\!]_0 \bar{A}\gamma)\mathsf{Eq}(R_0))))^{-1} = \mathsf{tr}(f_1 \mathsf{Eq}(R_0), \mathsf{refl})^{-1}.$$

In this way we can conclude

$$\begin{aligned}\mathsf{tr}(f_1 \mathsf{Eq}(R_0), \mathsf{refl})^{-1}(\Theta_{\Delta,0}(\mathsf{refl}([\![t]\!]_0 \vec{A}\gamma))\, R) &= \mathsf{tr}(f_1 \mathsf{Eq}(R_0), \mathsf{refl})^{-1} \mathsf{tr}(f_1 \mathsf{Eq}(R_0), \mathsf{refl}) f_1 R \\ &= f_1 R.\end{aligned}$$

Similarly, things are exactly lined up to make $\mathsf{tr}(\mathsf{pair}_=(p_0, p_1))([\![\Lambda X.t[X]]\!]_1) \equiv [\![t]\!]_1$ trivial.

For the β-rule, consider $\Gamma, X \vdash t \colon T$. We can use $p_{\bar{A}} = ([\![t]\!]_1 \mathsf{Eq}(\vec{A}, [\![S]\!]_0 \vec{A})\Theta_{\Delta,0}(\mathsf{refl}(\gamma)))^{-1}$ to prove $[\![(\Lambda X.t)[S]]\!]_0 \bar{A}\gamma \equiv [\![t[X \mapsto S]]\!]_0 \bar{A}\gamma$. This makes $\mathsf{tr}(p_{\bar{R}_0})([\![(\Lambda X.t)[S]]\!]_1)\bar{R}\bar{\gamma} \equiv [\![t[X \mapsto S]]\!]_1 \bar{R}\bar{\gamma}$ trivial, again using Lemma 1. □

References

Atkey, R.: Relational parametricity for higher kinds. In: Cégielski, P., Durand, A. (eds.) CSL 2012. LIPIcs, vol. 16, pp. 46–61. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl (2012)

Atkey, R., Ghani, N., Johann, P.: A relationally parametric model of dependent type theory. In: POPL, pp. 503–515. ACM (2014)

Bernardy, J.P., Coquand, T., Moulin, G.: A presheaf model of parametric type theory. In: Ghica, D.R. (ed.) MFPS, pp. 17–33. ENTCS, Elsevier, Amsterdam (2015)

Bernardy, J.P., Jansson, P., Paterson, R.: Proofs for free. J. Funct. Program. **22**, 107–152 (2012)

Bezem, M., Coquand, T., Huber, S.: A model of type theory in cubical sets. In: Types for Proofs and Programs (TYPES 2013). Leibniz International Proceedings in Informatics, vol. 26, pp. 107–128. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2014)

Brown, R., Higgins, P.J.: On the algebra of cubes. J. Pure Appl. Algebra **21**(3), 233–260 (1981)

Brown, R., Higgins, P.J., Sivera, R.: Nonabelian Algebraic Topology: Filtered Spaces, Crossed Complexes, Cubical Homotopy Groupoids. EMS Tracts in Mathematics, vol. 15. European Mathematical Society Publishing House, Zurich (2011)

Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**, 95–120 (1988)

Dunphy, B., Reddy, U.: Parametric limits. In: LICS, pp. 242–251 (2004)

Garner, R.: Two-dimensional models of type theory. Math. Struct. Comput. Sci. **19**(04), 687–736 (2009)

Ghani, N., Johann, P., Nordvall Forsberg, F., Orsanigo, F., Revell, T.: Bifibrational functorial semantics of parametric polymorphism. In: Ghica, D.R. (ed.) MFPS, pp. 67–83. ENTCS, Elsevier, Amsterdam (2015a)

Ghani, N., Nordvall Forsberg, F., Orsanigo, F.: Parametric polymorphism — universally. In: de Paiva, V., de Queiroz, R., Moss, L.S., Leivant, D., de Oliveira, A. (eds.) WoLLIC 2015. LNCS, vol. 9160, pp. 81–92. Springer, Heidelberg (2015b)

Grandis, M.: The role of symmetries in cubical sets and cubical categories (on weak cubical categories, I). Cah. Topol. Gom. Diff. Catg. **50**(2), 102–143 (2009)

Krishnaswami, N.R., Dreyer, D.: Internalizing relational parametricity in the extensional calculus of constructions. In: CSL, pp. 432–451 (2013)

Mac Lane, S.: Categories for the Working Mathematician, vol. 5. Springer, New York (1998)

Martin-Löf, P.: An intuitionistic theory of types. In: Twenty-Five Years of Constructive Type Theory (1972)

O'Hearn, P.W., Tennent, R.D.: Parametricity and local variables. J. ACM **42**(3), 658–709 (1995)

Polonsky, A.: Extensionality of lambda-*. In: Herbelin, H., Letouzey, P., Sozeau, M. (eds.) 20th International Conference on Types for Proofs and Programs (TYPES 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 39, pp. 221–250. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl (2015)

Reynolds, J.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing 1983, pp. 513–523. North-Holland, Amsterdam (1983)

Robinson, E., Rosolini, G.: Reflexive graphs and parametric polymorphism. In: LICS, pp. 364–371 (1994)

Strachey, C.: Fundamental concepts in programming languages. High. Order Symbolic Comput. **13**(1–2), 11–49 (2000)

The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics (2013). http://homotopytypetheory.org/book

Voevodsky, V.: The equivalence axiom and univalent models of type theory. Talk at CMU on 4 February 2010 (2010). http://arxiv.org/abs/1402.5556

Wadler, P.: Theorems for free! In: FPCA, pp. 347–359 (1989)

Wadler, P.: The Girard-Reynolds isomorphism (second edition). Theoret. Comput. Sci. **375**(1–3), 201–226 (2007)

Comprehending Ringads
For Phil Wadler, on the Occasion of his 60th Birthday

Jeremy Gibbons(✉)

Department of Computer Science, University of Oxford, Oxford, UK
jeremy.gibbons@cs.ox.ac.uk
http://www.cs.ox.ac.uk/jeremy.gibbons/

Abstract. *List comprehensions* are a widely used programming construct, in languages such as Haskell and Python and in technologies such as Microsoft's Language Integrated Query. They generalize from lists to arbitrary *monads*, yielding a lightweight idiom of imperative programming in a pure functional language. When the monad has the additional structure of a so-called *ringad*, corresponding to 'empty' and 'union' operations, then it can be seen as some kind of collection type, and the comprehension notation can also be extended to incorporate *aggregations*. Ringad comprehensions represent a convenient notation for expressing *database queries*. The ringad structure alone does not provide a good explanation or an efficient implementation of relational *joins*; but by allowing heterogeneous comprehensions, involving both bag and indexed table ringads, we show how to accommodate these too.

1 Introduction

We owe a lot to Phil Wadler, not least for his work over the years on monads and comprehensions. Wadler was an early proponent of list comprehensions as a syntactic feature of functional programming languages, literally writing the book (Peyton Jones 1987, Chap. 7) on them. Together with his student Phil Trinder, he argued (Trinder and Wadler 1989; Trinder 1991) for the use of comprehensions as a notation for database queries; this prompted a flourishing line of work within database programming languages, including the Kleisli language from Penn (Wong 2000) and LINQ from Microsoft (Meijer 2011), not to mention Wadler's own work on XQuery (Fernandez et al. 2001) and Links (Cooper et al. 2006).

Wadler was also the main driving force in explaining monads to functional programmers (Wadler 1992b), popularizing the earlier foundational work of Moggi (1991). He showed that the notions of list comprehensions and monads were related (Wadler 1992a), generalizing list comprehensions to arbitrary monads—of particular interest to us, to sets and bags, but also to monads like state and I/O. More recently, with Peyton Jones (Wadler and Peyton Jones 2007) he has extended the list comprehension syntax to support 'SQL-like' ordering and grouping constructs, and this extended comprehension syntax has subsequently been generalized to other monads too (Giorgidze et al. 2011).

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 132–151, 2016.
DOI: 10.1007/978-3-319-30936-1_7

In this paper, we look a little more closely at the use of comprehensions as a notation for queries. The monadic structure explains most of standard relational algebra, allowing for an elegant mathematical foundation for those aspects of database query language design. Unfortunately, monads per se offer no good explanation of relational joins, a crucial aspect of relational algebra: expressed as a comprehension, a typical equijoin $[(a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b]$ is very inefficient to execute. But the ingredients we need in order to do better are all there, in Wadler's work, as usual. The novel contribution of this paper is to bring together those ingredients: the generalizations of comprehensions to different monads and to incorporate grouping are sufficient for capturing a reasonable implementation of equijoins.

2 Comprehensions

The idea of comprehensions can be seen as one half of the adjunction between extension and intension in set theory—one can define a set by its *extension*, that is by listing its elements:

$$\{1, 9, 25, 49, 81\}$$

or by its *intension*, that is by characterizing those elements:

$$\{n^2 \mid 0 < n < 10 \wedge n \equiv 1 \pmod 2\}$$

Expressions in the latter form are called *set comprehensions*. They inspired the "set former" programming notation in the SETL language (Schwartz 1975; Schwartz et al. 1986), dating back to the late 1960s, and have become widely known through list comprehensions in languages like Haskell and Python.

2.1 List Comprehensions

Just as a warm-up, here is a reminder about Haskell's list comprehensions:

$$[2 \times a + b \mid a \leftarrow [1, 2, 3], b \leftarrow [4, 5, 6], b \text{ `}mod\text{` } a == 0]$$

This (rather concocted) example yields the list $[6, 7, 8, 8, 10, 12]$ of all values of the expression $2 \times a + b$, as a is drawn from $[1, 2, 3]$ and b from $[4, 5, 6]$ and such that b is divisible by a.

To the left of the vertical bar is the *term* (an expression). To the right is a comma-separated sequence of *qualifiers*, each of which is either a *generator* (of the form $a \leftarrow x$, with a variable a and a list expression x) or a *filter* (a boolean expression). The scope of a variable introduced by a generator extends to all subsequent generators and to the term.

Note that, in contrast to the naive set-theoretic inspiration, bound variables in list comprehensions need to be explicitly generated from some existing list, rather than being implicitly quantified. Without such a condition, the set-theoretic *axiom of unrestricted comprehension* ("for any predicate P, there exists a set B whose elements are precisely those that satisfy P") leads directly to Russell's Paradox; with the condition, we get the *axiom of specification* ("for any set A and predicate P, there exists a set B whose elements are precisely the elements of A that satisfy P"), which avoids the paradox.

The semantics of list comprehensions is defined by translation; see for example Wadler's chapter of Peyton Jones's book (1987, Chap. 7). The translation can be expressed equationally as follows:

$$\begin{array}{ll}
[\,e \mid \,] & = [\,e\,] \\
[\,e \mid b\,] & = \mathbf{if}\ b\ \mathbf{then}\ [\,e\,]\ \mathbf{else}\ [\,] \\
[\,e \mid a \leftarrow x\,] & = \mathit{map}\ (\lambda a \rightarrow e)\ x \\
[\,e \mid q, q'\,] & = \mathit{concat}\ [\,[\,e \mid q'\,] \mid q\,]
\end{array}$$

(Here, the first clause involves the empty sequence of qualifiers. This is not allowed in Haskell, but it is helpful in simplifying the translation.)

Applying this translation to the example at the start of the section gives

$$\begin{array}{l}
\quad [2 \times a + b \mid a \leftarrow [1,2,3], b \leftarrow [4,5,6], b\ \text{`}\mathit{mod}\text{`}\ a == 0] \\
= \mathit{concat}\ (\mathit{map}\ (\lambda a \rightarrow \mathit{concat}\ (\mathit{map}\ (\lambda b \rightarrow \\
\qquad \mathbf{if}\ b\ \text{`}\mathit{mod}\text{`}\ a == 0\ \mathbf{then}\ [2 \times a + b]\ \mathbf{else}\ [\,])\ [4,5,6]))\ [1,2,3]) \\
= [6,7,8,8,10,12]
\end{array}$$

More generally, a generator may match against a pattern rather than just a variable. In that case, it may bind multiple (or indeed no) variables at once; moreover, the match may fail, in which case it is discarded. This is handled by modifying the translation for generators to use a function defined by pattern-matching, rather than a straight lambda-abstraction:

$$[\,e \mid p \leftarrow x\,] = \mathit{concat}\ (\mathit{map}\ (\lambda a \rightarrow \mathbf{case}\ a\ \mathbf{of}\ p \rightarrow [\,e\,]; \_ \rightarrow [\,])\ x)$$

or, perhaps more perspicuously,

$$\begin{array}{ll}
[\,e \mid p \leftarrow x\,] = & \mathbf{let}\ h\ p = [\,e\,] \\
& \qquad h\ \_ = [\,] \\
& \mathbf{in}\ \mathit{concat}\ (\mathit{map}\ h\ x).
\end{array}$$

2.2 Monad Comprehensions

It is clear from the above translation that the necessary ingredients for list comprehensions are *map*, singletons, *concat*, and the empty list. The first three are the operations arising from lists as a functor and a monad, which suggests that the same translation might be applicable to other monads too.

$$\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ m\ \mathbf{where} \\
\quad \mathit{fmap} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b
\end{array}$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b \\
\quad \mathit{mult} \ :: m\ (m\ a) \to m\ a \\
\quad \mathit{mult} \ = (\gg\!= \mathit{id})
\end{array}
$$

(we write *mult* for the multiplication of the monad, rather than *join* as in Haskell, to avoid confusion with the relational joins discussed later). But the fourth ingredient, the empty list, does not come from the functor and monad structures; that requires an extra assumption:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZero}\ m\ \textbf{where} \\
\quad \mathit{mzero} :: m\ a
\end{array}
$$

Given this extra assumption, the translation for list comprehensions can be generalized to other monads:

$$
\begin{array}{ll}
[\,e \mid\,] & = \mathit{return}\ e \\
[\,e \mid b\,] & = \textbf{if}\ b\ \textbf{then}\ \mathit{return}\ e\ \textbf{else}\ \mathit{mzero} \\
[\,e \mid p \leftarrow x\,] & = \textbf{let}\ h\ p = \mathit{return}\ e; h\ \_ = \mathit{mzero}\ \textbf{in}\ \mathit{mult}\ (\mathit{fmap}\ h\ x) \\
[\,e \mid q, q'\,] & = \mathit{mult}\ [[\,e \mid q'\,] \mid q\,]
\end{array}
$$

Note that $[\,e \mid\,] = [\,e \mid \mathit{True}\,]$, so the empty case is not really needed; and that we could have written $\mathit{mult}\ (\mathit{fmap}\ h\ x)$ as $x \gg\!= h$, but this would not be so obviously a generalization of the list instance. The actual monad to be used is implicit in the notation, possibly determined by any input value m on the right-hand side of a generator; if we want to be explicit, we could write a subscript, as in "$[\,e \mid q\,]_{\mathit{List}}$".

This translation is different from the one used in the Haskell language specification for **do** notation (Haskell 2010, Sect. 3.14) and in the GHC documentation for monad comprehensions (GHC 7.10 2015, Sect. 7.3.15), which are arguably a little awkward: the empty list crops up in two different ways in the translation of list comprehensions—for filters, and for generators with patterns—and these are generalized in two different ways to other monads: to the *mzero* method of the *MonadPlus* class in the first case, and the *fail* method of the *Monad* class in the second. It is perhaps neater to have a monad subclass *MonadZero* with a single method subsuming both these operators. (The *fail* method of Haskell's *Monad* class is generally unpopular. There is a current proposal (Luposchainsky 2015) to remove *fail* to a *MonadFail* subclass, but the proposal would still retain both *fail* and *mzero*, and their applications in **do** notation and monad comprehensions.) Of course, this change does mean that the translation forces a monad comprehension with filters to be interpreted in an instance of the *MonadZero* subclass rather than just of *Monad*—the type class constraints that are generated depend on the features used in the comprehension (as already happens in Haskell for generators with patterns).

Taking this approach gives basically Wadler's monad comprehension notation (Wadler 1992a); it loosely corresponds to Haskell's **do** notation, except that the term is a value to the left of a vertical bar rather than a computation at the end, and that filters are just boolean expressions rather than introduced using *guard*.

We might impose the law that *mult* distributes over *mzero*, in the sense

$$\mathit{mult}\ \mathit{mzero} = \mathit{mzero}$$

or, in terms of comprehensions,

$$[\,e \mid a \leftarrow \mathit{mzero}\,] = \mathit{mzero}$$

Informally, this means that any failing steps of the computation cleanly cut off subsequent branches. Another way of saying it is that *mzero* is a 'left' zero of composition:

$$\mathit{mzero} \ggg\!= k = \mathit{mzero}$$

Conversely, we do not require that *mzero* is a 'right' zero of composition:

$$m \ggg\!= \lambda a \rightarrow \mathit{mzero} \neq \mathit{mzero} \qquad \text{(in general)}$$

Imposing this law would have the consequence that a failing step also cleanly erases any effects from earlier parts of the computation, which is too strong a requirement for many monads—particularly those of the "launch missiles now" variety.

2.3 Heterogeneous Comprehensions

We have seen that comprehensions can be interpreted in an arbitrary monad; for example, $[\,a^2 \mid a \leftarrow x, \mathit{odd}\ a\,]_{\mathit{Set}}$ denotes the set of the squares of the odd elements of the set x, whereas $[\,a^2 \mid a \leftarrow y, \mathit{odd}\ a\,]_{\mathit{Bag}}$ denotes the bag of squares of odd elements of bag y. These are both 'homogeneous comprehensions', each involving just one monad; can we make sense of 'heterogeneous comprehensions', involving several different monads?

For monads M and N, a monad morphism $\varphi : M \rightarrow N$ is a natural transformation $M \overset{.}{\rightarrow} N$—that is, a family $\varphi_\alpha :: M\ \alpha \rightarrow N\ \alpha$ of arrows, coherent in the sense that $\varphi_\beta \cdot \mathit{fmap}_M\ f = \mathit{fmap}_N\ f \cdot \varphi_\alpha$ for $f :: \alpha \rightarrow \beta$—that also preserves the monad structure:

$$\begin{aligned} \varphi \cdot \mathit{return}_M &= \mathit{return}_N \\ \varphi \cdot \mathit{mult}_M &= \mathit{mult}_N \cdot \varphi \cdot \mathit{fmap}_M\ \varphi = \mathit{mult}_N \cdot \mathit{fmap}_N\ \varphi \cdot \varphi \end{aligned}$$

A monad morphism behaves nicely with respect to monad comprehensions—a comprehension interpreted in monad M, using inputs of type M, with the result coerced via a monad morphism $\varphi : M \overset{.}{\rightarrow} N$ to monad N, is equivalent to the comprehension interpreted in monad N in the first place, with the inputs having been coerced to type N. Informally, there will be no surprises arising from when coercions take place, because the results are the same whatever stage this happens. This property is straightforward to show by induction over the structure of the comprehension.

For example, if $\mathit{bag2set} : \mathit{Bag} \stackrel{.}{\to} \mathit{Set}$ is the obvious monad morphism from bags to sets, discarding information about the multiplicity of repeated elements, and x a bag of numbers, then

$$\mathit{bag2set}\ [a^2 \mid a \leftarrow x, \mathit{odd}\ a]_{\mathit{Bag}} = [a^2 \mid a \leftarrow \mathit{bag2set}\ x, \mathit{odd}\ a]_{\mathit{Set}}$$

and both yield the set of squares of the odd members of bag x. As a notational convenience, we might elide use of the monad morphism when it is 'obvious from context'—we might write just $[a^2 \mid a \leftarrow x, \mathit{odd}\ a]_{\mathit{Set}}$ even when x is a bag, relying on the 'obvious' morphism *bag2set*. This would allow us to write heterogeneous comprehensions such as

$$[a + b \mid a \leftarrow [1,2,3], b \leftarrow \wr 4,4,5 \int]_{\mathit{Set}} = \{5,6,7,8\}$$

(writing $\wr \ldots \int$ for the extension of a bag), instead of the more pedantic

$$[a + b \mid a \leftarrow \mathit{list2set}\ [1,2,3], b \leftarrow \mathit{bag2set}\ \wr 4,4,5 \int]_{\mathit{Set}}$$

Sets, bags, and lists are all members of the so-called *Boom Hierarchy* of types (Backhouse 1988), consisting of types whose values are constructed from 'empty' and 'singleton's using a binary 'union' operator; 'empty' is the unit of 'union', and the hierarchy pertains to which properties out of associativity, commutativity, and idempotence the 'union' operator enjoys. The name 'Boom hierarchy' seems to have been coined at an IFIP WG2.1 meeting by Stephen Spackman, for an idea due to Hendrik Boom, who by happy coincidence is named in his native language after another member of the hierarchy, namely (externally labelled, possibly empty, binary) trees:

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

for which 'union' is defined as a smart *Bin* constructor, ensuring that the empty tree is a unit:

$$\begin{aligned} &\mathit{bin}\ \mathit{Empty}\ u &&= u \\ &\mathit{bin}\ t\ \mathit{Empty} &&= t \\ &\mathit{bin}\ t\ u &&= \mathit{Bin}\ t\ u \end{aligned}$$

but with no other properties of *bin*. It is merely a historical accident that the familiar members of the Boom Hierarchy are linearly ordered by their properties; one can perfectly well imagine other more exotic combinations of the laws—such as 'mobiles', in which the union operator is commutative but not associative or idempotent (Uustalu 2015).

There is a forgetful function from any poorer member of the Boom Hierarchy to a richer one, flattening some distinctions by imposing additional laws—for example, from bags to sets, flattening distinctions concerning multiplicity—and one could class these forgetful functions as 'obvious' morphisms. On the other hand, any morphisms in the opposite direction—such as sorting, from bags to lists, and one-of-each, from sets to bags—are not 'obvious' (and in particular, contradicting Wong (1994, p. 114–115), they are not monad morphisms), and so should not be elided; and similarly, it would be hard to justify as 'obvious' any morphisms involving non-members of the Boom Hierarchy, such as probability distributions.

3 Ringads and Collections

One more ingredient is needed in order to characterize monads that correspond to 'collection' types such as sets and lists, as opposed to other monads such as *State* and *IO*; that ingredient is an analogue of set union or list append. It's not difficult to see that this is inexpressible in terms of the operations introduced so far: given only collections x of at most one element, any comprehension using generators of the form $a \leftarrow x$ will only yield another such collection, whereas the union of two one-element collections will in general have two elements.

To allow any finite collection to be expressed, it suffices to introduce a binary union operator *mplus*:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}\\ \quad \mathit{mplus} :: m\ a \rightarrow m\ a \rightarrow m\ a\end{array}$$

We require *mult* to distribute over union, in the following sense:

$$\mathit{mult}\ (x\ \text{`}\mathit{mplus}\text{`}\ y) = \mathit{mult}\ x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mult}\ y$$

or, in terms of comprehensions,

$$[\,e \mid a \leftarrow x\ \text{`}\mathit{mplus}\text{`}\ y, q\,] = [\,e \mid a \leftarrow x, q\,]\ \text{`}\mathit{mplus}\text{`}\ [\,e \mid a \leftarrow y, q\,]$$

or of monadic bind:

$$(x\ \text{`}\mathit{mplus}\text{`}\ y) \gg\!= k = (x \gg\!= k)\ \text{`}\mathit{mplus}\text{`}\ (y \gg\!= k)$$

Note that, in contrast to the Haskell libraries, we have identified separate type classes *MonadZero*, *MonadPlus* for the two methods *mzero*, *mplus*. We have already seen that there are uses of *mzero* that do not require *mplus*; and conversely, there is no a priori reason to insist that all uses of *mplus* have to be associated with an *mzero*.

But for our model of collection types, we will insist on a monad in both *MonadZero* and *MonadPlus*. We will call this combination a *ringad*, a name coined by Wadler (1990):

$$\textbf{class}\ (\mathit{MonadZero}\ m, \mathit{MonadPlus}\ m) \Rightarrow \mathit{Ringad}\ m$$

There are no additional methods; the class *Ringad* is the intersection of the two parent classes *MonadZero* and *MonadPlus*, thereby denoting the union of the two interfaces. Haskell gives us no good way to state the laws that should be required of instances of a type class such as *Ringad*, but they are the three monad laws, distribution of *mult* over *mzero* and *mplus*:

$$\begin{aligned} &\mathit{mult}\ \mathit{mzero} &&= \mathit{mzero} \\ &\mathit{mult}\ (x\ \text{`}\mathit{mplus}\text{`}\ y) &&= \mathit{mult}\ x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mult}\ y \end{aligned}$$

and *mzero* being the unit of *mplus*:

$$\mathit{mzero}\ \text{`}\mathit{mplus}\text{`}\ x = x = x\ \text{`}\mathit{mplus}\text{`}\ \mathit{mzero}$$

(There seems to be no particular reason to insist also that *mplus* be associative; we discuss the laws further in Sect. 3.2.) To emphasize the additional constraints, we will write "$\varnothing$" for "*mzero*" and "$\uplus$" for "*mplus*" when discussing a ringad. All members of the Boom hierarchy—sets, bags, lists, trees, and exotica too—are ringad instances. Another ringad instance, but one that is not a member of the Boom Hierarchy, is the type of probability distributions—either normalized, with a weight-indexed family of union operators, or unnormalized, with an additional scaling operator.

3.1 Aggregation

The well-behaved operations over monadic values are called the *algebras* for that monad—functions k such that $k \cdot \mathit{return} = \mathit{id}$ and $k \cdot \mathit{mult} = k \cdot \mathit{fmap}\ k$. In particular, *mult* is itself a monad algebra. When the monad is also a ringad, k necessarily distributes also over $\uplus$—it is a nice exercise to verify that defining $a \oplus b = k\ (\mathit{return}\ a \uplus \mathit{return}\ b)$ establishes that $k\ (x \uplus y) = k\ x \oplus k\ y$. Without loss of generality, we write $\mathit{reduce}(\oplus)$ for k; these are the 'reductions' of the Bird–Meertens Formalism (Backhouse 1988). In that case, $\mathit{mult} = \mathit{reduce}(\uplus)$ is a ringad algebra.

The algebras for a ringad amount to aggregation functions for a collection: the sum of a bag of integers, the maximum of a set of naturals, and so on. We could extend the comprehension notation to encompass aggregations too, for example by adding an optional annotation, writing say "$[e \mid q]^{\oplus}$"; but this doesn't add much, because we could just have written "$\mathit{reduce}(\oplus)\ [e \mid q]$" instead. We could generalize from reductions $\mathit{reduce}(\oplus)$ to collection homomorphisms $\mathit{reduce}(\oplus) \cdot \mathit{fmap}\ f$; but this doesn't add much either, because the map is easily combined with the comprehension—it's easy to show the 'map over comprehension' property

$$\mathit{fmap}\ f\ [e \mid q] = [f\ e \mid q]$$

Fegaras and Maier (2000) develop a *monoid comprehension calculus* around such aggregations; but their name is arguably inappropriate, because it adds nothing essential to insist on associativity of the binary aggregating operator—'ringad comprehension calculus' might be a better term.

Note that, for $\mathit{reduce}(\oplus)$ to be well-defined, $\oplus$ must satisfy all the laws that $\uplus$ does—$\oplus$ must be associative if $\uplus$ is associative, and so on. It is not hard to show, for instance, that there is no $\oplus$ on sets of numbers for which $\mathit{sum}\ (x \cup y) = \mathit{sum}\ x \oplus \mathit{sum}\ y$; such an $\oplus$ would have to be idempotent, which is inconsistent with its relationship with *sum*. (So, although $[a^2 \mid a \leftarrow y, \mathit{odd}\ a]^{+}_{\mathit{Bag}}$ denotes the sum of the squares of the odd elements of bag y, the expression $[a^2 \mid a \leftarrow x, \mathit{odd}\ a]^{+}_{\mathit{Set}}$ (with x a set) is not defined, because $+$ is not idempotent.) In particular, $\mathit{reduce}(\oplus)\ \varnothing$ must be the unit of $\oplus$, which we write $1_\oplus$.

We can calculate from the definition

$$[e \mid q]^{\oplus} = \mathit{reduce}(\oplus)\ [e \mid q]$$

the following translation rules for aggregations:

$$\begin{aligned}
[e \mid\ ]^{\oplus} &= e \\
[e \mid b]^{\oplus} &= \textbf{if } b \textbf{ then } e \textbf{ else } 1_\oplus \\
[e \mid p \leftarrow x]^{\oplus} &= \textbf{let } h\ p = e; h\ \_ = 1_\oplus \textbf{ in } \mathit{reduce}(\oplus)\ (\mathit{fmap}\ h\ x) \\
[e \mid q, q']^{\oplus} &= [[e \mid q']^{\oplus} \mid q]^{\oplus}
\end{aligned}$$

Multiple aggregations can be performed in parallel: the so-called *banana split theorem* (Fokkinga 1990) shows how to lift two binary operators $\oplus$ and $\otimes$ into a single binary operator $\circledast$ on pairs, such that

$$(\mathit{reduce}(\oplus)\ x, \mathit{reduce}(\otimes)\ x) = \mathit{reduce}(\circledast)\ x$$

for every m.

If we are to allow aggregations over heterogeneous ringad comprehensions with automatic coercions, then we had better insist that the monad morphisms $\varphi : M \stackrel{.}{\to} N$ concerned are also ringad morphisms, that is, homomorphisms over the empty and union structure:

$$\begin{aligned}
\varphi\ (\varnothing_M) &= \varnothing_N \\
\varphi\ (x \uplus_M y) &= \varphi\ x \uplus_N \varphi\ y
\end{aligned}$$

(UnionUnit)	$\varnothing \uplus x$	$= x = x \uplus \varnothing$
(UnionAssoc)	$x \uplus (y \uplus z)$	$= (x \uplus y) \uplus z$
(EmptyBind)	$\varnothing \gg\!= k$	$= \varnothing$
(BindEmpty)	$x \gg\!= \lambda a \to \varnothing$	$= \varnothing$
(UnionBind)	$(x \uplus y) \gg\!= k$	$= (x \gg\!= k) \uplus (y \gg\!= k)$
(BindUnion)	$x \gg\!= \lambda a \to k\ a \uplus k'\ a$	$= (x \gg\!= k) \uplus (x \gg\!= k')$

Fig. 1. Six possible laws for ringads (not all of them required)

3.2 Notes on "Notes on Monads and Ringads"

As observed above, Wadler introduced the term 'ringad' a quarter of a century ago in an unpublished document *Notes on Monads and Ringads* (Wadler 1990).

He requires both distributivities for "monads with zero" (the EmptyBind and BindEmpty laws in Fig. 1); for ringads, he additionally requires that $\uplus$ is associative (UnionAssoc), with $\varnothing$ as its unit (UnionUnit), and that bind distributes from the right through $\uplus$ (UnionBind). He does not require that bind also distributes from the left through $\uplus$ (BindUnion).

The name 'ringad' presumably was chosen because, just as monads are monoids in a category of endofunctors, considered as a monoidal category under composition, ringads are 'right near-semirings' in such a category, considered as what Uustalu (2015) calls a 'right near-semiring category' under composition and product. A *right near-semiring* is an algebraic structure $(R, +, \times, 0, 1)$ in which $(R, +, 0)$ and $(R, \times, 1)$ are monoids, $\times$ distributes rightwards over $+$ (that is, $(a + b) \times c = (a \times c) + (b \times c)$) and is absorbed on the right by zero (that is, $0 \times a = a$). It is called 'semi-' because there is no additive inverse (a semiring is sometimes called a 'rig'), 'near-' because addition need not be commutative, and 'right-' because we require distributivity and absorption only from the right, not from the left too. This structure doesn't quite fit our circumstances, because we haven't insisted on associativity of the additive operation $\uplus$; but Wadler does in his note (and there seems to be no standard name for the structure when associativity of addition is dropped).

Wadler's note was cited in a few papers from the 1990s (Trinder 1991; Watt and Trinder 1991; Boiten and Hoogendijk 1995, 1996; Suciu 1993a, 1993b), of which only Trinder's DBPL paper (Trinder 1991) seems to have been formally published. Some other works (Wong 1994; Grust 1999) describe ringads, but cite Trinder's published paper (Trinder 1991) instead of Wadler's unpublished note (Wadler 1990).

Somewhat frustratingly, despite all citing a common source, the various papers differ on the laws they require for a ringad. For the monoidal aspects, everyone agrees that $\varnothing$ should be a unit of $\uplus$ (UnionUnit), but opinions vary on whether $\uplus$ should at least be associative (UnionAssoc). Trinder (1991) does not require (UnionAssoc); Boiten and Hoogendijk (1995, 1996) do, as does Uustalu (2015), and Wadler (1997) in a mailing list message about monads with zero and plus but not explicitly mentioning ringads; Grust requires it for his own constructions (Grust 1999, Sect. 72), but implies (Grust 1999, Sect. 73) that ringads need not satisfy it; Kiselyov (2015) argues that one should not insist on associativity without also insisting on commutativity, which not everyone wants.

For the distributivity aspects, everyone agrees that monadic bind should distribute from the right over $\uplus$ (UnionBind). Moreover, everyone agrees that bind need not distribute from the left over $\uplus$ (BindUnion); such a law would at least suggest that $\uplus$ should be commutative—in particular, it does not hold for the list monad. Nearly everyone agrees that bind should also distribute from the right over $\varnothing$ (EmptyBind)—the exception being Boiten and Hoogendijk (1995, 1996), who appear not to require it. Opinions vary as to whether bind should also distribute from the left over $\varnothing$ (BindEmpty); such a law does not hold for a monad that combines a 'global' log with the possibility of failure (such as Haskell's $MaybeT\ (Writer\ w)\ a$, which is equivalent to $(w, Maybe\ a)$), although it does hold when the log is 'local' (as with Haskell's *WriterT w Maybe a*, which is equivalent to *Maybe* (w, a)). Trinder (1991) does require (BINDEMPTY), in addition to (EMPTYBIND). Boiten and Hoogendijk (1995, 1996) require (BINDEMPTY), *instead* of (EMPTYBIND). Grust (1999, Sect. 58) requires both directions (EMPTYBIND) and (BINDEMPTY) for "monads with zero", following Wadler (1992a), but imposes only distributivity from the right (EMPTYBIND) for ringads (Grust 1999, Sect. 73). Wadler himself "would usually insist on" only (EMPTYBIND) and not (BINDEMPTY), writing later (Wadler 1997) about "monads with zero and plus". Buneman et al. (1995) attribute to ringads distributivity from the right over both $\varnothing$ (EMPTYBIND) and $\uplus$ (UNIONBIND), saying that "they seem to express fundamental properties", but say of the two distributivities from the left (BINDEMPTY, BINDUNION) that their "status [...] is unclear". (Whether one thinks of a particular property of bind as 'distributing from the left' or 'distributing from the right' depends of course on which way round one writes bind's arguments, and this varies from author to author.)

4 Comprehending Joins

Comprehensions form a convenient syntax for database queries, explaining the selection and projection operations of relational algebra. For example, consider a table of invoices

```
invoices(name, address, amount, due)
```

The SQL query

```
SELECT  name, address, amount
FROM    invoices
WHERE   due < today
```

which selects the overdue invoices and projects out the corresponding customer names and addresses and the outstanding amounts, can be expressed as the following comprehension:

$$\begin{aligned}&[\,(\mathit{name}, \mathit{address}, \mathit{amount})\\ &\mid (\mathit{name}, \mathit{address}, \mathit{amount}, \mathit{due}) \leftarrow \mathit{invoices},\\ &\quad \mathit{due} < \mathit{today}\,]\end{aligned}$$

This is a reasonable approximation to how database systems implement selection and projection. However, the comprehension notation does *not* explain the third leg of relational algebra, namely join operations. For example, if the invoice database is normalized so that customer names and addresses are in a separate table from the invoices,

```
customers(cid, name, address)
invoices(cust, amount, due)
```

then the query becomes

```
SELECT  name, address, amount
FROM    customers, invoices
WHERE   cid = cust AND due < today
```

and the obvious comprehension version

$$\begin{aligned}&[\,(\mathit{name}, \mathit{address}, \mathit{amount})\\ &\mid (\mathit{cid}, \mathit{name}, \mathit{address}) \leftarrow \mathit{customers}, (\mathit{cust}, \mathit{amount}, \mathit{due}) \leftarrow \mathit{invoices},\\ &\;\; \mathit{cid} == \mathit{cust}, \mathit{due} < \mathit{today}]\end{aligned}$$

entails a traversal over the entire cartesian product of the *customers* and *invoices* tables, only subsequently to discard the great majority of pairs of tuples. This is an extremely naive approximation to how database systems implement joins.

In this section, we sketch out how to achieve a more reasonable implementation for joins too, while still preserving the convenient comprehension syntax. We do this only for *equijoins*, that is, for joins where the matching condition on tuples is equality under two functions. The techniques we use are two extensions to comprehension syntax: *parallel* comprehensions (Plasmeijer and van Eekelen 1995), which introduce a kind of 'zip' operation, and *comprehensive* comprehensions (Wadler and Peyton Jones 2007), which introduce a 'group by'. Both were introduced originally just for list comprehensions, but have recently (Giorgidze et al. 2011) been generalized to other monads. The result will be a way of writing database queries using joins that can be executed in time linear in the number of input tuples, instead of quadratic.

4.1 Parallel Comprehensions

Parallel list comprehensions were introduced in Clean 1.0 in 1995 (Plasmeijer and van Eekelen 1995), and subsequently as a Haskell extension in GHC around 2001 (GHC 5.0 2001), desugaring to applications of the *zip* function:

$$\begin{aligned}&\mathit{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]\\ &\mathit{zip}\ (a:x)\ (b:y) = (a,b) : \mathit{zip}\ x\ y\\ &\mathit{zip}\ \_\qquad\quad \_\qquad\ = [\,]\end{aligned}$$

For example,

$$\begin{aligned}&\;\;[a+b \mid a \leftarrow [1,2,3] \mid b \leftarrow [4,5]]\\ &= \mathit{zip}\ [1,2,3]\ [4,5] \ggg\!= \lambda(a,b) \rightarrow \mathit{return}\ (a+b)\\ &= [5,7]\end{aligned}$$

Essentially the same translation can be used for an arbitrary monad (Giorgidze et al. 2011):

$$[e \mid (q \mid r), s] = \mathit{mzip}\ [\mathsf{v}^q \mid q]\ [\mathsf{v}^r \mid r] \ggg\!= \lambda(\mathsf{v}^q, \mathsf{v}^r) \rightarrow [e \mid s]$$

(where v^q denotes the tuple of variables bound by qualifiers q), provided that the monad supports an appropriate 'zip' function. In the GHC implementation, the appropriate choice of *mzip* function is type-directed: *mzip* is a method of a type class *MonadZip*, a subclass of *Monad*:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadZip}\ m\ \textbf{where}\\ \quad \mathit{mzip} :: m\ a \to m\ b \to m\ (a,b)\end{array}$$

of which the monad in question should be an instance.

There is some uncertainty about what laws one should require of instances of *mzip*, beyond naturality. The GHC documentation specifies only an information preservation requirement: that if two computations x, y have the same 'shape'

$$\mathit{fmap}_M\ (\mathit{const}\ ())\ x = \mathit{fmap}_M\ (\mathit{const}\ ())\ y$$

then zipping them can be undone:

$$(x,y) = \textbf{let}\ z = \mathit{mzip}\ x\ y\ \textbf{in}\ (\mathit{fmap}_M\ \mathit{fst}\ z, \mathit{fmap}_M\ \mathit{snd}\ z)$$

Petricek (2011) observes that one probably also wants associativity:

$$\mathit{fmap}_M\ (\lambda(a,(b,c)) \to ((a,b),c))\ (\mathit{mzip}\ x\ (\mathit{mzip}\ y\ z)) = \mathit{mzip}\ (\mathit{mzip}\ x\ y)\ z$$

so that it doesn't matter how one brackets a comprehension $[(a,b,c) \mid a \leftarrow x \mid b \leftarrow y \mid c \leftarrow z]$ with three parallel generators. One might also consider unit and commutativity properties.

4.2 Grouping Comprehensions

Grouping list comprehensions were introduced (along with 'ordering') by Wadler and Peyton Jones (2007), specifically motivated by trying to "make it easy to express the kind of queries one would write in SQL". Here is a simple example:

$$\begin{array}{l}\quad [(\mathit{the}\ a, b) \mid (a,b) \leftarrow [(1,\texttt{'p'}),(2,\texttt{'q'}),(1,\texttt{'r'})],\\ \qquad\qquad\qquad \textbf{then group by}\ a\ \textbf{using}\ \mathit{groupWith}]\\ = [(1,\texttt{"pr"}),(2,\texttt{"q"})]\end{array}$$

Here, $\mathit{the} :: [a] \to a$ returns the common value of a non-empty list of equal elements, and $\mathit{groupWith} :: \mathit{Ord}\ b \Rightarrow (a \to b) \to [a] \to [[a]]$ groups a list into sublists by some ordered key. The comprehension desugars to

$$\begin{array}{l}\mathit{groupWith}\ (\lambda(a,b) \to a)\ [(1,\texttt{'p'}),(2,\texttt{'q'}),(1,\texttt{'r'})] \gg\!= \lambda \mathit{abs} \to\\ \quad \textbf{case}\ (\mathit{map}\ \mathit{fst}\ \mathit{abs}, \mathit{map}\ \mathit{snd}\ \mathit{abs})\ \textbf{of}\ (a,b) \to \mathit{return}\ (\mathit{the}\ a, b)\end{array}$$

The full translation is given below, but in a nutshell, the input list of a, b pairs $[(1,\texttt{'p'}),(2,\texttt{'q'}),(1,\texttt{'r'})]$ is grouped according to their a component, and then for each group we return the common a component and the list b of corresponding characters. Note in particular the ingenious trick, that the **group** qualifier rebinds the previously bound variables $a :: Int$ and $b :: Char$ as lists $a :: [Int]$ and $b :: [Char]$—so of *different types*. (Suzuki et al. (2016) point out that this ingenious trick can backfire: it would be a runtime error to try to compute the b in the above query, because the b collection generally will not have all elements equal. They present a more sophisticated embedding of queries that turns this mistake into a type error.)

As with parallel comprehensions, grouping too generalizes to other monads, provided that they support an appropriate 'group' function. In Giorgidze et al.'s formulation (2011), this is again specified via a subclass of *Monad*:

$$\begin{array}{l}\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadGroup}\ m\ b\ \textbf{where} \\ \quad \mathit{mgroupWith} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ (m\ a)\end{array}$$

(so any given *mgroupWith* method is polymorphic in the a, but for fixed m and b). However, this type turns out to be unnecessarily restricted: there's no reason why the inner type constructor of the result, the type of each group, needs to coincide with the outer type constructor, the type of the collection of groups; in fact, all that the translation seems to require is

$$\mathit{mgroupWith} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ (n\ a)$$

with n some *Functor*. One could add n as another type class parameter; but the current GHC implementation (GHC 7.10 2015) instead dispenses with the *MonadGroup* type class altogether, and requires the grouping function to be explicitly specified. Let us call this variant "heterogeneous grouping". The translation is then:

$$\begin{array}{l}[\,e \mid q, \textbf{then group by } b \textbf{ using } f, r\,] \\ \quad = f\ (\lambda \mathsf{v}^q \rightarrow b)\ [\,\mathsf{v}^q \mid q\,] \ggg \lambda ys \rightarrow \\ \qquad \textbf{case } (\mathit{fmap}\ \mathsf{v}^q_1\ ys, ..., \mathit{fmap}\ \mathsf{v}^q_n\ ys)\ \textbf{of } \mathsf{v}^q \rightarrow [\,e \mid r\,]\end{array}$$

where, as before, v^q denotes the tuple of variables bound by qualifiers q, and in addition v^q_i projects out the ith component from this tuple. We make crucial use below of this extra generality. (GHC also provides a simpler variant omitting the "**by** b" clause, but we don't need it for this paper.)

It is not clear what laws we might expect of the grouping function f, at least partly because of its rather general type; moreover, because it is no longer associated with a type class, it is not clear where one ought morally to attach those laws. But for Giorgidze *et al.*'s homogeneous formulation, it is tempting to think of *mgroupWith* f as a partitioning function, which suggests

$$\mathit{mult} \cdot \mathit{mgroupWith}\ f = \mathit{id}$$

(which doesn't hold of the function *groupWith* on lists, since this reorders elements; that would be appropriate behaviour for bags, and Haskell's *groupBy* function, a pre-inverse of *concat*, more appropriate for lists). One might also want it to partition as finely as possible, so that subsequent grouping is redundant:

$$fmap_M\ (mgroupWith\ f) \cdot mgroupWith\ f = fmap_M\ return \cdot mgroupWith\ f$$

which would exclude size-based partitions, such as splitting a list of length n^2 into n lists of length n.

4.3 Relational Tables

The implementation that we will present of relational joins using comprehensions will depend crucially on the interplay between *two different monads*; that's why we need the extra generality of heterogeneous grouping described above. One of these is the *Bag* monad—that is, a ringad in which $\uplus$ is also commutative. The other is the *Table* monad, which to a first approximation (not yet accommodating aggregation—we turn to that question in Sect. 4.4) is simply a combination of the *Reader* and *Bag* monads:

$$\textbf{type}\ Table\ k\ v = Reader\ k\ (Bag\ v)$$

Readers are essentially just functions; in Haskell, there are a few type wrappers involved too, but we can hide these via the following isomorphism:

$$\begin{aligned} &apply &&:: Reader\ k\ v \to (k \to v) \\ &tabulate &&:: (k \to v) \to Reader\ k\ v \end{aligned}$$

Consider the canonical equijoin of two bags x, y by the two functions f, g, specified by

$$equijoin\ f\ g\ x\ y = [(a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b]$$

We can read this straightforwardly in the list monad; but as we have already observed, this leads to a very inefficient implementation. Instead, we want to be able to index the two input collections by their keys, and zip the two indices together. What else can we zip, apart from lists? *Readers* are the obvious instance:

$$\begin{aligned} &\textbf{instance}\ MonadZip\ (Reader\ k)\ \textbf{where} \\ &\quad mzip\ x\ y = tabulate\ (\lambda k \to (apply\ x\ k, apply\ y\ k)) \end{aligned}$$

Plain *Readers* aren't ringads; but *Tables* are, inheriting the ringad structure of *Bags*:

$$\begin{aligned} &\varnothing &&= tabulate\ (\lambda k \to \varnothing) \\ &x \uplus y &&= tabulate\ (\lambda k \to apply\ x\ k \uplus apply\ y\ k) \end{aligned}$$

Note that *Bags* alone won't do, because they don't support zip, only cartesian product.

For grouping, we will use a function

$$indexBy :: Eq\ k \Rightarrow (v \to k) \to Bag\ v \to Table\ k\ v$$

that partitions a bag of values by some key, collecting together subbags with a common key; with a careful choice of representation of the equality condition, this can be computed in linear time (Henglein and Larsen 2010). We will also use its postinverse

$$\mathit{flatten} :: \mathit{Table}\ k\ v \to \mathit{Bag}\ v$$

that flattens the partitioning. We need bags rather than lists, because *indexBy* reorders elements; the equation

$$\mathit{flatten} \cdot \mathit{indexBy}\ f = \mathit{id}$$

holds for bags, but would be hard to satisfy had we used lists instead.

With these ingredients, we can capture the equijoin using comprehensions:

$$\begin{aligned}&\mathit{equijoin}\ f\ g\ x\ y\\ &\quad = \mathit{flatten}\ [\mathit{cp}\ (a,b) \mid a \leftarrow x, \mathbf{then\ group\ by}\ f\ a\ \mathbf{using}\ \mathit{indexBy}\\ &\qquad\qquad\qquad\qquad\quad \mid b \leftarrow y, \mathbf{then\ group\ by}\ g\ b\ \mathbf{using}\ \mathit{indexBy}]\end{aligned}$$

which desugars to

$$\mathit{flatten}\ (\mathit{fmap}\ \mathit{cp}\ (\mathit{mzip}\ (\mathit{indexBy}\ f\ x)\ (\mathit{indexBy}\ g\ y)))$$

Informally, this indexes the two bags as tables by the key on which they will be joined, zips the two tables, computes small cartesian products for subbags with matching keys, then discards the index. In the common case that one of the two functions f, g extracts a primary key, the corresponding subbags will be singletons and so the cartesian products are trivial. Better still, one need not necessarily perform the *cp*s and *flatten* immediately; stopping with

$$\begin{aligned}&[(a,b) \mid a \leftarrow x, \mathbf{then\ group\ by}\ f\ a\ \mathbf{using}\ \mathit{indexBy}\\ &\qquad\quad \mid b \leftarrow y, \mathbf{then\ group\ by}\ g\ b\ \mathbf{using}\ \mathit{indexBy}]\end{aligned}$$

yields a table of pairs of subbags, and with care the cartesian products may never need actually to be expanded (Henglein and Larsen 2010).

4.4 Finite Maps

If we want to retain the ability to compute aggregations over tables—and we do, not least in order to define *flatten*—then we must restrict attention to *finite* maps, disallowing infinite ones. Then we can equip tables also with a mechanism to support traversal over the keys in the domain; so they should not simply be *Reader*s, they should also be paired with some traversal mechanism. (This is the 'refinement' mentioned earlier.) However, there is a catch: if we limit ourselves to finite maps, we can no longer define *return* for *Table*s or *Reader*s—*return a* yields an infinite map, when the key type itself is infinite.

Two possible solutions present themselves. One is to live without the *return* for *Table*, leaving what is sometimes called a *semi-monad* (but not obviously in the same sense as Fernandez et al. (2001)) or *non-unital monad*, that is, a monad with a *mult* but no unit. This seems to be feasible, while still retaining "semi-monad comprehensions"; we only really use *return* in the common base case $[\, e \mid \,]$ of comprehensions with an empty sequence of qualifiers, and this is mostly only for convenience of definition anyway (as noted already, it's not actually valid Haskell syntax). We instead have to provide separate base cases for each singleton sequence of qualifiers; we can't use a bare guard $[\, e \mid b \,]$, and we have to define comprehensions with guards and other qualifiers by

$$[\, e \mid b, q \,] = \mathbf{if}\ b\ \mathbf{then}\ [\, e \mid q \,]\ \mathbf{else}\ \varnothing$$

A second solution is to generalize from plain monads to what have variously been called *indexed monads* (Orchard et al. 2014), *parametric monads* (Katsumata 2014), or *graded monads* (Milius et al. 2015; Orchard and Yoshida 2016; Fujii et al. 2016); we use the latter term. In a graded monad $(M, \mathit{return}, \gg\!=)$ over a monoid $(I, \varepsilon, \otimes)$, the functor M has a type index drawn from the monoid I as well as the usual polymorphic type parameter; *return* yields a result at the unit index ε, and $\gg\!=$ combines indices using $\otimes$:

$$\begin{array}{ll} \mathit{return} & :: a \rightarrow M\ \varepsilon\ a \\ (\gg\!=) & :: M\ i\ a \rightarrow (a \rightarrow M\ j\ b) \rightarrow M\ (i \otimes j)\ b \end{array}$$

A familiar example is given by vectors. Vectors of a fixed length—say, vectors of length 3—form a monad, in which *return* replicates its argument and *mult* takes the diagonal of a square matrix. Vectors of different lengths, however, form not a simple monad but a graded monad, over the monoid $(\mathit{Nat}, 1, \times)$ of natural numbers with multiplication: *return* yields a singleton vector, and *mult* flattens an i-vector of j-vectors into a single $(i \times j)$-vector. Technically, a graded monad is no longer simply a monad; but it is still essentially a monad—it has the same kind of categorical structure (Fujii et al. 2016), supports the same operations, and can even still be used with monad comprehensions and **do** notation in Haskell, by using GHC's *RebindableSyntax* extension (Williams 2014). For our purposes, we want the monoid of finite sequences of finite types, using concatenation $+\!\!+$ and the empty sequence $\langle\rangle$, so that we can define

$$\begin{array}{ll} \mathit{return} & :: a \rightarrow \mathit{Table}\ \langle\rangle\ a \\ (\gg\!=) & :: \mathit{Table}\ k\ a \rightarrow (a \rightarrow \mathit{Table}\ k'\ b) \rightarrow \mathit{Table}\ (k +\!\!+ k')\ b \end{array}$$

Now *return* yields a singleton table, which is a finite map.

5 Conclusions

We have explored the list comprehension notation from languages like Haskell, its generalization to arbitrary monads, and the special case of collection monads or 'ringads' that support aggregation; and we have shown how to use GHC's parallel and grouping constructs for monad comprehensions to express relational join accurately and efficiently. All of these ingredients owe their genesis or their popularization to Phil Wadler's work; we have merely drawn them together.

Having said all that, writing this paper was still a voyage of discovery. The laws that one should require for *MonadPlus* and *MonadZero* are a subject of great debate; arguably, one expects different laws for backtracking search, for nondeterministic enumeration, and for collections of results—despite all conforming to the same interface. The matter is not settled definitely here (and perhaps the waters have been muddied a bit further—should we insist on associativity of $\uplus$, as some other authors do?).

Wadler's unpublished note (Wadler 1990) on ringads is not currently widely available; it is apparently not online anywhere, and even Phil himself does not have a copy (Wadler 2011)—for a long time I thought it had been lost entirely to history, despite being relatively widely cited. However, I am happy to report that Eerke Boiten had preserved a copy, and the document has now been secured.

Trinder's work with Wadler (Trinder and Wadler 1989; Trinder 1991) on using list comprehensions for database queries had quite an impact at the time, but they weren't the first to make the connection: Nikhil (1990), Poulouvassilis (1988), and Breuer (1989) at least had already done so.

Wadler and Peyton Jones (2007, Sect. 3.8) mention parallel list comprehensions, but don't connect them to grouping or to relational join; they write that "because of the generality of these new constructs, we wonder whether they might also constructively feed back into the design of new database programming languages". We hope that we have provided here more evidence that they could.

Acknowledgements. I am very grateful to Ralf Hinze, Nicolas Wu, Fritz Henglein, Tarmo Uustalu, and Oleg Kiselyov for their helpful discussions, to Eerke Boiten for finding a copy of Wadler's note (Wadler 1990), to Shin-ya Katsumata for sharing unpublished work (Fujii et al. 2016) on graded monads, and to the anonymous reviewers for constructive criticism. This work was partially supported by EPSRC grants *Unifying Theories of Generic Programming* (EP/J010995/1) and *A Theory of Least Change for Bidirectional Transformations* (EP/K020919/1).

References

Backhouse, R.: An exploration of the Bird-Meertens formalism. Technical report CS 8810, Department of Computer Science, Groningen University (1988). http://www.cs.nott.ac.uk/psarb2/papers/abstract.html#exploration

Boiten, E., Hoogendijk, P.: A database calculus based on strong monads and partial functions, March 1995. Submitted to DBPL

Boiten, E., Hoogendijk, P.: Nested collections and polytypism. Technical report 96/17, Eindhoven (1996)

Breuer, P.T.:. Applicative query languages. University Computing, the Universities and Colleges Information Systems Association of the UK (UCISA) Bulletin of Academic Computing and Information Systems (1989). https://www.academia.edu/2499641/Applicative_Query_Languages

Buneman, P., Navqi, S., Tannen, V., Wong, L.: Principles of programming with collections and complex object types. Theor. Comput. Sci. **149**(1), 3–48 (1995)

Cooper, E., Lindley, S., Yallop, J.: Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)

Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. **25**(4), 457–516 (2000). doi:10.1145/377674.377676

Fernandez, M., Simeon, J., Wadler, P.: A semi-monad for semi-structured data. In: International Conference on Database Theory, pp. 263–300 (2001)

Maarten, M.: Tupling and mutumorphisms. Squiggolist **1**(4), 81–82 (1990)

Fujii, S., Katsumata, S., Melliès, P.-A.: Towards a formal theory of graded monads. In: Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science. Springer (2016, to appear)

GHC 5.0. Glasgow Haskell Compiler users' guide, version 5.00, April 2001. https://downloads.haskell.org/~ghc/5.00/docs/set/book-users-guide.html

GHC 7.10. Glasgow Haskell Compiler users' guide, version 7.10, July 2015. https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/

Giorgidze, G., Grust, T., Schweinsberg, N., Weijers, J.: Bringing back monad comprehensions. In: Haskell Symposium, pp. 13–22 (2011)

Grust, T.: Comprehending queries. Ph.D. thesis, Universität Konstanz (1999)

Haskell 2010. Haskell 2010 language report, April 2010. https://www.haskell.org/onlinereport/haskell2010/

Henglein, F., Larsen, K.F.: Generic multiset programming with discrimination-based joins, symbolic cartesian products. High.-Order Symb. Comput. **23**(3), 337–370 (2010). doi:10.1007/s10990-011-9078-8

Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Principles of Programming Languages, pp. 633–645 (2014)

Kiselyov, O.: Laws of MonadPlus, January 2015. http://okmij.org/ftp/Computation/monads.html#monadplus

Luposchainsky, D.: MonadFail proposal, June 2015. https://github.com/quchen/articles/blob/master/monad_fail.md

Meijer, E.: The world according to LINQ. Commun. ACM **54**(10), 45–51 (2011)

Milius, S., Pattinson, D., Schröder, L.: Generic trace semantics and graded monads. In: Moss, L., Sobocinski, P., (eds.) 6th International Conference on Algebra and Coalgebra in Computer Science (CALCO 2015), pp. 251–266 (2015)

Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)

Nikhil, R.S.: The semantics of update in a functional database programming language. In: Bancilhon, F., Buneman, P., (eds.) Advances in Database Programming Languages (DBPL-1, 1987), pp. 403–421. ACM Press/Addison-Wesley (1990)

Orchard, D., Yoshida, N.: Effects as sessions, sessions as effects. In: Principles of Programming Languages (2016)

Orchard, D., Petricek, T., Mycroft, A.: The semantic marriage of monads, effects (2014). arXiv:1401.5391

Petricek, T.: Fun with parallel monad comprehensions. Monad Reader (18), (2011). https://themonadreader.wordpress.com/2011/07/05/issue-18/

Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice Hall, New Jersey (1987)

Plasmeijer, R., van Eekelen, M.: Concurrent clean language report (version 1.0). Technical report, University of Nijmegen (1995). ftp://ftp.science.ru.nl/pub/Clean/old/Clean10/doc/refman.ps.gz

Poulovassilis, A.: FDL: an integration of the functional data model and the functional computational model. In: British National Conference on Databases, pp. 215–236 (1988)

Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E.: Programming with Sets: An Introduction to SETL. Springer, New York (1986)

Schwartz, J.T.: On programming: an interim report on the SETL project. Technical report, Courant Institute of Mathematical Sciences, New York University, June 1975

Suciu, D.: Fixpoints and bounded fixpoints for complex objects. Technical report MS-CIS-93-32, University of Pennsylvania (1993a)

Suciu, D.: Queries on databases with user-defined functions. Technical report MS-CIS-93-62, University of Pennsylvania (1993b)

Suzuki, K., Kiselyov, O., Kameyama, Y.: Finally, safely-extensible and efficient language-integrated query. In: Partial Evaluation and Program Manipulation (2016)

Trinder, P.: Comprehensions: a query notation for DBPLs. In: Database Programming Languages (1991)

Trinder, P., Wadler, P.: Improving list comprehension database queries. In: TENCON 1989: Fourth IEEE Region 10 International Conference. IEEE (1989)

Uustalu, T.: A divertimento on MonadPlus and nondeterminism. J. Logical Algebr. Methods Program (2015, to appear). Special issue in honour of José Nuno Oliveira's 60th birthday. Extended abstract in HOpPE

Wadler, P.: Notes on monads and ringads. Internal Document, CS Department, University of Glasgow, September 1990

Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**, 461–493 (1992a)

Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Marktoberdorf Summer School on Program Design Calculi. NATO ASI Series F: Computer and Systems Sciences, vol. 118. Springer, Heidelberg (1992b). Also In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming. LNCS, vol. 925. Springer (1995)

Wadler, P.: Laws for monads with zero and plus. Post to Haskell Mailing List, May 1997

Wadler, P.: Monads and ringads. Personal communication, August 2011

Wadler, P., Jones, S.P.: Comprehensive comprehensions: comprehensions with 'order by' and 'group by'. In: Haskell Symposium, pp. 61–72 (2007)

Watt, D., Trinder, P.: Towards a theory of bulk types. FIDE technical report 91/26, Glasgow University, July 1991

Williams, T.: Map comprehensions, June 2014. http://www.timphilipwilliams.com/posts/2014-06-05-map-comprehensions.html

Wong, L.: Querying nested collections. Ph.D. thesis, University of Pennsylvania (1994)

Wong, L.: Kleisli, a functional query system. J. Funct. Program. **10**(1), 19–56 (2000)

Dragging Proofs Out of Pictures

Ralf Hinze[1(✉)] and Dan Marsden[2]

[1] Institute for Computing and Information Sciences (iCIS), Radboud University, Nijmegen, The Netherlands
ralf@cs.ru.nl

[2] Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, England, UK
daniel.marsden@cs.ox.ac.uk

Abstract. String diagrams provide category theory with a different and very distinctive visual flavour. We demonstrate that they are an effective tool for equational reasoning using a variety of examples evolving around the composition of monads. A deductive approach is followed, discovering necessary concepts as we go along. In particular, we show that the Yang–Baxter equation arises naturally when composing three monads. We are also concerned with the pragmatics of string diagrams. Diagrams are carefully arranged to provide geometric intution for the proof steps being followed. We introduce thick wires to distinguish composite functors and suggest a two-dimensional analogue of parenthesis to partition diagrams.

1 Introduction

Phil has a soft spot for presenting classic results in new clothes, providing a fresh or unifying view on old material. In this paper we follow his lead. The old results concern compositions of monads (Beck 1969), including iterated ones (Cheng 2007). The fresh view is afforded by string diagrams, which provide category theory with a different and very distinctive visual flavour.

String diagrams have been around since the work of Penrose (1971) and their use in a categorical setting was formalized in (Joyal and Street 1991). Despite the tremendous notational advantages they provide, their full potential for categorical calculations has yet to be realized. They are rarely mentioned in introductory texts and few research papers make explicit use of them, outside of certain sub-disciplines such as quantum computation. That's a shame—it's like not using a master's knife.

As we explore monad composition using string diagrams, we will "discover" Beck's distributive laws and the Yang-Baxter equation. These concepts can be identified visually from the structure of diagrams. To aid this discovery process, we stress the importance of good diagrammatic choices, introducing "fat" nodes and edges where needed for emphasis, and a two-dimensional analogue of parentheses for partitioning up our diagrams into components.

The rest of the paper is structured as follows. We briefly review monads and monad maps in Sect. 2, taking the opportunity to introduce string diagrams.

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 152–168, 2016.
DOI: 10.1007/978-3-319-30936-1_8

Sections 3 and 4 investigate Beck's work on composing monads, dressed in string-diagrammatic notation. The graphical exploration is continued in Sect. 5, which studies Cheng's work on iterated monad composition. Finally, Sect. 6 concludes. We assume a basic understanding of string diagrams, although we will briefly refresh some of the basics as we go along.

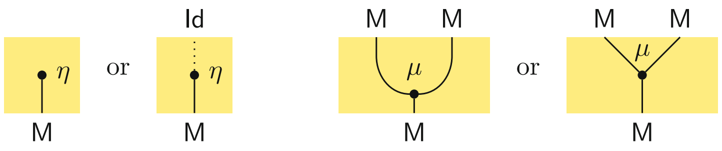
2 Monads and Monad Maps

Phil is Mr. Monad (Wadler 1992; King and Wadler 1993; Wadler 1990; Peyton Jones and Wadler 1993), therefore it seems appropriate to start the exploration of string diagrams with a discussion of monads. There are at least three different ways to introduce monads (Manes 1976). For our purposes, the so-called monoid form, which is based on two natural transformations, is most useful.

A monad, also historically referred to as a triple or a standard construction, consists of an endofunctor $\mathsf{M} : \mathcal{C} \to \mathcal{C}$ and natural transformations:

$$\eta : \mathsf{Id} \dot{\to} \mathsf{M} \quad \text{and} \quad \mu : \mathsf{M} \circ \mathsf{M} \dot{\to} \mathsf{M}$$

Drawn as a string diagram the unit η looks a tad like a lollipop whereas the multiplication μ resembles a tuning fork or a champagne cup:



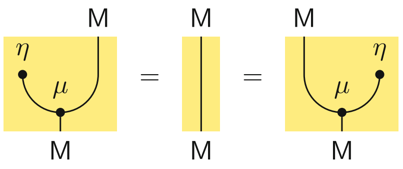
A string diagram is a planar graph. An edge in the graph corresponds to a functor, while a vertex (drawn as a small disc) corresponds to a natural transformation. The unit has no incoming functors and one outgoing functor. Identity functors are usually omitted from the diagrams. If required for emphasis, we hint at them using dotted edges. Likewise, the multiplication has two incoming and one outgoing functor. Edges can be arbitrary curves, as shown on the right, with the important restriction that they *must not have a horizontal tangent*.

The operations are required to satisfy unit and associativity axioms:

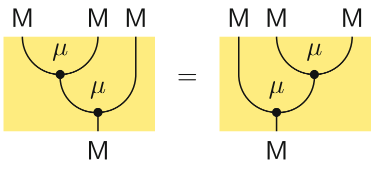
$$\mu \cdot (\eta \circ \mathsf{M}) = \mathsf{M} = \mu \cdot (\mathsf{M} \circ \eta) \tag{1a}$$

$$\mu \cdot (\mu \circ \mathsf{M}) = \mu \cdot (\mathsf{M} \circ \mu) \tag{1b}$$

where $\beta \cdot \alpha$ and $\beta \circ \alpha$ respectively denote vertical and horizontal composition of natural transformations. Written algebraically, these axioms are not particularly instructive. The equivalent string diagrams are much more revealing:



(2a)



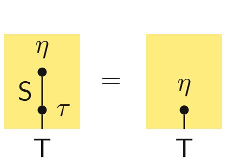
(2b)

Equations (2a) and (2b) make plain the idea that monads can be seen as monoids in an appropriate way, as hinted at by the suggestive names unit and multiplication for η and μ.

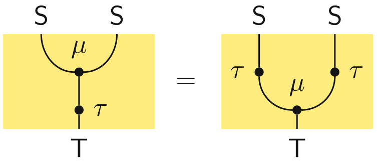
Observe that horizontal and vertical composition of natural transformations correspond to horizontal and vertical composition of diagrams in the obvious way. We stipulate that the flow is from right to left for horizontal composition $\beta \circ \alpha$, and from top to bottom for vertical composition $\beta \cdot \alpha$.

Whenever a new class of structures is introduced, the definition of structure-preserving maps follows hard on its heels. Monads are no exception, where these maps are variably called monad morphisms, monad transformers, monad functors, or, indeed, monad maps. We pick the latter term, because it is the shortest.

Given a pair of monads over the same base category, $\mathsf{S}, \mathsf{T} : \mathcal{C} \to \mathcal{C}$, a *monad map* is a natural transformation $\tau : \mathsf{S} \mathbin{\dot{\to}} \mathsf{T}$ that preserves unit and multiplication:



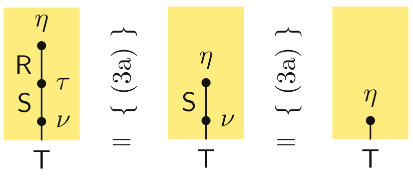
(3a)

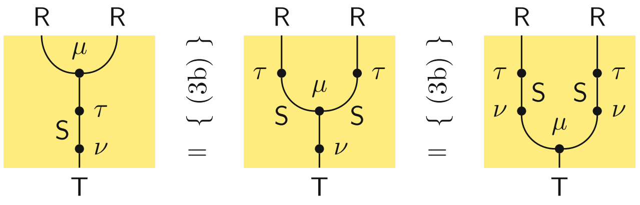


(3b)

Again the string notation is instructive. If we view monads as monoids, then τ is a monoid homomorphism, commuting appropriately with unit and multiplication.

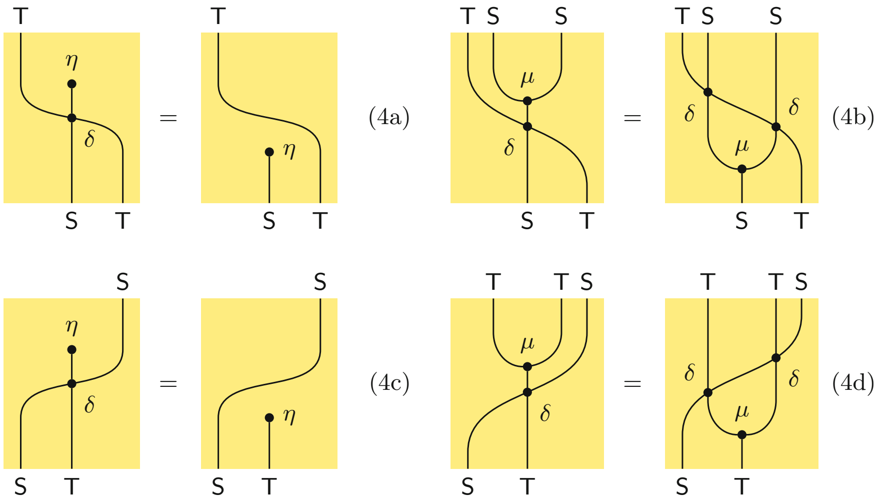
The identity $id : \mathsf{T} \mathbin{\dot{\to}} \mathsf{T}$ is clearly a monad map, and monad maps compose—we simply apply the coherence conditions twice, sliding the monad maps up, one after the other:





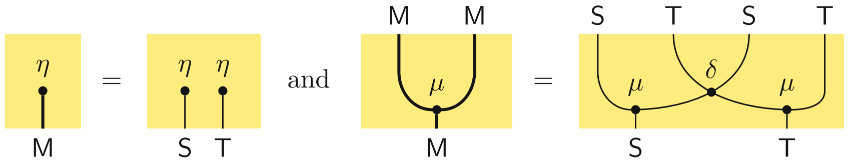
This entails that monads over some fixed category and monad maps between them form a category themselves.

Table 1. Properties of distributive laws.



3 Composing Monads

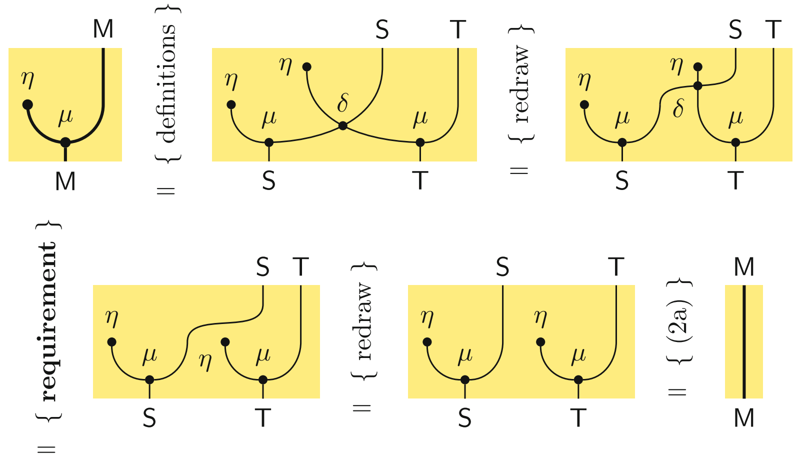
Suppose we wish to form the composite of monads S and T. To figure out the definitions of the unit and multiplication, we draw the obvious string diagrams:



The definition of the unit for $\mathsf{M} = \mathsf{S} \circ \mathsf{T}$ works out nicely. For the multiplication, we observe that we have to cross the S and T wires, requiring a natural transformation $\delta : \mathsf{T} \circ \mathsf{S} \stackrel{.}{\rightarrow} \mathsf{S} \circ \mathsf{T}$, commonly referred to as a *distributive law* (Beck 1969).

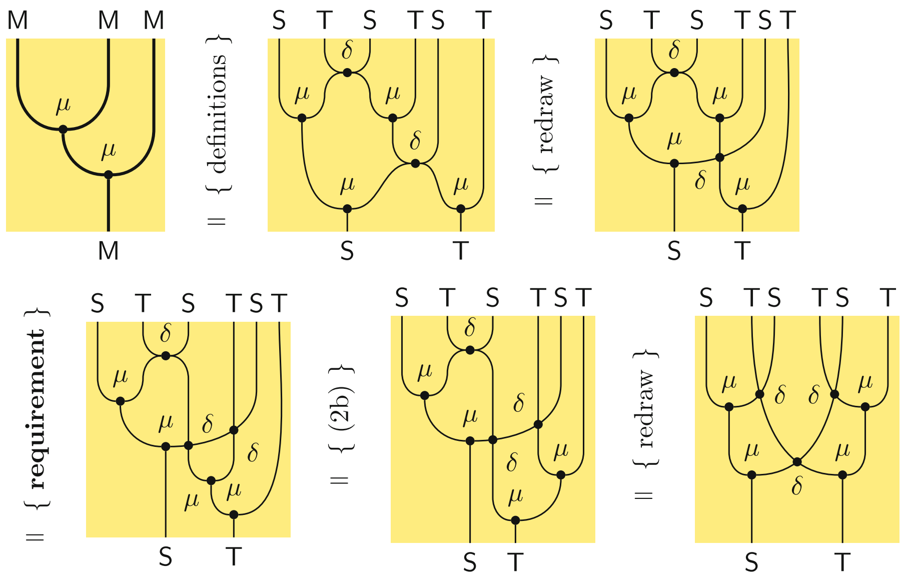
Note that we draw "fat wires" for the edges corresponding to the functor M simply to emphasise that is it a composite. As M is unfolded into $\mathsf{S} \circ \mathsf{T}$, each of the wires on the left-hand sides is expanded into a corresponding pair of wires on the right-hand sides.

Of course, not any old natural transformation δ will do. Let's try to prove the monad properties for the composite M and see what's needed. We begin by tackling the left unit law, the first equation of (2a):



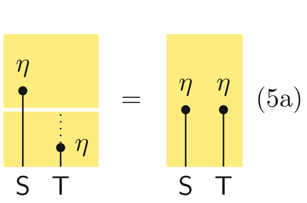
The goal is to invoke the left unit law for the constituent monads. In the third step, it therefore seems natural to "drag" the unit of T across the S edge. For reference, we record the requirement in Table 1. The right unit law, the second Eq. of (2a), follows symmetrically, leading to a symmetric requirement (4c).

In the proof above, as a first step after unfolding the definitions, we redraw the diagram to suggest a possible axiom to be applied. We take a similar approach in the proof of associativity:

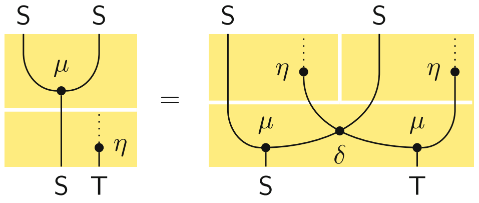


By assuming we can "drag" the multiplication of T across the S wire (4b), we have transformed our original diagram into a symmetrical position. Assuming a symmetric axiom (4d), we can then transform the right-hand side of the associativity law to the same position completing our proof. Quite pleasingly, by attempting the proof we have discovered the defining properties of distributive laws, see Table 1.

A composite monad $\mathsf{S}\circ\mathsf{T}$ is most useful if we are able to embed the base monads S and T into it. It turns out that the "one-sided" units $\mathsf{S}\circ\eta : \mathsf{S} \dot{\to} \mathsf{S}\circ\mathsf{T}$ and $\eta\circ\mathsf{T} : \mathsf{T} \dot{\to} \mathsf{S}\circ\mathsf{T}$ are suitable monad maps. For $\mathsf{S}\circ\eta$ we have to show:



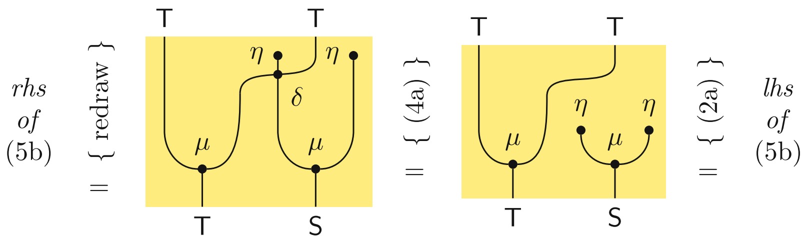
(5a)



(5b)

The diagrams are instances of (3a) and (3b). For emphasis, we have divided them into parts, to aid identifying the key components, the monad map and the monad operations. The white lines can be seen as the two-dimensional counterpart of parentheses: Eq. (5a) corresponds to $(\mathsf{S}\circ\eta)\cdot(\eta) = (\eta\circ\eta)$ and (5b) corresponds to $(\mathsf{S}\circ\eta)\cdot(\mu) = (\mu\circ\mu \cdot \mathsf{S}\circ\delta\circ\mathsf{T})\cdot(\mathsf{S}\circ\eta)\circ(\mathsf{S}\circ\eta)$.

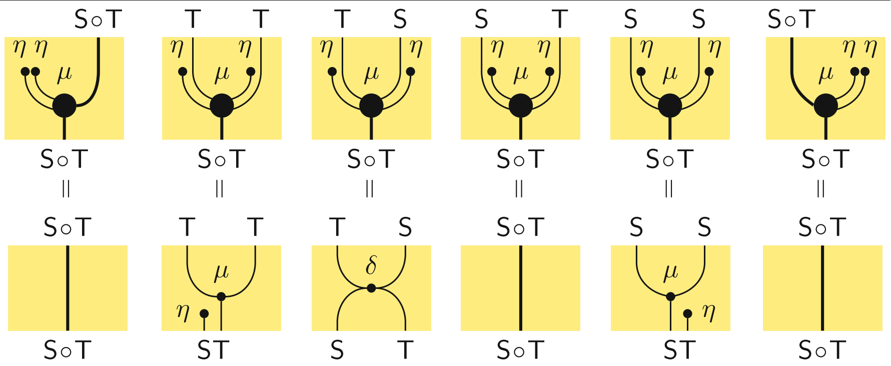
Now, the proof that $\mathsf{S}\circ\eta$ preserves the unit (5b) is void; the proof that it preserves multiplication (5b) essentially uses the fact that the distributive law preserves the unit (4a):



A symmetric argument establishes that $\eta\circ\mathsf{T}$ is a monad map.

4 Compatible Compositions

An interesting question is whether every monad of the form $(\mathsf{S}\circ\mathsf{T}, \eta\circ\eta, \mu)$ arises via a distributive law. In other words, can we reconstruct a distributive law from the data? Now, from a given composite multiplication we can extract a candidate law by placing a lollipop on its left- and rightmost arms. Exploring this space a bit more, we observe that there are $\binom{4}{2} = 6$ ways to place two lollipops on four arms—it is useful to investigate all of them:



left unit S∘T	$\eta\circ$T monad map (6a)	extract distr. law (6b)	*middle unitary law* (6c)	S$\circ\eta$ monad map (6d)	right unit S∘T

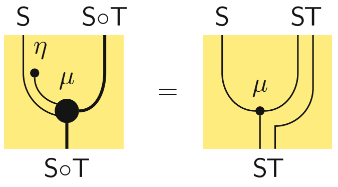
The diagrams now feature both thick composite wires, and their thinner component wires when we need to access the individual threads. The composite multiplication is depicted by a fat vertex, to provide room for plugging in parallel wires. Hopefully, these twists will improve the readability of the forthcoming diagrams.

All of the equalities with the notable exception of the fourth one, the so-called *middle unitary law* (6c), have a familiar reading. If the composite monad is constructed using a distributive law, then these equations are indeed satisfied. Perhaps surprisingly, the converse is also true: a composite monad that satisfies all of these is given by a distributive law! To introduce some terminology: the composition S∘T is called *compatible with* S *and* T if

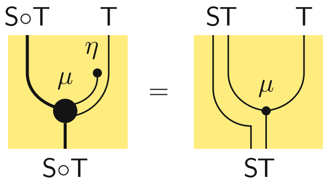
- S∘T is a monad with unit $\eta = \eta\circ\eta$;
- the one-sided units S$\circ\eta$: S $\dot{\to}$ S∘T $\dot{\leftarrow}$ T : $\eta\circ$T are monad maps; and
- the middle unitary law (6c) holds.

Distributive laws and compatible compositions are in one-to-one correspondence. We have already noted that a composite monad given by a distributive law is compatible. For the other direction we need to work a bit harder:

As a first warm-up, we relate the multiplication of the composite monad to the multiplications of the base monads, generalizing (6d) and (6a):



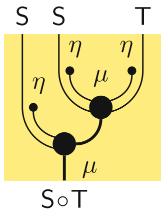
(7a)



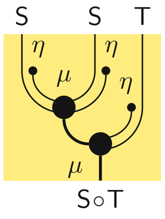
(7b)

The proof of (7a) makes essential use of the middle unitary law (6c):

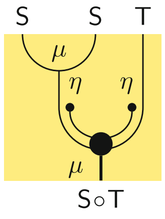
lhs of (7a) $= \{ \text{(6c)} \}$



$= \{ \text{(2b)} \}$



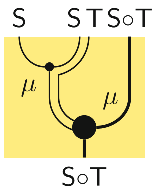
$= \{ \text{(6d)} \}$



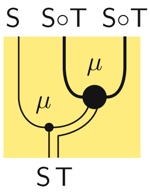
$= \{ \text{(6c)} \}$ *rhs of* (7a)

The middle unitary law is first used to create a nested composite multiplication. After reassociating the multiplication, we can then apply the monad map axiom (6d), which (7a) generalizes. A second application of the middle unitary law drives the proof home.

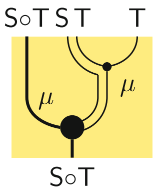
As a second warm-up, we establish *pseudo-associative laws*, which allow us to reassociate a nesting of a composite fork and a base fork:



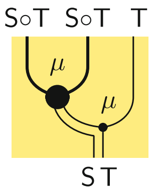
$=$



(8a)



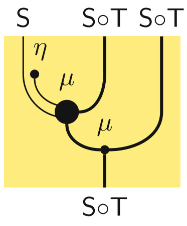
$=$



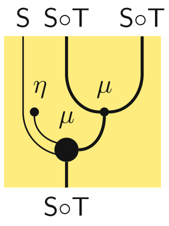
(8b)

The proof of left pseudo-associativity (8a) combines the previous auxiliary result (7a) and associativity (2b):

lhs of (8a) $= \{ \text{(7a)} \}$



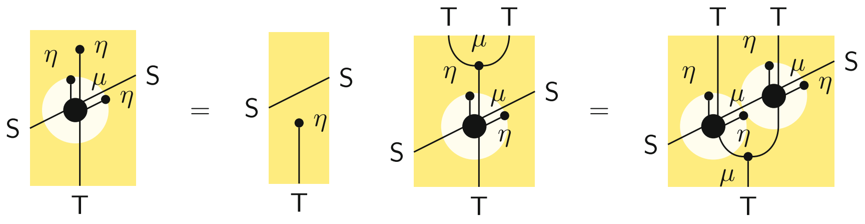
$= \{ \text{(2b)} \}$



$= \{ \text{(7a)} \}$ *rhs of* (8a)

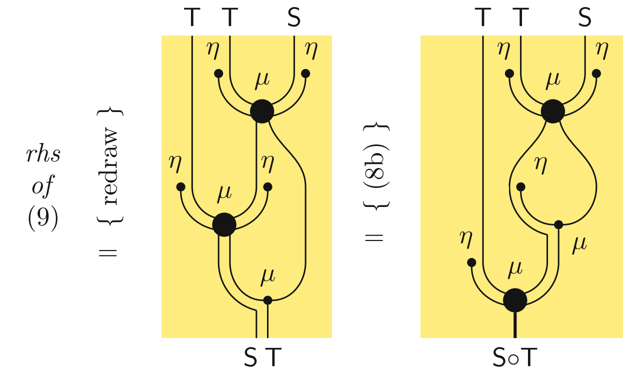
The "proof strategy" is similar to the previous one: we work towards a situation where associativity for the composite multiplication can be invoked.

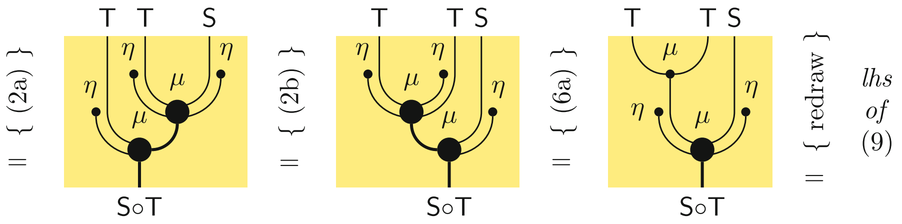
We are now in a position to prove the first main result: the natural transformation defined by (6b) is a distributive law. We content ourselves with proving (4a) and (4b), as the remaining properties, (4c) and (4d), follow dually. Instantiating the equations to the natural transformation at hand (highlighted below for clarity) we have the following proof obligations:



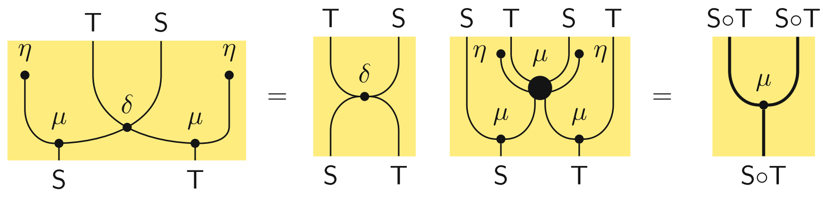
(9)

The first property follows immediately from the unit law for S∘T. For the second property we reason:





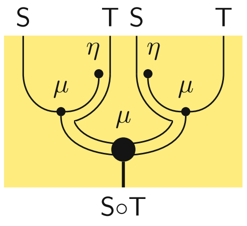
Finally, for the round trip—distributive laws and compatible monads are in one-to-one correspondence—we have to show:



(10)

The first equation is an immediate consequence of the unit laws (2a); the proof of the second equation is a one-liner:

lhs of (10) $= \{ \text{(8a) and (8b)} \}$



$= \{ \text{(2a)} \}$ *rhs of* (10)

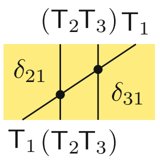
In the first step we apply the two pseudo-associative laws (8a) and (8b) in parallel. This moves the forks upwards, where they meet their lollipops, only to dissolve collectively into thin air.

5 Iterated Distributive Laws

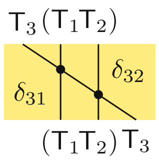
Using a distributive law we can form the composite of two monads. Does the construction also work for three or more monads over the same category? The answer due to Cheng (2007) is an enthusiastic 'yes, but ...". So let's see what's involved. To form the composite of T_1, T_2, and T_3, using Beck's result, we either need a distributive law that distributes the composite $\mathsf{T}_2\circ\mathsf{T}_3$ over T_1, or a law that distributes T_3 over the composite $\mathsf{T}_1\circ\mathsf{T}_2$. Assuming the existence of laws

$$\delta_{31} : \mathsf{T}_3\circ\mathsf{T}_1 \stackrel{.}{\to} \mathsf{T}_1\circ\mathsf{T}_3 \qquad \delta_{21} : \mathsf{T}_2\circ\mathsf{T}_1 \stackrel{.}{\to} \mathsf{T}_1\circ\mathsf{T}_2 \qquad \delta_{32} : \mathsf{T}_3\circ\mathsf{T}_2 \stackrel{.}{\to} \mathsf{T}_2\circ\mathsf{T}_3$$

suitable candidates are:

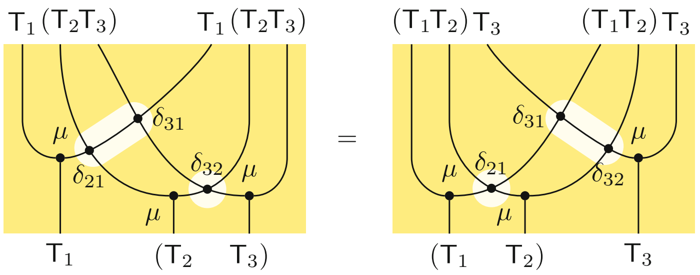
$\delta \;=$


(11a)

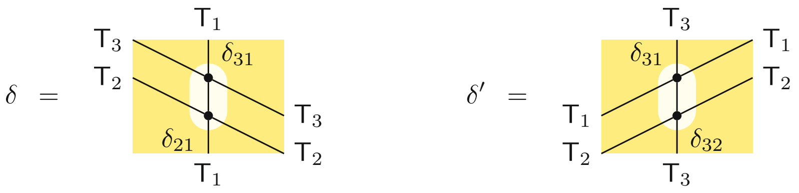
$\delta' \;=$


(11b)

The candidates δ and δ' are formed by joining two of the given laws. Unfortunately, distributive laws are not closed under this form of composition (simply because monads do not compose in general), so we have to actually verify the coherence conditions. However, *if* the candidates satisfy the requirements, then the resulting monads are the same, regardless of the bracketing. Clearly, the underlying functors, $(\mathsf{T}_1\circ\mathsf{T}_2)\circ\mathsf{T}_3$ and $\mathsf{T}_1\circ(\mathsf{T}_2\circ\mathsf{T}_3)$, and the units, $(\eta_1\circ\eta_2)\circ\eta_3$ and $\eta_1\circ(\eta_2\circ\eta_3)$, are identical; to see that the multiplications do not depend on the bracketing, we construct the corresponding string diagrams:

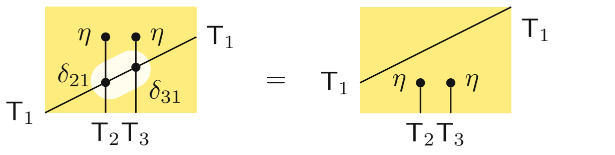


Returning to the coherence requirements of the composite laws, the candidate law δ (11a) satisfies (4a) and (4b). This can be seen most easily if we redraw the diagrams:

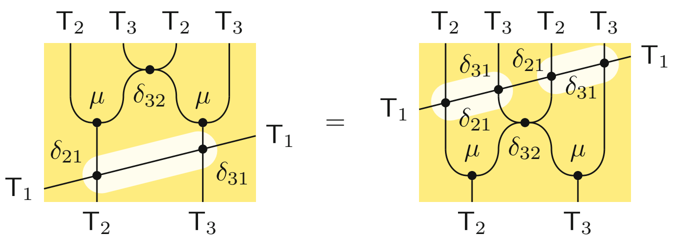


To show the unit axiom (4a), we drag the unit of T_1 twice, first across the functor T_3 and then a second time across the functor T_2. We proceed in an analogous fashion to establish the multiplication axiom (4b), dragging the fork twice. Thus, it remains to show that δ satisfies the dual axioms, (4c) and (4d). Dually, the candidate law δ' (11b) satisfies the latter, but not in general the former axioms. See redrawing on the right above.

We concentrate on the left candidate; a dual argument establishes the requirements for the right one. We need to prove that δ preserves unit (4c) and multiplication (4d) of the composite monad $T_2 \circ T_3$:



(12a)

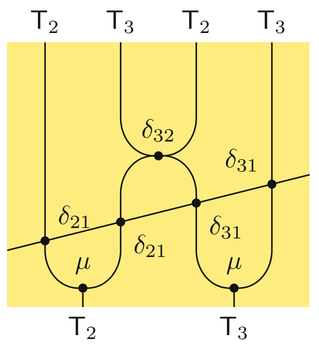


(12b)

The first equation follows from the fact that the constituent laws δ_{21} and δ_{31} preserve the units (4c); for the second equation we reason:

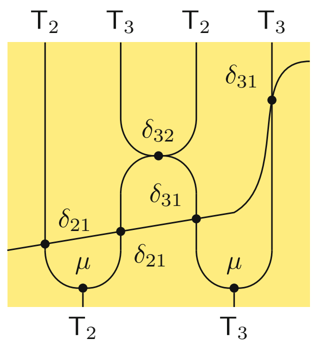
lhs of (12b)

$= \{ \text{(4d)} \}$



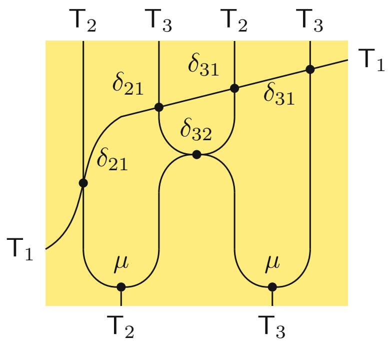
T_2 T_3 T_2 T_3
δ_{32}
δ_{31}
δ_{21}
δ_{21}
δ_{31}
μ
μ
T_2 T_3

$= \{ \text{redraw, see below} \}$



T_2 T_3 T_2 T_3
δ_{31}
δ_{32}
δ_{31}
δ_{21}
δ_{21}
μ
μ
T_2 T_3

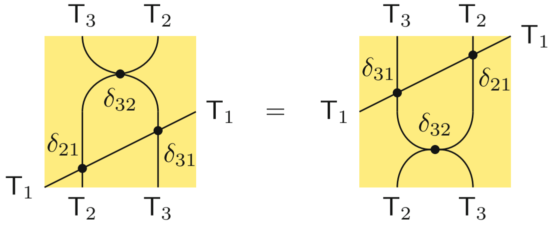
$= \{ \text{Yang–Baxter (13)} \}$



$= \{ \text{redraw} \}$

rhs of (12b)

The second and the fourth step are redrawings employing the topological freedom inherent in the notation. The central step is the third one where we drag the distributive law δ_{32} across the diagonal line, shown below in detail:



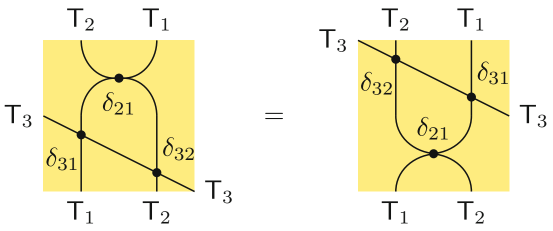
(13)

We have discovered the so-called *Yang–Baxter equation*, which relates three distributive laws. The left- and the right-hand side are related by a rotation of 180°. This entails that the order of δ_{21} and δ_{31} is reversed as we drag δ_{32} down.

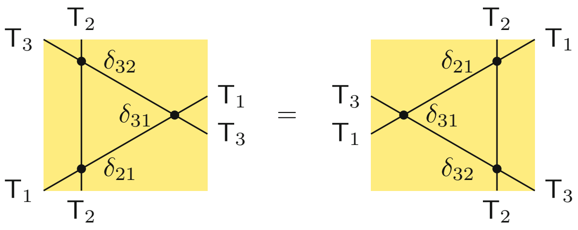
Up to now we have added redrawing operations for clarity, to emphasize the moves being made. This is one of the exceptional cases where such a move is actually necessary. Consider the second step. If we hadn't lifted the right-most distributive law δ_{31}, then we wouldn't be able to execute the subsequent "dragging" step, as this step would cause the last edge of the diagonal to have a horizontal tangent.[1] (The second redrawing is then a cosmetic one.)

[1] String diagrams that are equivalent up to *planar isotopy* denote the same natural transformation. A planar isotopy is a continuous deformation of a plane diagram that preserves cusps, crossings, and the property of having no horizontal tangents (Joyal and Street 1991).

For the right candidate δ' (11b) we obtain vertically reflected diagrams, which are, however, semantically equivalent—the diagram on the left (right) below denotes the same natural transformation as the diagram on the right (left) above.

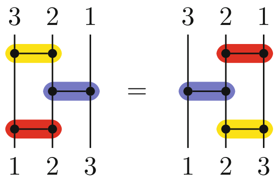


A more neutral depiction places the laws on the corners of equilateral triangles:



The diagrams above suggest an alternative reading of Yang–Baxter: the condition allows us to exchange the laws δ_{21} and δ_{32}. Note that T_2 is the vertical edge; T_3 travels from left to right; and T_1 in the opposite direction. The laws that swap adjacent indices, δ_{21} and δ_{32}, are either on the top or at the bottom; the third law δ_{31} is bound to be in between.

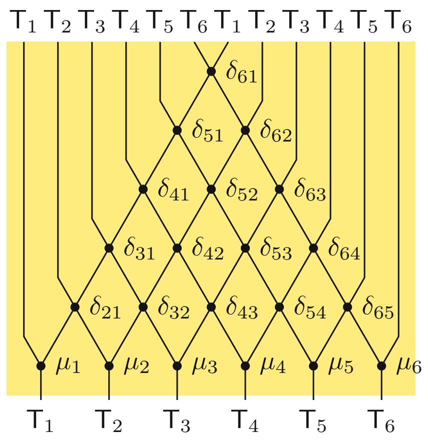
As an amusing aside, note that both composites "sort" the monads: $\mathsf{T}_3 \circ \mathsf{T}_2 \circ \mathsf{T}_1$ is transmogrified to $\mathsf{T}_1 \circ \mathsf{T}_2 \circ \mathsf{T}_3$. And indeed, the two arrangements correspond to the two ways of constructing an *odd-even transposition network* (Knuth 1998) for three inputs:

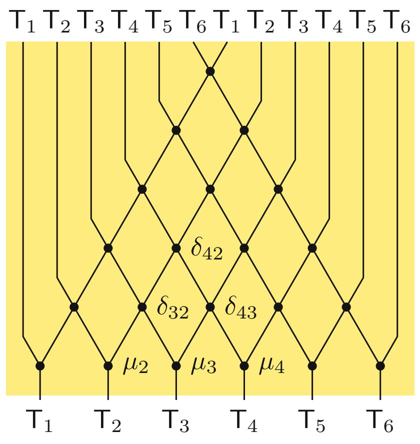


A transposition network is a special comparator network where comparators only connect adjacent wires. Such a network is a sorting network if and only if it sorts the decreasing sequence $n, n-1, \ldots, 2, 1$, a remarkable property that links odd-even transposition networks to Yang–Baxter.

As a short intermediate summary, we can compose three monads if the three distributive laws for the constituent monads satisfy Yang–Baxter (13).

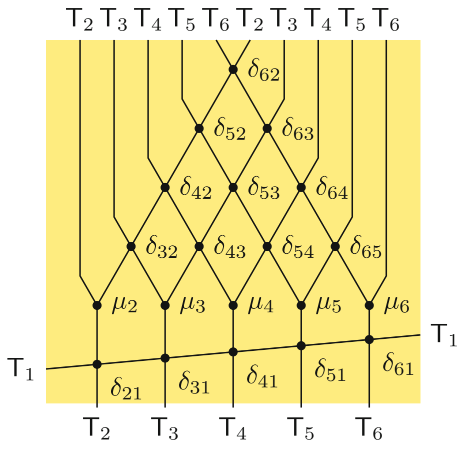
Now, let's become more ambituous and consider compositions of n monads: $\mathsf{T}_1, \ldots, \mathsf{T}_n$. It is instructive to visualize the multiplication of the composite monad for some small n, say, 6 first (ignore the diagram on the right):



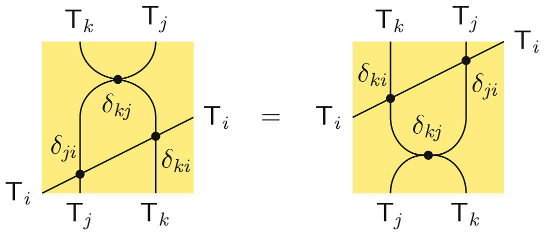


The left diagram has a very regular structure, consisting of two key parts. Firstly, an "interchanging network", the triangle of δs that pairwise swaps the positions of the functors, sending $\mathsf{T}_1\circ\cdots\circ\mathsf{T}_6\circ\mathsf{T}_1\circ\cdots\circ\mathsf{T}_6$ to $\mathsf{T}_1\circ\mathsf{T}_1\circ\cdots\circ\mathsf{T}_6\circ\mathsf{T}_6$. Secondly, a "merging network", the bottom row of μs that combine adjacent matching pairs of wires, taking the reorganized sequence to the composite $\mathsf{T}_1\circ\cdots\circ\mathsf{T}_6$. The triangle consists of no less than 15 distributive laws. In general, we need $\delta_{ji} : \mathsf{T}_j\circ\mathsf{T}_i \dot{\to} \mathsf{T}_i\circ\mathsf{T}_j$ for $n \geq j > i \geq 1$. Perhaps unsurprisingly, the total number of required laws is given by the triangular numbers: $\binom{n}{2} = \frac{1}{2}(n-1)n$. The network above contains as sub-diagrams the multiplications for all "sub-composite monads", i.e. compositions of contiguous subsequences of $\mathsf{T}_1, \ldots, \mathsf{T}_6$. The diagram on the right above provides an example. The sub-diagram of labelled vertices is the multiplication of $\mathsf{T}_2\circ\mathsf{T}_3\circ\mathsf{T}_4$. In fact, each distributive law serves as the root of a sub-diagram: δ_{ji} spans the multiplication of the composite monad $\mathsf{T}_i\circ\cdots\circ\mathsf{T}_j$.

Of course, the distributive laws have to go together—they must satisfy suitable instances of Yang–Baxter (13). To see which instances are needed, it is again useful to visualize the proof for some small n. Continuing our running example, we have to show that we can distribute $\mathsf{T}_2\circ\cdots\circ\mathsf{T}_6$ over T_1 i.e. that the composite distributive law respects the multiplication of $\mathsf{T}_2\circ\cdots\circ\mathsf{T}_6$.



The bottom diagonal line amounts to the composite law, the horizontal composition of $\delta_{21}, \ldots, \delta_{61}$. The proof involves moving the diagonal upwards, first across the multiplications and then across the triangle of distributive laws. The first move is justified as the δ_{ji} are distributive laws which preserve multiplications (4b). The other moves require instances of Yang–Baxter, in fact, a triangular number of instances: δ_{ji}, δ_{kj}, and δ_{ki} for $i = 1$ and $6 \geq k > j \geq 2$. Analogous proofs are needed to show that we can distribute $\mathsf{T}_3 \circ \cdots \circ \mathsf{T}_6$ over T_2, $\mathsf{T}_4 \circ \cdots \circ \mathsf{T}_6$ over T_3, and $\mathsf{T}_5 \circ \mathsf{T}_6$ over T_4. In general, we require that δ_{ji}, δ_{kj}, and δ_{ki} satisfy the Yang–Baxter equation for $n \geq k > j > i \geq 1$:



The total number of proof obligations is therefore given by the sum of the triangular numbers, the so-called tetrahedral numbers: $\binom{n}{3} = \frac{1}{6}(n-2)(n-1)n$.

6 Conclusion

We have explored various aspects of composing monads using string diagrams. Although not emphasized in the paper, but see (Marsden 2014) and (Hinze and Marsden 2015), this notation silently handles trivial bookkeeping steps such associativity, identity laws, functoriality, and naturality, allowing us to focus on the essentials of our proofs. In order to highlight key proof steps, we enthusiastically exploited the visual nature of the notation, and the topological freedom it affords. Many diagrams were drawn so that proof steps have intuitive interpretations such as sliding or dragging components within them. We have also addressed the pragmatics of string diagrams, adding new bells and whistles to the graphical notation. Fat wires were introduced to distinguish composite functors within our diagrams, and large vertices provided space to clearly insert parallel wires where required. We also introduced a two-dimensional analogue of parentheses by partitioning diagrams into regions using white lines.

Traditional mathematics follows a rhythm of "definition, theorem, proof, definition, theorem, proof". In this paper we adopted a different approach. We typically began with an equation we wished to prove. We then optimistically attempted a proof, and our graphical notation allowed us to *see* the missing properties that were required to complete it. These properties were then incorporated into our definitions. In this way, we rediscovered Beck's distributive laws for composing monads, and the axioms they are required to satisfy. We also identified the significance of the Yang-Baxter equation in iterating composition of monads, in a straightforward visual manner.

As we have been working within a single fixed category, our diagrams have been "monochrome", with all regions the same colour. In fact string diagrams can be extended to support coloured regions, so we can work with multiple categories within our proofs. This extension also allows us to work with ordinary objects and arrows, and in fact we can conduct a large part of elementary category theory within this setting. Detailed discussion of these colourful diagrams can be found in the recent papers (Marsden 2014) and (Hinze and Marsden 2015).

String diagrams are not the only two-dimensional notation for categories, functors, and natural transformations. Their Poincaré dual, so-called pasting schemes, seem to be in wider use. When the first author presented string diagrams at a recent IFIP WG 2.8 meeting (June 2015), Phil initially wasn't very convinced of their advantages. Overnight he worked out the properties using the more traditional pasting schemes only to discover that they don't provide any geometric intuition at all. Hopefully the present paper fully convinces you of the virtues of string diagrams, Phil.

Acknowledgements. The authors are very grateful to Tarmo Uustalu for bringing compatible compositions and iterated distributive laws to their attention. We would also like to thank the two anonymous reviewers for their helpful comments and feedback. This work has been partially funded by EPSRC grant numbers EP/J010995/1, EP/I03808X/1 and AFSOR grant "Algorithmic and Logical Aspects when Composing Meanings". The main part of the work was done while the first author was affiliated with the University of Oxford.

References

Beck, J.: Distributive laws. In: Appelgate, H., et al. (eds.) Seminar on Triples and Categorical Homotopy Theory. Lecture Notes in Mathematics, vol. 80, pp. 119–140. Springer, Heidelberg (1969)

Cheng, E.: Iterated distributive laws. ArXiv e-prints, October 2007

Hinze, R., Marsden, D.: Equational reasoning with lollipops, forks, cups, caps, snakes, and speedometers. José Oliveira Festschrift (2015, to appear)

Joyal, A., Street, R.: Geometry of tensor calculus I. Adv. Math. **88**, 55–113 (1991)

King, D.J., Wadler, P.: Combining monads. In: Launchbury, J., Sansom, P. (eds.) Proceedings of the 1992 Glasgow Workshop on Functional Programming. Workshops in Computing, pp. 134–143. Springer, Berlin/Heidelberg (1993)

Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, vol. 3, 2nd edn. Addison-Wesley Publishing Company, Reading (1998)

Manes, E.G.: Algebraic Theories. Springer, Heidelberg (1976)

Marsden, D.: Category theory using string diagrams (2014). arXiv preprint arXiv:1401.7220

Penrose, R.: Applications of negative dimensional tensors. In: Welsh, D.J.A. (ed.) Combinatorial Mathematics and its Applications, pp. 221–244. Academic Press, New York (1971)

Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, pp. 71–84, Charleston, South Carolina, January 1993

Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, pp. 61–78. ACM Press, June 1990

Wadler, P.: The essence of functional programming. In: Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico, pp. 1–14, January 1992

Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane

John Hughes[1,2](✉)

[1] Chalmers University, Göthenburg, Sweden
`rjmh@chalmers.se`
[2] Quviq AB, Göthenburg, Sweden

Abstract. This is not a typical scientific paper. It does not present a new method, with careful experiments to evaluate it, and detailed references to related work. Rather, it recounts some of my experiences over the last 15 years, working with QuickCheck, and its purpose is as much to entertain as to inform.

QuickCheck is a random testing tool that Koen Claessen and I invented, which has since become the testing tool of choice in the Haskell community. In 2006 I co-founded Quviq, to develop and market an Erlang version, which we have since applied for a wide variety of customers, encountering many fascinating testing problems as a result.

This paper introduces Quviq QuickCheck, and in particular the extensions made for testing stateful code, via a toy example in C. It goes on to describe the largest QuickCheck project to date, which developed acceptance tests for AUTOSAR C code on behalf of Volvo Cars. Finally it explains a race detection method that nailed a notorious bug plaguing Klarna, northern Europe's market leader in invoicing systems for e-commerce. Together, these examples give a reasonable overview of the way QuickCheck has been used in industrial practice.

1 Introduction

Here's an interesting little experiment. Try asking a room full of software developers, "who *really, really* loves testing?" Very, very few will raise their hands. For whatever reason, most developers see testing as more of a chore than a pleasure; few go to work in the morning raring to write some test cases. And yet, testing is a vital part of software development! Why should it be so unpopular?

To understand why, imagine writing a suite of unit tests for software with, say, n different features. Probably you will write 3–4 test cases per feature. This is perfectly manageable—it's a linear amount of work. But, we all know you will not find all of your bugs that way, because some bugs can only be triggered by a pair of features interacting. Now, you *could* go on to write test cases for every pair of features—but this is a quadratic amount of work, which is much less appealing. And even if you do so, you will still not find all of your bugs—some bugs only appear when *three* features interact! Testing for all of these would involve $O(n^3)$ work, which is starting to sound very unappealing indeed—and this is before we

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 169–186, 2016.
DOI: 10.1007/978-3-319-30936-1_9

even start to consider race conditions, which by definition involve at least two features interacting, and even worse, only manifest themselves occasionally even in test cases that can provoke them in principle!

This is the fundamental problem with testing—you can never be "done". No wonder it's not so popular. What is the answer to this conundrum? Simply this:

DON'T WRITE TESTS!

But of course, we can't just deliver untested code to users. So the message is actually more nuanced: don't *write* tests—*generate* them!

This is how I have spent much of my time in recent years, working with a test case generator called QuickCheck. QuickCheck was first developed in Haskell by Koen Claessen and myself (Claessen and Hughes 2000), and has become the testing tool of choice among Haskell developers. The idea has been widely emulated—Wikipedia now lists no fewer than 35 reimplementations of the basic idea, for languages ranging from C to Swift. Thomas Arts and I founded a company, Quviq, to market an Erlang version in 2006 (Hughes 2007), and since then we have made many extensions, and had great fun finding bugs for Ericsson (Arts et al. 2006), Volvo Cars (Arts et al. 2015), and many others. In this paper I will sketch what Quviq QuickCheck does, and tell a few stories of its applications over the years.

```
#include <stdlib.h>

typedef struct queue
{ int *buf;
  int inp, outp, size;
} Queue;

Queue *new(int n)
{ int *buff =
    malloc(n*sizeof(int));
  Queue q = {buff,0,0,n};
  Queue *qptr =
    malloc(sizeof(Queue));
  *qptr = q;
  return qptr; }
```

```
void put(Queue *q, int n)
{ q -> buf[q -> inp] = n;
  q -> inp = (q -> inp + 1)
               % q -> size; }

int get(Queue *q)
{ int ans = q -> buf[q -> outp];
  q -> outp = (q -> outp + 1)
                % q -> size;
  return ans; }

int size(Queue *q)
{ return (q->inp - q->outp)
           % q -> size; }
```

Fig. 1. A queue ADT, implemented as a circular buffer in C.

2 A Simple Example

Let us begin with a simple example: a queue, implemented as a circular buffer, in C. The code appears in Fig. 1. It declares a struct to represent queues, containing a pointer to a buffer area holding integers, the indices in the buffer where the next value should be inserted respectively removed, and the total size of the buffer. The `new` function, which creates a queue of size `n`, allocates memory for the struct and the buffer, initializes the struct, and returns a pointer to it. `get` and `put` read and write the buffer respectively at the appropriate index, incrementing the index afterwards modulo the size. Finally, `size` returns the number of elements currently in the queue, by taking the difference of the input and the output indices, modulo the size. The code is straightforward, and obviously correct—it's just a simple example to give us something to test.

Quviq QuickCheck provides a mechanism for testing C code directly from Erlang. In the Erlang shell, we can call

```
1> eqc_c:start(q).
```

which compiles `q.c` (containing the code from Fig. 1), and makes the functions in it callable from Erlang. This is done by parsing the C code to extract the types and functions it contains, generating a server that calls the C functions on demand, generating client code in Erlang that passes a function and arguments to the server and collects the result, and then running the server in a separate OS process so that buggy C code cannot interfere with the OS process running QuickCheck. The result is a safe test environment for C functions that might behave arbitrarily badly.

We can now perform simple tests to verify that the code is working as it should:

```
2> Q = q:new(5).             | 4> q:put(Q,2).   | 6> q:get(Q).
{ptr,"Queue",6696712}        | ok               | 1
3> q:put(Q,1).               | 5> q:size(Q).    | 7> q:get(Q).
ok                           | 2                | 2
```

We create a queue with space for 5 elements, binding the pointer returned to `Q`. Then we put 1 and 2 into the queue, test `size`, and take the elements out of the queue again. All the results are as expected. We can even continue the test:

```
8> q:get(Q).     | 10> q:get(Q).  |
100663302        | 6696824        | 12> q:get(Q).
9> q:get(Q).     | 11> q:get(Q).  | 2
27452            | 1              |
```

which returns the contents of uninitialized memory, as expected, until we reach the values 1 and 2 again. We really are running C code, which is just as unsafe as ever, and the queue implementation does not—and is not intended to—perform any error checking. If you abuse it by removing elements from an empty queue, you will get exactly what you deserve.

Testing with QuickCheck. Like much of the code Quviq works with, this example is stateful. We test stateful systems by generating *sequences* of calls to the API under test, just like the test cases that developers write by hand. But we also *model* the state of the system abstractly, and define *model state transitions* for each operation in the API. Using these state transition functions,

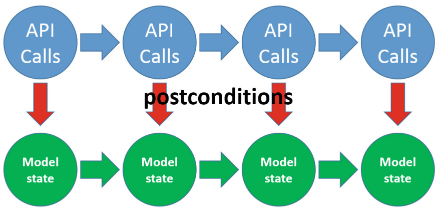


Fig. 2. Adjudging a test based on a state machine model.

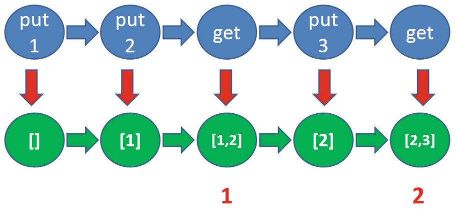


Fig. 3. A sample test case for the queue.

we compute the model state at every point in a test. We define *postconditions* for each API call, relating the *actual* result of the call to the *model* state—see Fig. 2. A test passes if all postconditions hold. While the code we test can be in any programming language (such as C), the *model*—the state transition functions, postconditions, *etc.*—is defined in Erlang.

In this case, we might model the state of the queue by a *list* of the integers that should be in it; with this model, a sample test case might appear as in Fig. 3. Here we start with an empty queue (modelled by the empty list `[]`), each `put` operation appends an element to the model state, each `get` operation removes an element from the state, and `get`'s postcondition checks that the actual result (in red) was the first element of the model state before the call.

We will not present all the code of the model here, but to give its flavour, we present the specification of `get`:

```
get_pre(S) -> S#state.ptr /= undefined andalso
               S#state.contents /= [].

get_next(S,_Value,_Args) ->
              S#state{contents=tl(S#state.contents)}.

get_post(S,_Args,Res) -> eq(Res,hd(S#state.contents)).
```

Here the model state `S` is actually an Erlang record (`state`) with fields `ptr` (pointer to the queue), `contents` (expected list of elements), and `size` (maximum size). `get_pre` is the precondition: `get` may only be called if the `ptr` is defined (*i.e.* `new` has been called), and the queue is non-empty. `get_next` is the state transition function: it replaces the list of elements in the model state by its tail. `get_post` is the postcondition, which checks that the actual result of `get`, Res, is equal to the first element of the expected contents. QuickCheck recognises these functions by virtue of their names (get followed by a particular suffix), and uses them to generate and run random tests. The property tested is: if all operation preconditions hold during a test run, then so do the postconditions.

Tests are run by invoking QuickCheck from the Erlang shell, giving this property as a parameter. In this case, the tests fail almost immediately:

```
18> eqc:quickcheck(q_eqc:prop_q()).
....Failed! After 5 tests.
[{set,{var,1},{call,q,new,[1]}},
 {set,{var,2},{call,q,put,[{var,1},1]}},
...10 lines of output elided...

q:new(1) -> {ptr,"Queue", 4003800}
q:put({ptr,"Queue", 4003800}, 1) -> ok
q:get({ptr,"Queue", 4003800}) -> 1
q:put({ptr,"Queue", 4003800}, -1) -> ok
q:put({ptr,"Queue", 4003800}, -1) -> ok
q:put({ptr,"Queue", 4003800}, 0) -> ok
q:get({ptr,"Queue", 4003800}) -> 0
Reason: Post-condition failed: 0 /= -1
```

After four successful tests (represented by a '.' in the output), QuickCheck generated a failing example, which appears next in a symbolic form—but in this paper, we focus on the pretty-printed results that follow it (beginning with the line q:new(1) ->...). These show the calls made and results returned—and clearly, something is wrong, because the postcondition of the last call failed. 0, the *actual* result of the call, was not equal to -1, the *expected* value.

Like most randomly generated failing tests, this one contains irrelevant calls as well as those few that actually provoked the failure. So QuickCheck then *shrinks* the test case to a simpler one, searching for the *smallest similar test* that also fails. In this case,the output continues:

```
Shrinking xxxx..xx.xx.xxxx.xxx(5 times)
[{set,{var,1},{call,q,new,[1]}},
 {set,{var,2},{call,q,put,[{var,1},1]}},
 {set,{var,3},{call,q,put,[{var,1},0]}},
 {set,{var,4},{call,q,get,[{var,1}]}}]

q:new(1) -> {ptr,"Queue", 4009248}
q:put({ptr,"Queue", 4009248}, 1) -> ok
q:put({ptr,"Queue", 4009248}, 0) -> ok
q:get({ptr,"Queue", 4009248}) -> 0
Reason: Post-condition failed: 0 /= 1
```

During shrinking, each '.' represents a smaller test that also failed—progress towards the minimal failing test—while 'x' represents a smaller test that passed. Shrinking not only removes unnecessary *calls* from the test case, but also simplifies *arguments* where possible—in particular, -1 no longer appears in the test.

While randomly generated failing tests vary widely, the result after shrinking is very consistent—always either the test case above, or this very similar one:

```
q:new(1) -> {ptr,"Queue", 4027504}
q:put({ptr,"Queue", 4027504}, 0) -> ok
q:put({ptr,"Queue", 4027504}, 1) -> ok
q:get({ptr,"Queue", 4027504}) -> 1
Reason: Post-condition failed: 1 /= 0
```

Debugging the Model. But why does the test above fail? Inspecting the results, we see that first we create a queue with space for one element, then we put a 0 into it, then we put a 1 into it—and at this point, the queue should contain $[0, 1]$—then finally we get the first element, which should be 0, but was actually 1! Clearly this is wrong!

But *why* did the C code return 1? Tracing through the example, we see that *we allocated a buffer large enough to hold one integer*, and then put two integers into it! Since we implemented a circular buffer, then the second value (1) overwrites the first (0). This is why `get` returns 1—the data has been corrupted by putting too many elements into the queue.

This really is a *minimal* test case for this error: not only *must* we perform two `put`s and a `get`, but we *must* put two different values—otherwise we overwrite a value with an equal one, and the error is not detectable. This is why the values put are 0 and 1—one of them is arbitrary, so can shrink to 0, while the other *must be different.* The simplest value different from 0 is 1, so this is the value we see. In retrospect, it was very informative that we did not see *two* zeroes—this told us immediately that if we were to put two zeroes, then the test would pass.

So where is the error? Recall that the code is not *intended* to behave sensibly when abused—and putting two elements into a queue of size one is surely abuse. Arguably, there is nothing wrong with the code—this is *a bad test.* This means that the fault is not in the implementation, but in the *model.* Inspecting the *precondition* of `put` in the model we have been using so far, we find

```
put_pre(S) -> S#state.ptr /= undefined.
```

This allows tests to call `put` at any time, as long as the queue pointer is defined. We should of course only allow calls of `put` if the queue is not already full:

```
put_pre(S) -> S#state.ptr /= undefined andalso
                  length(S#state.contents) < S#state.size.
```

With this change to the precondition, we can repeat the last failed test, and see

```
q:new(1) -> {ptr,"Queue", 4027648}
q:put({ptr,"Queue", 4027648}, 0) -> ok
q:put({ptr,"Queue", 4027648}, 1) -> !!! precondition_failed
Reason: precondition_failed
```

It still fails, but at an earlier point and for a different reason—the second call to `put` can no longer be made, because its precondition is not satisfied.

Random tests now pass, and we see satisfying output from QuickCheck:

```
28> eqc:quickcheck(q_eqc:prop_q()).
...........................................................
.................................
OK, passed 100 tests
53.5% {q,put,2}
40.2% {q,get,1}
6.3% {q,new,1}
```

We usually collect statistics as tests are run; in this case we collect the names of the functions called ({`q,new,1`} represents `q:new/1`, the `new` function with 1 argument in module `q`, and so on). Over 50 % of the calls tested were to `put`, around 40 % were to `get`—which is not so surprising, since given our preconditions, we cannot call `get` unless there is a corresponding call to `put`. However, these statistics reveal that we have *not* tested the `size` function at all! There is a simple reason for this—for simplicity, we omitted it from the model.

Debugging the Code. It is easy to model the behaviour of `size` as well, but with this extension to the model, tests fail immediately. The shrunk example is this:

```
q:new(1) -> {ptr,"Queue", 4033488}
q:put({ptr,"Queue", 4033488}, 0) -> ok
q:size({ptr,"Queue", 4033488}) -> 0
Reason: Post-condition failed: 0 /= 1
```

We create a queue with room for one element, put an element into it, then ask how many there are—which of course should be 1, but `size` returns 0! This time the model is not at fault—the C code is simply returning the wrong answer.

Using this example we can trace through the code in Fig. 1 to find the problem. `new(1)` initializes `q->size` to `1`. `size` returns

```
  (q->inp - q->outp) % q->size
```

But `q->size` is 1, and any value modulo 1 is zero! It follows that `size` *always* returns 0 when the queue size is 1, no matter how many times we call `put`. Likewise, `put` and `get` increment their respective indices *modulo* 1, so these indices are also always zero. Putting an element into the queue does not change its state—a full queue looks exactly the same as an empty one! This is the real problem, and it is not limited to queues of size one. If we create a queue of size n, and put n items into it, then the same thing happens—the input index wraps around back to zero, and a full queue looks the same as an empty one. Our *representation* of queues is inadequate—it cannot distinguish full from empty.

How to fix this? There are several possibilities—we could, for example, add a boolean to the `Queue` struct to distinguish full queues, but this would lead to special cases throughout the code. Much nicer is to modify the `new` function so that, when asked to create a queue with space for n elements, we actually create one with space for $n + 1$ elements!

```
Queue *new(int n)
{ int *buff = malloc((n+1)*sizeof(int));
  Queue q = {buff,0,0,n+1};
  Queue *qptr = malloc(sizeof(Queue));
  *qptr = q;
  return qptr;
}
```

Let this be our secret. Now, the only way for a caller to witness the 'bug' is by putting $n+1$ elements into a queue with room for n—violating the precondition of put, which means *the problem is their fault!* We successfully avoid blame, and can go home happy.

Sure enough, with this change then repeating the last test succeeds:

```
32> eqc:check(q_eqc:prop_q()).
OK, passed the test.
```

However, running random tests provokes another failure, which always shrinks to the same test case:

```
q:new(1) -> {ptr,"Queue", 5844232}
q:put({ptr,"Queue", 5844232}, 0) -> ok
q:get({ptr,"Queue", 5844232}) -> 0
q:put({ptr,"Queue", 5844232}, 0) -> ok
q:size({ptr,"Queue", 5844232}) -> -1
Reason: Post-condition failed: -1 /= 1
```

We create a queue of size one (that's really two!), and put a zero into it—now it's full. We get the zero—and now it's empty, so we can put another zero in. Now the queue should contain one zero... but when we ask the size, then -1 is returned! This is clearly a bug in the C code—the size should never be negative.

Tracing this example through the code in Fig. 1, we see that we really created a queue with a q->size of 2, then called put twice, so the input index q->inp wraps around back to zero. We called get once, so q->outp is 1, and in size, (q->inp - q->outp) % q->size computes (-1) % 2—but $(-1) \bmod 2$ is $+1$, as we all learned in school. Or is it? Trying it in Erlang:

```
34> (-1) rem 2.
-1
```

In Erlang, at least, rem is *not* the modulo operator—it computes the remainder after integer division, *rounding towards zero*. Since (-1) div 2 is zero, then (-1) rem 2 must be -1. The same is true in C.

With this in mind, we see that size returns a negative result whenever q->inp has wrapped around, but q->outp has not—whenever

```
  q->inp - q->outp
```

is negative. To fix the bug, we must ensure that this expression *is* never negative, and a simple way to do so is to apply abs to it. We modify size as follows,

```
int size(Queue *q)
{ return abs(q->inp - q->outp) % q -> size;
}
```

and repeat the last test:

```
36> eqc:check(q_eqc:prop_q()).
OK, passed the test.
```

Of course it passes, as we would expect.

Now, notice what we did. We found a subtle bug in our implementation. We created a test case that provoked the bug. We fixed the code, and our test now passes. All our tests are green! So... we're done, right?

In practice, many developers would be satisfied at this point. But let us just run a few more random tests... we find they fail immediately!

```
q:new(2) -> {ptr,"Queue", 6774864}
q:put({ptr,"Queue", 6774864}, 0) -> ok
q:put({ptr,"Queue", 6774864}, 0) -> ok
q:get({ptr,"Queue", 6774864}) -> 0
q:put({ptr,"Queue", 6774864}, 0) -> ok
q:size({ptr,"Queue", 6774864}) -> 1
Reason: Post-condition failed: 1 /= 2
```

What's going on here? First, we create a queue of size two. Progress! The code now works—for queues of size one. We put *three* items into the queue, and remove one. Now there should be two items in the queue, but `size` returns 1.

This is actually very similar to the example we just fixed. The queue is size two (really three!), so calling `put` three times makes the input index wrap around back to zero. We called `get` once, so the output index is one. The correct answer for `size` is thus $(-1) \bmod 3$, which is 2, but we computed `abs(-1) % 2`, which is `1`. Taking the absolute value was the wrong thing to do; it worked for the first test case we considered, but not in general.

A correct implementation must ensure that the operand of `%` is non-negative, *without* changing its value modulo `q->size`. We can do this by *adding* `q->size` to it, instead of taking the absolute value:

```
int size(Queue *q)
{ return (q->inp - q->outp + q->size) % q -> size;
}
```

When we make this change, then this last failing test—and all the others we can generate—pass. The code is correct at last.

Lessons from the Queue Example. What can we learn from this? Three lessons that apply in general:

– The *same* property can find many *different* bugs. This is the whole point of test case generation: one model can provoke a wide variety of bugs.

– Errors are often in the model, rather than the code. Calling something a specification does not make it right! QuickCheck finds *discrepancies* between the model and the code; it is up to the developer to decide which is correct.
– *Minimal failing test cases* make debugging easy. They are often much smaller than the first failing one found. We think of shrinking as "extracting the signal from the noise". Random tests contain a great deal of junk—that is their purpose! Junk provokes unexpected behaviour and tests scenarios that the developer would never think of. But tests usually *fail* because of just a few features of the test case. Debugging a test that is 90 % irrelevant is a nightmare; presenting the developer with a test where *every part* is known to be relevant to the failure, simplifies the debugging task enormously.

Property-Based Testing. I referred several times above to the "property" that is being tested, namely that "*if all operation preconditions hold during a test run, then so do the postconditions*". Properties are fundamental to QuickCheck: indeed, QuickCheck *always* tests a property, by generating suitable test cases and checking that the property holds in each case. Properties are often expressed in terms of a stateful model—as in this section—but this is not always the case, and even for the same model, the properties tested may differ. Since properties are fundamental here, we call the general approach "*property-based testing*", rather than "model-based testing" which is a better-established term.

3 The Volvo Project

So far, we only considered a toy example—the reader could be forgiven for wondering if the method really scales. After all, in this example, the specification was actually larger than the code! QuickCheck would not be useful if this were true in general. So, let us summarize the results of the largest QuickCheck project so far—acceptance testing of AUTOSAR Basic Software for Volvo Cars (Arts et al. 2015).

Modern cars contain 50–100 processors, and many manufacturers base their software on the AUTOSAR standard. Among other things, this defines the 'Basic Software', which runs on *every* processor in the car. It consists of an Ethernet stack, a CAN-bus stack (the CAN bus, or "Controller Area Network", is heavily used in vehicles), a couple of other protocol stacks, a routing component, and a diagnostics cluster (which records fault codes as you drive). The basic software is made up of many modules, whose behaviour is *specified* in detail, but there are a number of suppliers offering competing implementations. Usually a car integrates basic software from several different suppliers—which means that if they do *not* follow the standard, then there is a risk of system integration problems. It is not even easy to tell, if two processors cannot talk to each other, *which* of the suppliers is at fault!

Volvo Cars wanted to *test* their suppliers code for conformance with the standard, and funded Quviq to develop QuickCheck models for this purpose.

Most errors we found are confidential, but we can describe one involving the CAN bus stack, which we tested using a mocked hardware bus driver. Now, every message sent on the CAN bus has a *CAN identifier*, or message type, which also serves as the *priority* of the message—the smaller the CAN identifier, the higher the priority. QuickCheck found a failing test with the following form:

– Send a message with CAN identifier 1 (and check that the hardware driver is invoked to put it onto the bus).
– Send a message with CAN identifier 2 (which should *not* be passed to the driver yet, because the bus is busy).
– Send a message with CAN identifier 3 (which should also not be sent yet).
– Confirm transmission of message 1 by the bus (and check that the driver is now invoked to send message 2).

Of course, the bug was that the stack sent the message with identifier 3 instead.

This *is* the smallest test case that can provoke the bug. We must first send a message, making the bus busy, and then send two more with different priorities, which are both queued. Then we must release the bus, and observe that the wrong message is taken from the queue. This is just what the test does.

The source of the bug was this: the original CAN standard allowed only 11 bits for the message identifier, but today 2048 different message types is no longer enough, so the current standard *also* allows an 'extended CAN id' of 29 bits. Both kinds have the same meaning—in particular, the priority does not depend on the encoding—but when the stack sends a message, it must know which format to use. This particular stack stored both kinds of identifier in the same unsigned integer, but used bit 31 to indicate an extended identifier. In this test, message number 2 used an extended CAN identifier, while message number 3 used the original format. Of course, when comparing stored message identifiers in order to decide which message to send, it is essential to mask off the top bit... which the developer forgot to do in this case. So the second message was treated as priority $2^{31} + 2$, and was not chosen for sending next.

The actual fault is a tricky mistake in low-level C code—nevertheless, *it can be provoked with a short sequence of API calls*, and *we can find this sequence by generating a random one and shrinking it.*

The fault is also potentially serious. Those priorities are there for a reason. Almost everything in the car can talk on the CAN bus—the brakes, the stereo.... Here's a tip: don't adjust the volume during emergency braking!

For this project we read around 3,000 pages of PDFs (the standards documents). We formalized the specification in 20,000 lines of QuickCheck code. We used it to test a million lines of C code in total, from 6 different suppliers, finding more than 200 problems—of which well over 100 were ambiguities or inconsistencies in the standard itself! In the few cases where we could compare our test code to traditional test suites (in TTCN3), our code was an order of magnitude smaller—but tests more.

So, does the method scale? Yes, it scales.

4 Debugging a Database

The last story (Hughes and Bolinder 2011) begins with a message to the Erlang mailing list, from Tobbe Törnqvist at Klarna.

"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year. We have not been able to track the bug down since the dets files is repaired automatically next time it is opened."

The context: Klarna was a Swedish start-up offering invoicing services to web shops[1]. They built their product in Erlang, and stored their data in Mnesia, the database packaged with the Erlang distribution. Mnesia provides transactions, distribution, replication, and so on, but needs a back end to actually store the data. The back end for storage on disk is *dets*, which stores tuples in files. This problem piqued my interest—it was quite notorious in the Erlang community, and sounded as though it might involve a race condition.

To explain how QuickCheck tests for race conditions, let us consider a much simpler example: a ticket dispenser such as one finds at delicatessen counters, which issues each customer with a ticket indicating their place in the queue. We might model such a dispenser in Erlang by two functions, `take_ticket()` and—because the roll of tickets needs to be replaced sometimes—`reset()`. A simple unit test for these functions might look as follows:

```
ticket_test() ->
  ok = reset(),
  1  = take_ticket(),
  2  = take_ticket(),
  3  = take_ticket(),
  ok = reset(),
  1  = take_ticket().
```

Here we use Erlang pattern matching to compare the result of each call with its *expected value* (`ok` being an atom, a constant, in Erlang—conventionally used as the result of a function with no interesting result). This test case is typical of test cases written in industry: we make a sequence of calls, and compare the results to expected values.

It is easy to create a QuickCheck state machine for testing the dispenser also. We can model its state as a single integer, which is set to 0 by `reset()`, incremented by `take_ticket()`, and used in the postcondition of `take_ticket` to check the result.

However, running *sequential* tests of the ticket dispenser is not enough: the whole point of the system is to regulate the flow of many concurrent customers to one delicatessen counter. If we do not also run parallel tests of the API, then we are failing to test an essential part of its behaviour.

[1] They have since grown to well over 1,000 people, serving 8 countries and counting.

Now, consider one simple parallel test:

reset()	
take_ticket()	take_ticket()
take_ticket()	

which represents first resetting the dispenser, then *two* customers taking tickets in parallel—the left customer taking two tickets, the right customer taking one. *What are the expected results?* Well, the left customer might get tickets #1 and #2, and the right customer ticket #3. Or, the left customer might get tickets #1 and #3, while the right customer gets ticket #2! But if *both* customers get ticket number #1, then the test has failed.

This poses a problem for the traditional way of writing test cases, because *we cannot predict the expected results*. In this case, we could in principle compute all three possibilities (in which the right customer gets ticket number #1, #2 or #3), and compare the actual results against all three—but this approach does not scale at all. Consider this only slightly larger test:

reset()		
take_ticket()	take_ticket()	reset()
take_ticket()	take_ticket()	

This test has *thirty* possible correct outcomes! No developer in their right mind would try to enumerate all of these—and anyone who tried would surely get at least one wrong. The *only* practical way to decide if a test such as this has passed or failed, is via a *property* that distinguishes correct from wrong results. Parallel testing is a 'killer app' for property-based testing.

The way that QuickCheck decides whether a test like this has passed is by searching for *any interleaving* of the calls which makes their results match the model. If there is such an interleaving, then the test passes—but if there is *no* such interleaving, then it has definitely failed. Here we are using the *same* model as we used for sequential testing, to test a *different property* of the implementation—namely, serializability of the operations.

Testing a buggy version of the dispenser, QuickCheck immediately reports:

```
Parallel:
1. dispenser:take_ticket() --> 1
2. dispenser:take_ticket() --> 1
Result: no_possible_interleaving
```

That is, two parallel calls of `take_ticket` can both return 1—but no *sequence* of two calls can return these results. In this case, there is a blatant bug in the code: `take_ticket()` reads and then writes a global variable containing the next ticket number, with no synchronization at all—so no wonder there is a race condition. But the point is that QuickCheck *finds* the bug very fast, and *shrinks* it to this minimal example very reliably indeed.

Returning to dets, it stores tuples of the form {Key,Val$_1$, Val$_2$...} in a file, and its API is unsurprising. For example,

- insert(Table,List) inserts a list of tuples into a table,
- delete(Table,Key) deletes all tuples with the given key from the table, and
- insert_new(Table,List) is like insert, unless one of the keys to be inserted is already in the table, in which case it is a no-op.

These operations are all guaranteed to be atomic.

A state machine modelling dets is easy to construct—it is almost enough just to model the table by a list of the tuples that should be in it. My model of the core of the dets API is less than 100 lines of code—which compares well to the implementation, over 6,000 lines of code spread over four modules, maintaining hash tables on the disk, supporting an old format for the files, and so on. Thus, although dets is quite complex, its intended behaviour is simple.

I first tested the model thoroughly by running tens of thousands of sequential tests. This ensures that the model and the code are consistent—and it did turn up a few surprises. There is no point running parallel tests of a system, if inconsistencies can already be found using sequential tests. (I assumed that dets behaves correctly in the sequential case—it is mature and well-tested code, after all—so any inconsistencies indicate a misunderstanding of the intended behaviour, *i.e.* a bug in the model).

Once the model was correct, I began to run parallel tests. This only required writing a few lines of code, because we reuse the *same* model for sequential and parallel testing. Almost immediately, a real bug appeared:

```
Prefix:
   open_file(dets_table,[{type,bag}]) --> dets_table
Parallel:
1. insert(dets_table,[]) --> ok
2. insert_new(dets_table,[]) --> ok
Result: no_possible_interleaving
```

In the sequential prefix of the test, we open the dets file, then in parallel call insert and insert_new. Both insert an *empty* list of tuples—so they are both no-ops—and they both return ok! At first sight this looks reasonable—but the documentation for insert_new says it should return a boolean! Indeed, in tens of thousands of sequential tests, insert_new returned true or false every time—now, suddenly, it returned ok. Strange!

Another run of QuickCheck reported a second bug:

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table
Parallel:
1. insert(dets_table,{0,0}) --> ok
2. insert_new(dets_table,{0,0}) ...time out...
```

In this case, both insert and insert_new try to insert the same tuple, which means that insert_new could succeed or fail... but it should *not* time out, and we should *not* see the message

```
=ERROR REPORT==== 4-Oct-2010::17:08:21 ===
** dets: Bug was found when accessing table dets_table
```

At this point I disabled the testing of insert_new, which seemed not work in parallel, and discovered a third bug:

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table
Parallel:
1. open_file(dets_table,[{type,set}]) --> dets_table
2. insert(dets_table,{0,0}) --> ok
   get_contents(dets_table) --> []
Result: no_possible_interleaving
```

Here we first open the file, and then *in parallel* open it again, and insert a tuple, then fetch the entire contents of the table. The test fails because process 2 sees an empty table, *even though it just inserted a tuple*! It may seem a little suspicious to open the table twice, but it is not at all—Erlang is designed for highly concurrent applications, so we *expect* many processes to be using the table at the same time, and all of them need to make sure it is open.

I reported these bugs to Hans Bolinder at Ericsson, who was responsible for the dets code, and next day he sent me a thank you and a fixed version. However, he thought these were probably not the bugs that were plaguing Klarna, because the symptoms were different. At Klarna, the file was being corrupted. Hans gave me one line of code that could check for corruption, and I added it to the test to check that, after executing a parallel test, the file was not corrupt.

This time it took around 10 min of testing to find a bug:

```
Prefix:
   open_file(dets_table,[{type,bag}]) --> dets_table
   close(dets_table) --> ok
   open_file(dets_table,[{type,bag}]) --> dets_table
Parallel:
1. lookup(dets_table,0) --> []
2. insert(dets_table,{0,0}) --> ok
3. insert(dets_table,{0,0}) --> ok
Result: ok
```

First we open, close, and open the file again, and then we do *three* things in parallel: a lookup of a key that is not found, and two insertions of the same tuple. All results are consistent with the model—but the corruption check encountered a 'premature eof'.

This is really the *smallest test case* that can provoke this error. It was initially hard to believe that the open–close–open sequence was really necessary. I manually removed the first open and close, and ran the smaller test case tens of thousands of times. It passed, every single time. Today I know why: the first call to open *creates* the file, while the second call opens an *existing* file. It's slightly different, and dets enters a slightly different state—and that state is key to provoking the bug. Three parallel operations are needed because the first makes the dets server busy, causing the next two to be queued up, and then (wrongly) dispatched in parallel by the server, once the first call is complete.

The final bug was found when preparing a talk on this experience soon afterwards, while rerunning the tests in order to copy the output onto slides. Again, after around ten minutes of testing, the following bug was found:

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table
   insert(dets_table,[{1,0}]) --> ok
Parallel:
1. lookup(dets_table,0) --> []
   delete(dets_table,1) --> ok
2. open_file(dets_table,[{type,set}]) --> dets_table
Result: ok
```

We open the file and insert a tuple, then, in parallel, *reopen* the file, and lookup a key not in the table, then delete the key that is. All the results are consistent with the model—but the corruption check encountered a 'bad object'. Recall the mailing list message:

> *"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year."*

So it seemed these might indeed be the Klarna bugs.

In each case, Hans Bolinder returned a fixed version of the code within a day of receiving the bug report—and at Klarna, where the problem had meanwhile begun to appear once a week, there has been only *one* 'bad object' error since the new code went into production... when a reading a file last written *before* the new code was installed.

Prior to this work Hans Bolinder had spent six weeks at Klarna hunting for the bug, and the best theory was that it struck when the file was around 1 GB, and might be something to do with rehashing. Now we know the bugs could be provoked with a database with at most one record, using only 5–6 calls to the API. In each case, given the minimal failing test, it was less than a day's work to find and fix the bug. This really demonstrates the enormous value of shrinking—and shows just how hopeless it is to try to debug this kind of problem from failures that happen in production, where not only the five or six relevant things have occurred, but also millions of irrelevant others.

5 In Conclusion

This paper recounts some of *our* experiences of using an advanced testing tool in real applications—its purpose is as much to entertain as to inform. Of course, there is a wealth of other work in the area, that this paper makes no attempt to cover—RANDOOP (Pacheco et al. 2007), DART (Godefroid et al. 2005), and their successors would be good starting points for further reading.

I have also just touched the surface of the work that we have done. There was far more to the Volvo project than could be described here. We needed to develop a property-based approach to mocking (Svenningsson et al. 2014). We needed to develop ways to *cluster* models of individual components into an integrated model of a subsystem, because AUTOSAR subsystems are typically *implemented* monolithically, but *specified* as a collection of interacting components. We found that QuickCheck's shrinking tended to isolate the *same* bug in each run—which is not ideal when generating a test report on a component with *several different* bugs—so we developed methods to direct testing away from already-known problems towards areas where unknown bugs might still lurk, and report *all* the bugs at once. We were asked "does your model cover this test case?", so we developed tools to answer such questions—and when the AUTOSAR consortium released six "official" test cases for the CAN bus stack, we used them to show that three of these test cases were *not* covered by the model, because they were not consistent with the standard (Gerdes et al. 2015). We were asked: "which *requirements* have you tested?", so we developed ways to collect requirements coverage during testing—leading us to question what testing a requirement even *means* (Arts and Hughes 2016).

In other areas, we helped Basho test their no-SQL database, Riak, for the key property of *eventual consistency*—and found a bug (now fixed, of course) that was present, not only in Riak, but in the original Amazon paper (DeCandia et al. 2007) that kicked off the no-SQL trend. Recently we tested Dropbox's file synchronization service, with some surprising results there too (Hughes et al. 2016). Both these applications were based in part on the state machine framework presented here, but added additional *properties* to express subtle correctness conditions.

Perhaps the most general lesson from all this experience is that formulating specifications is *hard*—and many developers struggle with it. It may often be easier to give *examples* of correct behaviour, than to define *in general* what correctness means. This should not come as a surprise, since mathematicians have used examples for centuries—but it does suggest that starting from a specification may not always be appropriate. Sometimes I am asked: "wouldn't it be great if we could just synthesize code from a specification?" It would not: we would just replace buggy code with buggy specifications. A key insight from using QuickCheck is that we find bugs by comparing *two independent descriptions* of the desired behaviour—the implementation, and the specification—and it is the inconsistencies between the two that reveal errors.

The *need for a specification* is the real weakness of property-based testing—not that we use testing, rather than static analysis or proof, to relate specifications and implementations. Testing works well enough! The real question is: how can we make specifications more useable, and easier to construct—for real systems with all their complex behaviours? There is much work to be done here—perhaps future tools will even help developers construct specifications from examples.

In any event, there are many fascinating problems waiting to be addressed. If this paper helps persuade the reader that testing is not a chore that must be done, but a fascinating research area in its own right, then it will have fulfilled its purpose.

And don't forget,

DON'T WRITE TESTS, GENERATE THEM!

Acknowledgements. The work described here was partially supported by the EU FP7 project "PROWESS", and by the Swedish Foundation for Strategic Research grant "RAWFP".

References

Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing AUTOSAR software with QuickCheck. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–4, April 2015

Arts, T., Hughes, J.: How well are your requirements tested? In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST), IEEE, April 2016

Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ERLANG 2006, pp. 2–10. ACM, New York (2006)

Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York (2000)

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 205–220. ACM, New York (2007)

Gerdes, A., Hughes, J., Smallbone, N., Wang, M.: Linking unit tests and properties. In: Proceedings of the 14th ACM SIGPLAN Workshop on Erlang, Erlang 2015, pp. 19–26. ACM, New York (2015)

Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 213–223. ACM, New York (2005)

Hughes, J.: QuickCheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2007)

Hughes, J., Pierce, B., Arts, T., Norell, U.: Mysteries of dropbox: property-based testing of a distributed synchronization service. In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST), IEEE, April 2016

Hughes, J.M., Bolinder, H.: Testing a database for race conditions with quickcheck. In: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang 2011, pp. 72–77. ACM, New York (2011)

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 75–84. IEEE Computer Society, Washington, DC (2007)

Svenningsson, J., Svensson, H., Smallbone, N., Arts, T., Norell, U., Hughes, J.: An expressive semantics of mocking. In: Gnesi, S., Rensink, A. (eds.) FASE 2014 (ETAPS). LNCS, vol. 8411, pp. 385–399. Springer, Heidelberg (2014)

Cutting Out Continuations

Graham Hutton[1(✉)] and Patrick Bahr[2]

[1] University of Nottingham, Nottingham, UK
graham.hutton@nottingham.ac.uk
[2] IT University of Copenhagen, Copenhagen, Denmark
paba@itu.dk

Abstract. In the field of program transformation, one often transforms programs into continuation-passing style to make their flow of control explicit, and then immediately removes the resulting continuations using defunctionalisation to make the programs first-order. In this article, we show how these two transformations can be fused together into a single transformation step that cuts out the need to first introduce and then eliminate continuations. Our approach is calculational, uses standard equational reasoning techniques, and is widely applicable.

1 Introduction

In his seminal work on definitional interpreters for higher-order programming languages, Reynolds [12] popularised two techniques that have become central to the field of program transformation: (i) transforming programs into *continuation-passing style* (CPS) to make their flow of control explicit; and (ii) transforming higher-order programs into first-order style using *defunctionalisation*. These two transformations are often applied together in sequence, for example to transform high-level denotational semantics for programming languages into low-level implementations such as abstract machines or compilers [1,2,11–13].

However, there is something rather unsatisfactory about applying these two transformations in sequence: the CPS transformation introduces continuations, only for these to immediately be removed by defunctionalisation. This seems like an ideal opportunity for applying *fusion*, which in general aims to combine a series of transformations into a single transformation that achieves the same result, but without the overhead of using intermediate structures between each step. In our case, the intermediate structures are continuations.

Despite CPS transformation and defunctionalisation being obvious candidates for being fused into a single transformation, this does not appear to have been considered before. In this article, we show how this can be achieved in a straightforward manner using an approach developed in our recent work on calculating compilers [5]. Our approach starts with a specification that captures the intended behaviour of the desired output program in terms of the input program. We then calculate definitions that satisfy the specification by *constructive induction* [4], using the desire to apply induction hypotheses as the driving force for the calculation process. The resulting calculations only require

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 187–200, 2016.
DOI: 10.1007/978-3-319-30936-1_10

standard equational reasoning techniques, and achieve the combined effect of CPS and defunctionalisation without the intermediate use of continuations.

We illustrate our approach by means of two examples: a minimal language of arithmetic expressions to introduce the basic ideas in a simple manner, and a call-by-value lambda calculus to show how it can be applied to a language with variable binding. More generally, the approach scales up to a wide range of programming language features and their combination, including exceptions, state, loops, non-determinism, interrupts, and different orders of evaluation.

The article is aimed at a general functional programming audience rather than specialists, and all the programs and calculations are written in Haskell. The calculations have also been mechanically verified using the Coq proof assistant, and the proof scripts for our two examples, together with a range of other applications, are freely available online via GitHub [7].

2 Arithmetic Expressions

Consider a simple language of arithmetic expressions built up from integers and addition, whose syntax and denotational semantics are defined as follows:

$$\begin{array}{ll}
\mathbf{data}\ \mathit{Expr} & = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) & = n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = \mathit{eval}\ x + \mathit{eval}\ y
\end{array}$$

Now suppose that we wish to calculate an abstract machine for this language, which is given by a set of first-order, tail-recursive functions that are together semantically equivalent to the function *eval*. We initially perform this calculation using the standard two step approach [1]: transformation into continuation-passing style, followed by defunctionalisation. Then in Sect. 3 we show how these two transformation steps can be combined into a single step.

Step 1 - Add Continuations

The first step is to transform the evaluation function into continuation-passing style. More specifically, we seek to derive a more general evaluation function

$$\mathit{eval}' :: \mathit{Expr} \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{Int}$$

that takes a *continuation* of type $\mathit{Int} \rightarrow \mathit{Int}$ as additional argument, which is applied to the result of evaluating the expression. More precisely, the desired behaviour of eval' is specified by the following equation:

$$\mathit{eval}'\ x\ c = c\ (\mathit{eval}\ x) \qquad (1)$$

Rather than first defining the new function eval' and then separately proving by induction that it satisfies the above equation, we aim to *calculate* a definition for *eval′* that satisfies this equation by *constructive induction* on the expression argument x. In each case, we start with the term $eval'\ x\ c$ and gradually transform it by equational reasoning, aiming to arrive at a term t that does not refer to the original semantic function *eval*, such that we can then take $eval'\ x\ c = t$ as a defining equation for *eval′* in this case. For the base case, when the expression has the form $Val\ n$, the calculation is trivial:

$$\begin{array}{ll} & eval'\ (Val\ n)\ c \\ = & \quad \{\text{ specification (1) }\} \\ & c\ (eval\ (Val\ n)) \\ = & \quad \{\text{ applying } eval \}\\ & c\ n \end{array}$$

By means of this calculation, we have *discovered* the definition for *eval′* in the base case, namely: $eval'\ (Val\ n)\ c = c\ n$. In the inductive case, when the expression has the form $Add\ x\ y$, we start in the same manner as above:

$$\begin{array}{ll} & eval'\ (Add\ x\ y)\ c \\ = & \quad \{\text{ specification (1) }\} \\ & c\ (eval\ (Add\ x\ y)) \\ = & \quad \{\text{ applying } eval \}\\ & c\ (eval\ x + eval\ y) \end{array}$$

Now we appear to be stuck, as no further definitions can be applied. However, as we are performing an inductive calculation, we can use the induction hypotheses for x and y, that is, $eval'\ x\ c' = c'\ (eval\ x)$ and $eval'\ y\ c'' = c''\ (eval\ y)$. To use these hypotheses, we must rewrite the term being manipulated into the form $c'\ (eval\ x)$ and $c''\ (eval\ y)$ for some continuations c' and c''. This can be readily achieved by abstracting over $eval\ x$ and $eval\ y$ using lambda expressions. Using this idea, the rest of the calculation is then straightforward:

$$\begin{array}{ll} & c\ (eval\ x + eval\ y) \\ = & \quad \{\text{ abstracting over } eval\ y \}\\ & (\lambda m \rightarrow c\ (eval\ x + m))\ (eval\ y) \\ = & \quad \{\text{ induction hypothesis for } y \}\\ & eval'\ y\ (\lambda m \rightarrow c\ (eval\ x + m)) \\ = & \quad \{\text{ abstracting over } eval\ x \}\\ & (\lambda n \rightarrow eval'\ y\ (\lambda m \rightarrow c\ (n + m)))\ (eval\ x) \\ = & \quad \{\text{ induction hypothesis for } x \}\\ & eval'\ x\ (\lambda n \rightarrow eval'\ y\ (\lambda m \rightarrow c\ (n + m)) \end{array}$$

The same result can also be achieved by applying the induction hypotheses in the opposite order to the above, but for the purposes of the later fused calculation it is important that we apply the induction hypothesis on y first if we want to ensure that the resulting abstract machine evaluates the arguments of addition from left-to-right. In conclusion, we have calculated the following definition:

$$
\begin{array}{ll}
\mathit{eval}' & :: \mathit{Expr} \to (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\
\mathit{eval}'\ (\mathit{Val}\ n)\quad c & = c\ n \\
\mathit{eval}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{eval}'\ x\ (\lambda n \to \mathit{eval}'\ y\ (\lambda m \to c\ (n+m)))
\end{array}
$$

Finally, our original evaluation function can now be redefined in terms of the new version by taking the identity function as the continuation in specification (1), which results in the definition $\mathit{eval}\ x = \mathit{eval}'\ x\ (\lambda n \to n)$.

Step 2 - Defunctionalise

The second step is to transform the new evaluation function back into first-order style, using defunctionalisation. The basic idea is to replace the higher-order type $\mathit{Int} \to \mathit{Int}$ of continuations by a first-order datatype whose values represent the specific forms of continuations that we actually need.

Within the definitions for the functions *eval* and *eval′*, there are only three forms of continuations that are used, namely one to terminate the evaluation process, one to continue once the first argument of an addition has been evaluated, and one to add two integer results together. We begin by defining an abbreviation *Cont* for the type of continuations, together with three combinators *halt*, *next* and *add* for constructing the required forms of continuations:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Cont} = \mathit{Int} \to \mathit{Int} & \\
\mathit{halt} & :: \mathit{Cont} \\
\mathit{halt} & = \lambda n \to n \\
\mathit{next} & :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Cont} \\
\mathit{next}\ y\ c & = \lambda n \to \mathit{eval}'\ y\ (\mathit{add}\ n\ c) \\
\mathit{add} & :: \mathit{Int} \to \mathit{Cont} \to \mathit{Cont} \\
\mathit{add}\ n\ c & = \lambda m \to c\ (n+m)
\end{array}
$$

Using these definitions, our evaluator can now be rewritten as:

$$
\begin{array}{ll}
\mathit{eval} & :: \mathit{Expr} \to \mathit{Int} \\
\mathit{eval}\ x & = \mathit{eval}'\ x\ \mathit{halt} \\
\mathit{eval}' & :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Int} \\
\mathit{eval}'\ (\mathit{Val}\ n)\quad c & = c\ n \\
\mathit{eval}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{eval}'\ x\ (\mathit{next}\ y\ c)
\end{array}
$$

The next stage in the process is to define a first-order datatype whose constructors represent the three combinators:

$$
\begin{array}{l}
\textbf{data}\ \mathit{CONT}\ \textbf{where} \\
\quad \mathit{HALT} :: \mathit{CONT} \\
\quad \mathit{NEXT} :: \mathit{Expr} \to \mathit{CONT} \to \mathit{CONT} \\
\quad \mathit{ADD}\ \ :: \mathit{Int} \to \mathit{CONT} \to \mathit{CONT}
\end{array}
$$

Note that the constructors for the new type have the same names and types as the combinators for the *Cont* type, except that all the items are now capitalised.

The fact that values of type *CONT* represent continuations of type *Cont* is formalised by the following denotational semantics:

$$\begin{array}{ll}
\mathit{exec} & :: \mathit{CONT} \to \mathit{Cont} \\
\mathit{exec}\ \mathit{HALT} & = \mathit{halt} \\
\mathit{exec}\ (\mathit{NEXT}\ y\ c) & = \mathit{next}\ y\ (\mathit{exec}\ c) \\
\mathit{exec}\ (\mathit{ADD}\ n\ c) & = \mathit{add}\ n\ (\mathit{exec}\ c)
\end{array}$$

In the literature this function is typically called *apply* [12]. The reason for using the name *exec* in our setting will become clear shortly. Using these ideas, our aim now is to derive a new evaluation function

$$\mathit{eval}'' :: \mathit{Expr} \to \mathit{CONT} \to \mathit{Int}$$

that behaves in the same way as our continuation semantics $\mathit{eval}' :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Int}$, except that it uses values of type *CONT* rather than continuations of type *Cont*. The desired behaviour of eval'' is specified by the equation:

$$\mathit{eval}''\ x\ c = \mathit{eval}'\ x\ (\mathit{exec}\ c) \tag{2}$$

We can now calculate a definition for eval'' from this specification, and in turn new definitions for *exec* and *eval* in terms of the new function eval''. The calculations are entirely straightforward, and can be found in Hutton and Wright [8]. The end result is the following set of definitions:

$$\begin{array}{lll}
\mathit{eval}'' & & :: \mathit{Expr} \to \mathit{CONT} \to \mathit{Int} \\
\mathit{eval}''\ (\mathit{Val}\ n) & c & = \mathit{exec}\ c\ n \\
\mathit{eval}''\ (\mathit{Add}\ x\ y) & c & = \mathit{eval}''\ x\ (\mathit{NEXT}\ y\ c) \\
\\
\mathit{exec} & & :: \mathit{CONT} \to \mathit{Int} \to \mathit{Int} \\
\mathit{exec}\ \mathit{HALT} & n & = n \\
\mathit{exec}\ (\mathit{NEXT}\ y\ c) & n & = \mathit{eval}''\ y\ (\mathit{ADD}\ n\ c) \\
\mathit{exec}\ (\mathit{ADD}\ n\ c) & m & = \mathit{exec}\ c\ (n + m) \\
\\
\mathit{eval} & & :: \mathit{Expr} \to \mathit{Int} \\
\mathit{eval}\ x & & = \mathit{eval}''\ x\ \mathit{HALT}
\end{array}$$

Together with the new type, these definitions form an abstract machine for evaluating expressions: *CONT* is the type of *control stacks* for the machine, which specify how it should continue after the current evaluation has concluded; eval'' evaluates an expression in the context of a control stack; *exec* executes a control stack in the context of an integer argument; and finally, *eval* evaluates an expression by invoking eval'' with the empty control stack *HALT*. The fact that the machine operates by means of two mutually recursive functions, eval'' and *exec*, reflects the fact that it has two modes of operation, depending on whether it is being driven by the structure of the expression or the control stack.

3 Fusing the Transformation Steps

We now show how the two separate transformation steps that were used to derive the abstract machine for evaluating expressions can be combined into a single step that avoids the use of continuations. In the previous section, we started off by defining a datatype *Expr* and a function $eval :: Expr \rightarrow Int$ that respectively encode the syntax and semantics for arithmetic expressions. Then in two steps we derived an abstract machine comprising four components:

– A datatype *CONT* that represents control stacks;
– A function $eval'' :: Expr \rightarrow CONT \rightarrow Int$ that evaluates expressions;
– A function $exec :: CONT \rightarrow Int \rightarrow Int$ that executes control stacks;
– A new definition for $eval :: Expr \rightarrow Int$ in terms of $eval''$.

By combining specifications (1) and (2), the relationship between the three functions is captured by the following equation:

$$eval''\ x\ c = exec\ c\ (eval\ x) \tag{3}$$

That is, evaluating an expression in the context of a control stack gives the same result as executing the control stack in the context of the value of the expression. The key to combining the CPS and defunctionalisation steps is to use this equation *directly* as a specification for the four additional components, from which we then aim to calculate definitions that satisfy the specification. Given that the equation involves two known definitions (*Expr* and the original *eval*) and four unknown definitions (*CONT*, $eval''$, *exec*, and the new *eval*), this may seem like an impossible task. However, with the benefit of the experience gained from our earlier calculations, it turns out to be straightforward.

We proceed from specification (3) by constructive induction on the expression x. In each case, we aim to rewrite the term $eval''\ x\ c$ into a term t that does not refer to the original semantics *eval*, such that we can then take $eval''\ x\ c = t$ as a defining equation for $eval''$. In order to do this we will find that we need to introduce new constructors into the *CONT* type, along with their interpretation by the function *exec*. The base case is once again trivial:

$$\begin{aligned}
& eval''\ (Val\ n)\ c \\
= & \quad \{ \text{ specification (3) } \} \\
& exec\ c\ (eval\ (Val\ n)) \\
= & \quad \{ \text{ applying } eval \ \} \\
& exec\ c\ n
\end{aligned}$$

The inductive case begins in the same way:

$$\begin{aligned}
& eval''\ (Add\ x\ y)\ c \\
= & \quad \{ \text{ specification (3) } \} \\
& exec\ c\ (eval\ (Add\ x\ y)) \\
= & \quad \{ \text{ applying } eval \ \} \\
& exec\ c\ (eval\ x + eval\ y)
\end{aligned}$$

At this point no further definitions can be applied. However, we can make use of the induction hypotheses for the argument expressions x and y. In order to use the induction hypothesis for y, that is, $eval''\ y\ c' = exec\ c'\ (eval\ y)$, we must rewrite the term that is being manipulated into the form $\mathit{exec}\ c'\ (\mathit{eval}\ y)$ for some control stack c'. That is, we need to solve the equation:

$$\mathit{exec}\ c'\ (\mathit{eval}\ y) = \mathit{exec}\ c\ (\mathit{eval}\ x + \mathit{eval}\ y)$$

First of all, we generalise $\mathit{eval}\ x$ and $\mathit{eval}\ y$ to give:

$$\mathit{exec}\ c'\ m = \mathit{exec}\ c\ (n + m)$$

Note that we can't simply use this equation as a definition for exec, because n and c would be unbound in the body of the definition. The solution is to package these two variables up in the control stack argument c' by adding a new constructor to the CONT type that takes these two variables as arguments,

$$\mathit{ADD} :: \mathit{Int} \to \mathit{CONT} \to \mathit{CONT}$$

and define a new equation for exec as follows:

$$\mathit{exec}\ (\mathit{ADD}\ n\ c)\ m = \mathit{exec}\ c\ (n + m)$$

That is, executing the command $\mathit{ADD}\ n\ c$ in the context of an integer argument m proceeds by adding the two integers and then executing the remaining commands in the control stack c, hence the choice of the name for the new constructor. Using these ideas, we continue the calculation:

$$\begin{array}{ll}
 & \mathit{exec}\ c\ (\mathit{eval}\ x + \mathit{eval}\ y) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ n\ c)\ m = \mathit{exec}\ c\ (n + m)\ \} \\
 & \mathit{exec}\ (\mathit{ADD}\ (\mathit{eval}\ x)\ c)\ (\mathit{eval}\ y) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \mathit{eval}''\ y\ (\mathit{ADD}\ (\mathit{eval}\ x)\ c)
\end{array}$$

No further definitions can be applied at this point, so we seek to use the induction hypothesis for x, that is, $\mathit{eval}''\ x\ c' = \mathit{exec}\ c'\ (\mathit{eval}\ x)$. In order to do this, we must rewrite the term $\mathit{eval}''\ y\ (\mathit{ADD}\ (\mathit{eval}\ x)\ c)$ into the form $\mathit{exec}\ c'\ (\mathit{eval}\ x)$ for some control stack c'. That is, we need to solve the equation:

$$\mathit{exec}\ c'\ (\mathit{eval}\ x) = \mathit{eval}''\ y\ (\mathit{ADD}\ (\mathit{eval}\ x)\ c)$$

As with the case for y, we first generalise $\mathit{eval}\ x$ to give

$$\mathit{exec}\ c'\ n = \mathit{eval}''\ y\ (\mathit{ADD}\ n\ c)$$

and then package the free variables y and c up in the argument c' by adding a new constructor to CONT that takes these variables as arguments

$$\mathit{NEXT} :: \mathit{Expr} \to \mathit{CONT} \to \mathit{CONT}$$

and define a new equation for exec as follows:

$$\mathit{exec}\ (\mathit{NEXT}\ y\ c)\ n = \mathit{eval}''\ y\ (\mathit{ADD}\ n\ c)$$

That is, executing the command *NEXT y c* in the context of an integer argument n proceeds by evaluating the expression y and then executing the control stack *ADD n c*. Using this idea, the calculation can now be completed:

$$
\begin{array}{ll}
 & eval''\ y\ (ADD\ (eval\ x)\ c) \\
= & \{\ \text{define: } exec\ (NEXT\ y\ c)\ n = eval''\ y\ (ADD\ n\ c)\ \} \\
 & exec\ (NEXT\ y\ c)\ (eval\ x) \\
= & \{\ \text{induction hypothesis for } x\ \} \\
 & eval''\ x\ (NEXT\ y\ c)
\end{array}
$$

Finally, we conclude the development of the abstract machine by aiming to redefine the original evaluation function *eval* :: *Expr* → *Int* in terms of the new evaluation function *eval''* :: *Expr* → *CONT* → *Int*. In this case there is no need to use induction as simple calculation suffices, during which we introduce a new constructor *HALT* :: *CONT* to transform the term being manipulated into the required form in order that specification (3) can then be applied:

$$
\begin{array}{ll}
 & eval\ x \\
= & \{\ \text{define: } exec\ HALT\ n = n\ \} \\
 & exec\ HALT\ (eval\ x) \\
= & \{\ \text{specification (3)}\ \} \\
 & eval''\ x\ HALT
\end{array}
$$

In summary, we have calculated the following definitions:

```
data CONT where
   HALT :: CONT
   NEXT :: Expr → CONT → CONT
   ADD  :: Int → CONT → CONT

eval''                :: Expr → CONT → Int
eval'' (Val n)    c   = exec c n
eval'' (Add x y) c    = eval'' x (NEXT y c)

exec                  :: CONT → Int → Int
exec HALT         n   = n
exec (NEXT y c) n     = eval'' y (ADD n c)
exec (ADD n c)   m    = exec c (n + m)

eval                  :: Expr → Int
eval x                = eval'' x HALT
```

These are precisely the same definitions as in the previous section, except that they have now been calculated directly from a specification for the correctness of the abstract machine, rather than indirectly using two transformation steps.

In a similar manner to the first calculation step in Sect. 2, we could have reversed the order in which we apply the induction hypotheses in the case of *Add*, which would result in an abstract machine that evaluates the arguments of addition right-to-left rather than left-to-right.

Reflection

The fundamental drawback of the two step approach is that we have to find the right specification for the first step in order for the second step to yield the desired result. This is non-trivial. How would we change the specification for the CPS transformation such that subsequent defunctionalisation yields a compiler rather than an abstract machine? Why did the specification we used yield an abstract machine? By combining the two transformation steps into a single transformation, we avoid this problem altogether: we write one specification that directly relates the old program to the one we want to calculate. As a result, we have an immediate link between the decisions we make during the calculations and the characteristics of the resulting program. Moreover, by avoiding CPS and defunctionalisation, the calculations become conceptually simpler. Because the specification directly relates the input program and the output program, it only requires the concepts and terminology of the domain we are already working on: no continuations or higher-order functions are needed.

4 Lambda Calculus

For our second example, we consider a call-by-value variant of the untyped lambda calculus. To this end, we assume for the sake of simplicity that our meta-language is strict. For the purposes of defining the syntax for the language, we represent variables using de Bruijn indices:

$$\textbf{data}\ \mathit{Expr} = \mathit{Var}\ \mathit{Int} \mid \mathit{Abs}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr}$$

Informally, *Var i* is the variable with de Bruijn index $i \geqslant 0$, *Abs x* constructs an abstraction over the expression x, and *App x y* applies the abstraction that results from evaluating the expression x to the value of the expression y.

In order to define the semantics for the language, we will use an *environment* to interpret the free variables in an expression. Using de Bruijn indices we can represent an environment simply as a list of values, in which the value of variable i is given by indexing into the list at position i:

$$\textbf{type}\ \mathit{Env} = [\mathit{Value}]$$

In turn, we will use a value domain for the semantics in which functional values are represented as *closures* [10] comprising an expression and an environment that captures the values of its free variables:

$$\textbf{data}\ \mathit{Value} = \mathit{Clo}\ \mathit{Expr}\ \mathit{Env}$$

Using these ideas, it is now straightforward to define an evaluation function that formalises the semantics of lambda expressions:

$$\begin{array}{ll} \mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Value} \\ \mathit{eval}\ (\mathit{Var}\ i)\ e & = e\ !!\ i \end{array}$$

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Abs}\ x)\ e &&= \mathit{Clo}\ x\ e\\
&\mathit{eval}\ (\mathit{App}\ x\ y)\ e &&= \mathbf{case}\ \mathit{eval}\ x\ e\ \mathbf{of}\\
& && \qquad \mathit{Clo}\ x'\ e' \to \mathit{eval}\ x'\ (\mathit{eval}\ y\ e : e')
\end{aligned}$$

Keep in mind that although we use Haskell syntax, we now assume a strict semantics for the meta language. In particular, in the case of $\mathit{App}\ x\ y$ above, this means that the argument expression y is evaluated before x'.

Unlike our first example, however, the above semantics is not compositional. That is, it is not structurally recursive: in the case for $\mathit{App}\ x\ y$ we make a recursive call $\mathit{eval}\ x'$ on the expression x' that results from evaluating the argument expression x. As a consequence, we can no longer use simple structural induction to calculate an abstract machine, but must use the more general technique of *rule induction* [14]. To this end, it is convenient to first reformulate the semantics in an explicit rule-based manner. We define an evaluation *relation* $\Downarrow \subseteq \mathit{Expr} \times \mathit{Env} \times \mathit{Value}$ by means of the following set of inference rules, which are obtained by rewriting the definition of *eval* in relational style:

$$\frac{e\ !!\ i\ \text{ is defined}}{\mathit{Var}\ i\ \Downarrow_e\ e\ !!\ i} \qquad\qquad \frac{}{\mathit{Abs}\ x\ \Downarrow_e\ \mathit{Clo}\ x\ e}$$

$$\frac{x\ \Downarrow_e\ \mathit{Clo}\ x'\ e' \qquad y\ \Downarrow_e\ u \qquad x'\ \Downarrow_{u:e'}\ v}{\mathit{App}\ x\ y\ \Downarrow_e\ v}$$

Note that *eval* is not a total function because (i) evaluation may fail to terminate, and (ii) looking up a variable in the environment may fail to return a value. The first cause of partiality is captured by the $\Downarrow$ relation: if $\mathit{eval}\ x\ e$ fails to terminate, then there is no value v with $x\ \Downarrow_e\ v$. The second cause is captured by the side-condition "$e\ !!\ i$ is defined" on the inference rule for $\mathit{Var}\ i$.

Specification

For the purposes of calculating an abstract machine based on the above semantics, the types for the desired new evaluation and execution functions remain essentially the same as those for arithmetic expressions,

$$\begin{aligned}
&\mathit{eval}'' :: \mathit{Expr} \to \mathit{CONT} \to \mathit{Int}\\
&\mathit{exec}\ \ :: \mathit{CONT} \to \mathit{Int} \to \mathit{Int}
\end{aligned}$$

except that eval'' now requires an environment to interpret free variables in its expression argument, and the value type is now *Value* rather than *Int*:

$$\begin{aligned}
&\mathit{eval}'' :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{Value}\\
&\mathit{exec}\ \ :: \mathit{CONT} \to \mathit{Value} \to \mathit{Value}
\end{aligned}$$

For arithmetic expressions, the desired behaviour of the abstract machine was specified by the equation $\mathit{eval}''\ x\ c = \mathit{exec}\ c\ (\mathit{eval}\ x)$. For lambda expressions, this equation needs to be modified to take account of the presence of an environment e, and the fact that the semantics is expressed by a relation $\Downarrow$ rather than a function *eval*. The resulting specification is as follows:

$$x \Downarrow_e v \quad \Rightarrow \quad \mathit{eval}''\ x\ e\ c = \mathit{exec}\ c\ v \tag{4}$$

Note that there is an important qualitative difference between the specification above and specification (3) for arithmetic expressions. The latter expresses soundness and completeness of the abstract machine, whereas the above only covers completeness, that is, every result produced by the semantics is also produced by the machine. But the specification does not rule out that *exec* terminates with a result for an expression that diverges according to the semantics. A separate argument about soundness has to be made. Bahr and Hutton [5] discuss this issue in more detail in the context of calculating compilers.

Calculation

Based upon specification (4), we now calculate definitions for the functions *eval*$''$ and *exec* by constructive rule induction on the assumption $x \Downarrow_e v$. In each case, we aim to rewrite the left-hand side $\mathit{eval}''\ x\ e\ c$ of the equation into a term t that does not refer to the evaluation relation $\Downarrow$, from which we can then conclude that the definition $\mathit{eval}''\ x\ e\ c = t$ satisfies the specification in this case. As in our first example, along the way we will find that we need to introduce new constructors into the *CONT* type, together with their interpretation by *exec*. The base cases for variables and abstractions are trivial.

Assuming $\mathit{Var}\ i \Downarrow_e e\,!!\,i$, we have that

$$\begin{array}{ll} & \mathit{eval}''\ (\mathit{Var}\ i)\ e\ c \\ = & \quad \{\ \text{specification (4)}\ \} \\ & \mathit{exec}\ c\ (e\,!!\,i) \end{array}$$

and assuming $\mathit{Abs}\ x \Downarrow_e \mathit{Clo}\ x\ e$, we have that

$$\begin{array}{ll} & \mathit{eval}''\ (\mathit{Abs}\ x)\ e\ c \\ = & \quad \{\ \text{specification (4)}\ \} \\ & \mathit{exec}\ c\ (\mathit{Clo}\ x\ e) \end{array}$$

In the case for $\mathit{App}\ x\ y \Downarrow_e v$, we may assume $x \Downarrow_e \mathit{Clo}\ x'\ e'$, $y \Downarrow_e u$ and $x' \Downarrow_{u:e'} v$ by the inference rule that defines the behaviour of $\mathit{App}\ x\ y$, together with induction hypotheses for the three component expressions x, y and x'. The calculation then proceeds by aiming to rewrite the term being manipulated into a form to which these induction hypotheses can be applied. Applying the induction hypothesis for x', i.e. $\mathit{eval}''\ x'\ (u\!:\!e')\ c' = \mathit{exec}\ c'\ v$, is straightforward:

$$\begin{array}{ll} & \mathit{eval}''\ (\mathit{App}\ x\ y)\ e\ c \\ = & \quad \{\ \text{specification (4)}\ \} \\ & \mathit{exec}\ c\ v \end{array}$$

$$\begin{array}{ll} = & \{ \text{ induction hypothesis for } x' \ \} \\ & \mathit{eval}''\ x'\ (u : e')\ c \end{array}$$

In turn, to apply the induction hypothesis for y, i.e. $\mathit{eval}''\ y\ e\ c' = \mathit{exec}\ c'\ u$, we need to rewrite the term $\mathit{eval}''\ x'\ (u : e')\ c$ into the form $\mathit{exec}\ c'\ u$ for some control stack c', i.e. we need to solve the equation:

$$\mathit{exec}\ c'\ u = \mathit{eval}''\ x'\ (u : e')\ c$$

As in our first example, the solution is to package the free variables x', e' and c in this equation up in the control stack argument c' by adding a new constructor to the CONT type that takes these variables as arguments

$$\mathit{FUN} :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{CONT}$$

adding a new equation for exec

$$\mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u = \mathit{eval}''\ x'\ (u : e')\ c$$

and then continuing the calculation:

$$\begin{array}{ll} & \mathit{eval}''\ x'\ (u : e')\ c \\ = & \{ \text{ define: } \mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u = \mathit{eval}''\ x'\ (u : e')\ c \ \} \\ & \mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u \\ = & \{ \text{ induction hypothesis for } y \ \} \\ & \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c) \end{array}$$

Finally, to apply the induction hypothesis for the expression x, i.e. $\mathit{eval}''\ x\ e\ c' = \mathit{exec}\ c'\ (\mathit{Clo}\ x'\ e')$, we need to solve the equation

$$\mathit{exec}\ c'\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c)$$

for which purposes we add another new constructor to the CONT type that takes the free variables y, e and c in this equation as arguments,

$$\mathit{ARG} :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{CONT}$$

add a new equation for exec

$$\mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c)$$

and then conclude as follows:

$$\begin{array}{ll} & \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c) \\ = & \{ \text{ define: } \mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c) \ \} \\ & \mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') \\ = & \{ \text{ induction hypothesis for } x \ \} \\ & \mathit{eval}''\ x\ e\ (\mathit{ARG}\ y\ e\ c) \end{array}$$

In a similar manner to the first example, we can then define the top-level evaluation function simply by applying eval'' to a nullary constructor HALT.

In summary, we have calculated the following definitions, which together form an abstract machine for evaluating lambda expressions:

```
data CONT where
  HALT :: CONT
  ARG  :: Expr → Env → CONT → CONT
  FUN  :: Expr → Env → CONT → CONT

eval''                         :: Expr → Env → CONT → Value
eval'' (Var i) e c             = exec c (e !! i)
eval'' (Abs x) e c             = exec c (Clo x e)
eval'' (App x y) e c           = eval'' x e (ARG y e c)

exec                           :: CONT → Value → Value
exec (ARG y e c) (Clo x' e')   = eval'' y e (FUN x' e' c)
exec (FUN x' e' c) u           = eval'' x' (u : e') c

eval                           :: Expr → Env → Value
eval x e                       = eval'' x e HALT
```

The names of the control stack commands are based on the intuition that *ARG* evaluates the argument of a function application, whereas *FUN* evaluates the function body. The resulting abstract machine coincides with the CEK machine [6]. We have also calculated abstract machines for lambda calculi with call-by-name and call-by-need semantics, which correspond to the Krivine machine [9] and a lazy variant of the Krivine machine derived by Ager et. al [3], respectively. The calculations can be found in the associated Coq proofs [7].

5 Summary and Conclusion

We presented the simple idea of applying fusion on the level of program calculations. Instead of first producing continuations via CPS transformation and then immediately turning them into data via defunctionalisation, we transform the original program in one go into the target program. Despite its conceptual simplicity, the benefits of this idea are considerable: the input program and the output program are linked *directly* to the desired output program via the specification that drives the calculation. This directness simplifies the calculation process and allows us to guide the process towards the desired output more easily. Moreover, by cutting out the continuations as the middleman, we avoid some of its complexities. For example, we have found that using CPS transformation for deriving compilers often yields non-strictly-positive datatypes, which makes the transformation unsuitable for formalisation in proof assistants. However, the non-strictly-positive datatypes disappear after defunctionalisation, and by fusing the two transformations we avoid the issue entirely.

To demonstrate the generality of the idea presented in this paper, we applied it to a variety of examples. The associated Coq proofs [7] contain formal calculations of abstract machines for different varieties of lambda calculi as well as languages with exception handling and state. The systematic nature of the calculations indicates the potential for automation of the entire derivation process with little or no interaction from the user apart from the specification.

Acknowledgements. We thank the anonymous reviewers for their comments and suggestions.

References

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (2003)
2. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From interpreter to compiler and virtual machine: a functional derivation. Technical report RS-03-14, BRICS, Department of Computer Science, University of Aarhus (2003)
3. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Inf. Process. Lett. **90**(5), 223–232 (2004)
4. Backhouse, R.: Program Construction: Calculating Implementations from Specifications. Wiley, New York (2003)
5. Bahr, P., Hutton, G.: Calculating correct compilers. J. Funct. Program. **25**, 47 (2015)
6. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III, pp. 193–217. Elsevier Science Publishers B.V., Amsterdam (1986)
7. Hutton, G., Bahr, P.: Associated Coq proofs. http://github.com/pa-ba/cps-defun
8. Hutton, G., Wright, J.: Calculating an exceptional machine. In: Loidl, H.W. (ed.) Trends in Functional Programming. Intellect, February 2006
9. Krivine, J.L.: Un Interpréteur du Lambda-calcul (1985). Unpublished manuscript
10. Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964)
11. Meijer, E.: Calculating compilers. Ph.D. thesis, Katholieke Universiteit Nijmegen (1992)
12. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference, pp. 717–740 (1972)
13. Wand, M.: Deriving target code as a representation of continuation semantics. ACM Trans. Program. Lang. Syst. **4**(3), 496–517 (1982)
14. Winskel, G.: The Formal Semantics of Programming Languages - An Introduction. Foundation of Computing Series. MIT Press, Cambridge (1993)

The Lambda Calculus: Practice and Principle

Hugh Leather[1](✉) and Janne Irgens[2]

[1] University of Edinburgh, Edinburgh, Scotland, UK
`hughleat@gmail.com`
[2] Number 15, Edinburgh, Scotland, UK
`jannefirgens@gmail.com`

Abstract. The Lambda Calculus has perplexed students of computer science for millennia, rendering many incapable of understanding even the most basic precepts of functional programming. This paper gently introduces the core concepts to the lay reader, assuming only a minimum of background knowledge in category theory, quantum chromodynamics, and paleomagnetism.

In addition, this paper goes on to its main results, showing how the Lambda Calculus can be used to easily prove the termination of Leibniz' Hailstone numbers for all $n > 0$, to show that matrix multiplication is possible in linear time, and to guarantee Scottish independence.

1 Introduction

The fascinating story of the discovery of the Lambda Calculus is well known to all school children, but we here recount the essentials of the story for completeness. In the 27^{th} century BC, Egyptian slaves were quarrying stone for the Pyramid of Djoser, northwest of Memphis. Buried deep in the sand, they unearthed a rare stele, complete with the full Lambda Calculus, written in hieroglyphics. At the time, none of the finest scholars of the ancient world could decipher the tablet and for many years it lay forgotten, a relic of a bygone era.

All that changed, however, when in 1929 Alonzo Church came upon the Djoser stele in the British Museum. In the decade before, Church had been working as a chorus girl in a music hall production of the Social Network during its long and infamous tour of the Middle East. There he had developed a passion for hieroglyphics and early Egyptian history. Thus it was that when he first laid eyes on the stone carvings, he instantly knew the significance of the markings and began fervently to decode them. He completed the work in 1932 with the paper, "A set of postulates for the foundation of logic". His article was an instant success, reaching number one on the New York Times best seller list for twelve weeks running, and earning him a much coveted Turing Award[1]. The rest, as they say, is history.

1.1 Lambda Calculus Defined

But, what is the Lambda Calculus? Despite the usual confusion that surrounds it, this dainty and delightful logic can be perfectly described in one short sentence:

"The square of the hypotenuse is the sum of the squares of the other two sides."

Or more formally:

$$E = mc^2 \tag{1}$$

Now that the reader is fully conversant with the calculus, the next section describes the importance of Lambda Calculus to circadian rhythm biology.

[1] This seriously annoyed Alan Turing who did not win a Turing Award. In response, Turing changed his name to Benedict Cumberbatch and won an Oscar. Oscar didn't really mind.

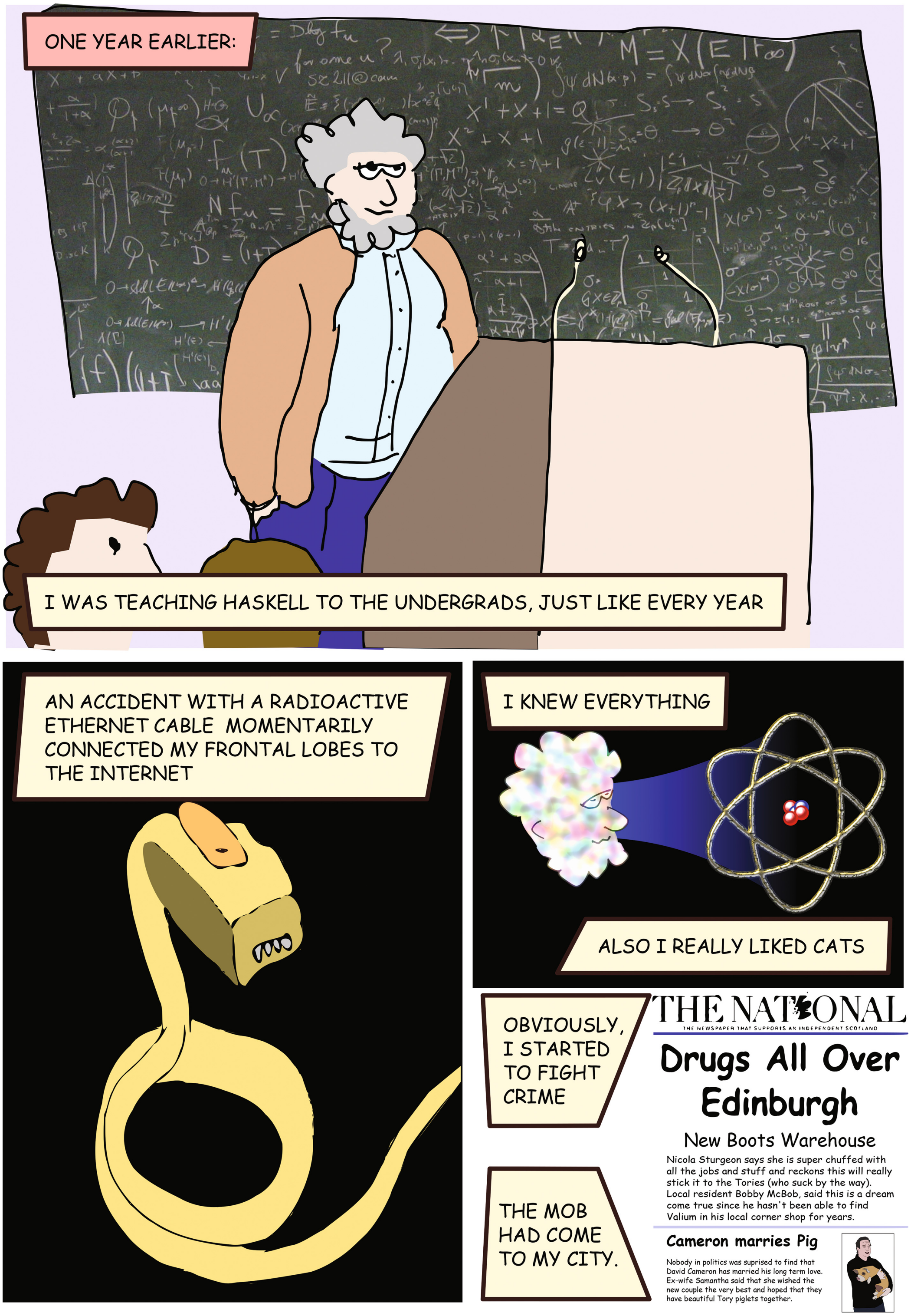
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 201–206, 2016.
DOI: 10.1007/978-3-319-30936-1_11

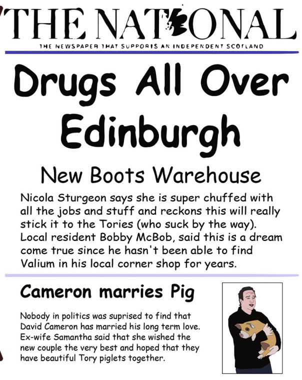
THE ADVENTURES OF
LAMBDA MAN
Origins
Story by Hugh Leather
Drawing by Janne Irgens

THIS IS MY CITY
Rosser
Church

MINE TO PROTECT

MINE TO WATCH OVER
BUT IT WASN'T ALWAYS THIS WAY


ONE YEAR EARLIER:
I WAS TEACHING HASKELL TO THE UNDERGRADS, JUST LIKE EVERY YEAR
AN ACCIDENT WITH A RADIOACTIVE ETHERNET CABLE MOMENTARILY CONNECTED MY FRONTAL LOBES TO THE INTERNET
I KNEW EVERYTHING
ALSO I REALLY LIKED CATS
OBVIOUSLY, I STARTED TO FIGHT CRIME
THE MOB HAD COME TO MY CITY.


THE NATIONAL

THE NEWSPAPER THAT SUPPORTS AN INDEPENDENT SCOTLAND

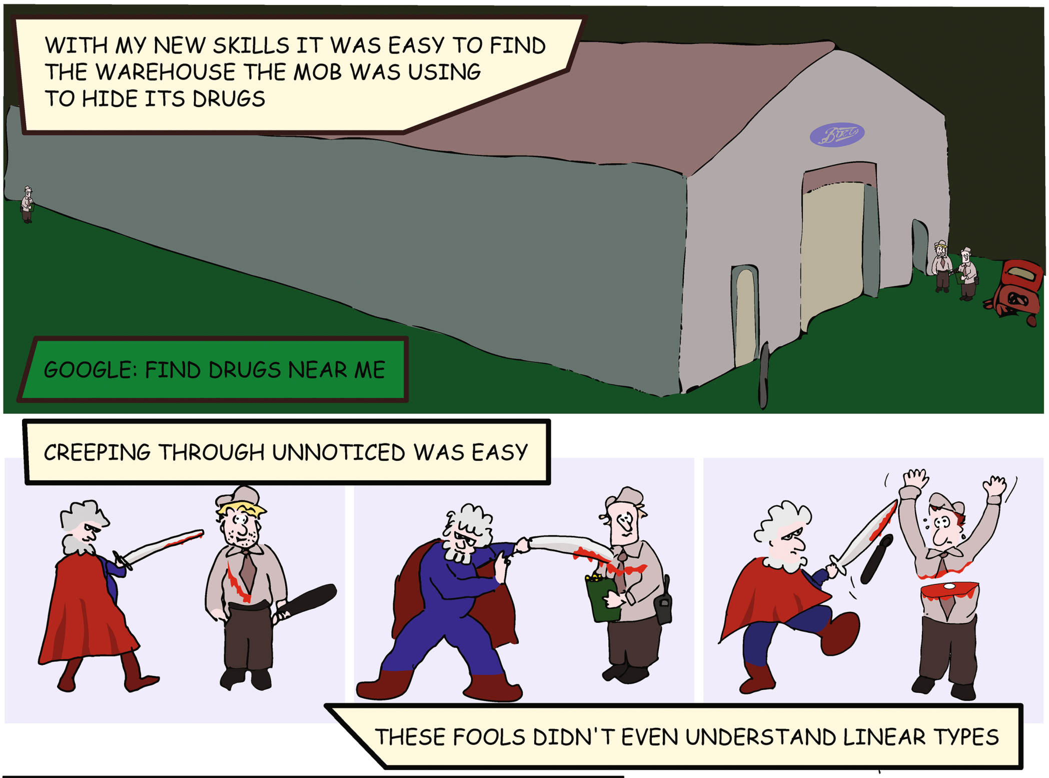
Drugs All Over Edinburgh

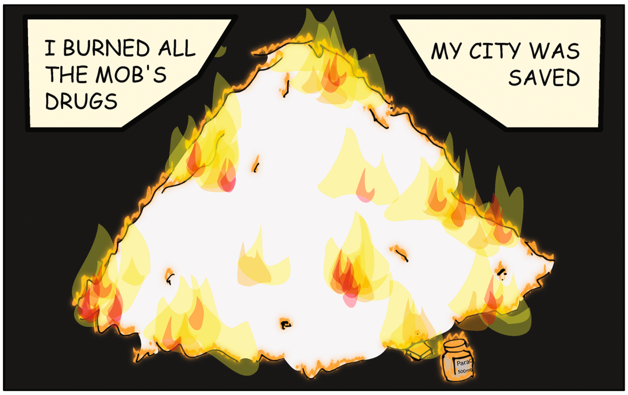
New Boots Warehouse

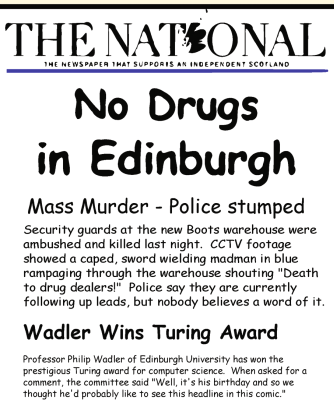
Nicola Sturgeon says she is super chuffed with all the jobs and stuff and reckons this will really stick it to the Tories (who suck by the way). Local resident Bobby McBob, said this is a dream come true since he hasn't been able to find Valium in his local corner shop for years.

Cameron marries Pig

Nobody in politics was suprised to find that David Cameron has married his long term love. Ex-wife Samantha said that she wished the new couple the very best and hoped that they have beautiful Tory piglets together.


WITH MY NEW SKILLS IT WAS EASY TO FIND THE WAREHOUSE THE MOB WAS USING TO HIDE ITS DRUGS
GOOGLE: FIND DRUGS NEAR ME
CREEPING THROUGH UNNOTICED WAS EASY
THESE FOOLS DIDN'T EVEN UNDERSTAND LINEAR TYPES




THE NATIONAL

THE NEWSPAPER THAT SUPPORTS AN INDEPENDENT SCOTLAND

No Drugs in Edinburgh

Mass Murder - Police stumped

Security guards at the new Boots warehouse were ambushed and killed last night. CCTV footage showed a caped, sword wielding madman in blue rampaging through the warehouse shouting "Death to drug dealers!" Police say they are currently following up leads, but nobody believes a word of it.

Wadler Wins Turing Award

Professor Philip Wadler of Edinburgh University has won the prestigious Turing award for computer science. When asked for a comment, the committee said "Well, it's his birthday and so we thought he'd probably like to see this headline in this comic."

DON'T MISS THE NEXT EPISODE: **LAMBDA MAN VS THE GOTO GUY** IN STORES 2026

2 Conclusion and Further Work

This paper has two major contributions. The first is that we are the first researchers to the best of our knowledge to have found a bonefide use for Comic Sans in a scientific paper[2]. The second, arguably more important contribution, is that we have wished Phil Wadler a very happy sixtieth birthday. In the future we hope that while we are exploring more uses for Comic Sans, that Phil has a jolly good time.

We would also strongly recommend that, no matter how much Phil wants to have a washboard stomach and to fight crime, he does not experiment trying to make a radioactive ethernet cable. And, under no circumstances should he go inserting ethernet cables anywhere other than ethernet sockets.

References

1. Barendregt, H.P.: The Lambda Calculus, vol. 3. North-Holland, Amsterdam (1984)
2. Bergson, H.: Laughter: An Essay on the Meaning of the Comic. Macmillan, New York (1914)
3. Bowman, J.L., Sakai, H., Jack, T., Weigel, D., Mayer, U., Meyerowitz, E.M.: Superman, a regulator of floral homeotic genes in arabidopsis. Development **114**(3), 599–615 (1992)
4. Church, A.: A set of postulates for the foundation of logic. Ann. Math. **33**, 346–366 (1932)
5. Clueless, J., Itworks, H.: 101 dubious ways to improve your h-index. In: Proceedings of the 1901 Conference on Unlikely Ways to Get Ahead (CUWGA 1901), January 1901
6. Collins, A., Fensch, C., Leather, H.: Auto-tuning parallel skeletons. Parallel Process. Lett. (PPL) **22**(2), 1240005-1–1240005-16 (2012)
7. Collins, A., Fensch, C., Leather, H.: Optimization space exploration of the fast-flow parallel skeleton framework. In: Proceedings of High-Level Programming for Heterogeneous and Hierarchical Parallel Systems, HLPGPU 2012, January 2012
8. Harris, J., Wonderwoman, S.: The Ethics of Human Biotechnology. Oxford University Press, Oxford (1992)
9. Hummert, S., Schuster, S., Hummert, C.: Batman - the dark knight: a game theoretical approach. In: Arabnia, H.R., Gravvanis, G.A., Solo, A.M.G. (eds.) FCS, pp. 29–34. CSREA Press, USA (2010)
10. Kyle, S., Bohm, I., Franke, B., Leather, H., Topham, N.: Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation. In: Proceedings of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2012, June 2012
11. Kyle, S., Leather, H., Franke, B., Butcher, D., Monteith, S.: Application of domain-aware binary fuzzing to aid android virtual machine testing. In: Proceedings of the 2015 International Conference on Virtual Execution Environments (VEE 2015), March 2015

[2] We acknowledge the pioneering work done by Simon Peyton-Jones rehabilitating this font in presentations. We also admit that "to the best of our knowledge" really means "we haven't the faintest idea and don't want to do any research into the matter". To the best of our knowledge we are the first to ever make such an admission.

I Got Plenty o' Nuttin'

Conor McBride[(✉)]

Mathematically Structured Programming,
University of Strathclyde, Glasgow, Scotland, UK
conor@strictlypositive.org

Abstract. Work to date on combining linear types and dependent types has deliberately and successfully avoided doing so. Entirely fit for their own purposes, such systems wisely insist that types depend only on the replicable sublanguage, thus sidestepping the issue of counting uses of limited-use data either within types or in ways which are only really needed to shut the typechecker up. As a result, the linear implication ('lollipop') stubbornly remains a non-dependent $S \multimap T$. This paper defines and establishes the basic metatheory of a type theory supporting a 'dependent lollipop' $(x : S) \multimap T[x]$, where what the input used to be is in some way commemorated by the type of the output. For example, we might convert list to length-indexed vectors *in place* by a function with type $(l : \mathsf{List}\ X) \multimap \mathsf{Vector}\ X\ (\mathsf{length}\ l)$. Usage is tracked with resource annotations belonging to an arbitrary rig, or 'riNg without Negation'. The key insight is to use the rig's zero to mark information in contexts which is present for purposes of contemplation rather than consumption, like a meal we remember fondly but cannot eat twice. We need no run-time copies of l to form the above vector type. We can have plenty of nothing with no additional runtime resource, and nothing is plenty for the construction of dependent types.

1 Introduction

Girard's linear logic [15] gives a discipline for parsimonious control of resources, and Martin-Löf's type theory [18] gives a discipline for precise specification of programs, but the two have not always seen eye-to-eye.

Recent times have seen attempts to change that. Pfenning blazed the trail in 2002, working with Cervesato on the linear logical framework [8], returning to the theme in 2011 to bring dependency on values to session types in joint work with Toninho and Caires [27]. Shi and Xi have linear typing in ATS [24]. Swamy and colleagues have linear typing in $F^\star$ [25]. Brady's Idris [6] has uniqueness typing, after the fashion of Clean [9]. Vákár has given a categorical semantics [28] where dependency is permitted on the cartesian sublanguage. Gaboardi and colleagues use linear types to study differential privacy [12]. Krishnaswami, Pradic and Benton [16] give a beautiful calculus based on a monoidal adjunction which relates a monoidal closed category of linear types to a cartesian closed category of intuitionistic dependent types.

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 207–233, 2016.
DOI: 10.1007/978-3-319-30936-1_12

Linear dependent types are a hot topic, but for all concerned bar me, linearity stops when dependency starts. The application rules (for twain they are, and never shall they meet) from Krishnaswami and coauthors illustrate the puzzle.

$$\frac{\Gamma; \Delta \vdash e : A \multimap B \quad \Gamma; \Delta' \vdash e' : A}{\Gamma; \Delta, \Delta' \vdash e\, e' : B} \qquad \frac{\Gamma; \Delta \vdash e : \Pi x{:}X.\, A \quad \Gamma \vdash e' : X}{\Gamma; \Delta \vdash e\, e' : A[e'/x]}$$

Judgments have an intuitionistic context, Γ, shared in each premise, and a linear context Δ, carved up between the premises. Later types in Γ may depend on earlier variables, so Γ cannot be freely permuted, but in order to distribute resources in the linear application, the linear context *must* admit permutation. Accordingly, types in Δ can depend on variables from Γ, but not on linear variables. How could they? When the linear context is chopped in two, some of the linear variables disappear! Accordingly, the argument in the dependent application is a purely intutionistic term, depending only on shared information, and the result type is thus well formed.

In this paper, I resolve the dilemma, with one idea: *nothing.* Contexts account for *how many* of each variable we have, and when we carve them up, we retain all the variables in each part but we split their quantities, so that we know of which we have none. That is, contexts with typed variables in common are *modules*, in the algebraic sense, with pointwise addition of resources drawn from some rig ('riNg without Negation'). Judgments account for how many of the given term are to be constructed from the resources in the context, and when we are constructing types, that quantity will be *zero*, distinguishing dynamic *consumption* from static *contemplation.* Correspondingly, we retain the ability to contemplate variables which stand for things unavailable for computation. My application rule (for there is but one) illustrates the point.

$$\frac{\Delta_0 \vdash \rho\ f \in (\pi\, x{:}S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ f\ s \in T[s{:}S/x]}$$

The function type is decorated with the 'unit price' π to be paid in copies of the input for each output required, so to make ρ outputs, our resource must be split between ρ functions and $\rho\pi$ arguments. The two contexts Δ_0 and Δ_1 have the same variables, even if some of them have zero resource, so the resulting type makes sense. If the ring has a 1, we may write $(1\, x{:}S) \to T$ as $(x{:}S) \multimap T$.

In summary, this paper contributes the definition of a type theory with uniform treatment of **unit-priced dependent function spaces** which is even unto its syntax **bidirectional**, in the sense of Pierce and Turner [22], resulting in a metatheoretic treatment of novel simplicity, including **type preservation** and a proof that **erasure** of all zero-resourced components retains type safety.

2 A Rig of Resources

Let us model resources with a rig, rather as Petricek, Orchard and Mycroft do, in their presentation of *coeffects* [21].

Definition 1 (Rig). *Let $\mathcal{R}$ be a set (whose elements are typically called ρ, π, ϕ), equipped with a value* 0*, an addition $\rho + \pi$, and a 'multiplication', $\phi\rho$, such that*

$$0 + \rho = \rho \qquad \rho + (\pi + \phi) = (\rho + \pi) + \phi \qquad \rho + \pi = \pi + \rho$$
$$\phi(\rho\pi) = (\phi\rho)\pi \qquad 0\rho = 0 = \rho 0 \qquad \phi(\rho + \pi) = \phi\rho + \phi\pi \qquad (\rho + \pi)\phi = \rho\phi + \pi\phi$$

I was brought up not to presume that a rig has a 1. Indeed, the trivial rig $\{0\}$ will give us the purely intuitionistic type theory we know and love. Moreover, every rig has the trivial sub-rig, a copy of intuitionistic type theory, in which we shall be able to construct objects (types, especially) whose runtime resource footprint is nothing but whose contemplative role allows more precise typechecking. The $\{0, 1\}$ rig gives a purely linear system, but the key example is 'none-one-tons'.

Definition 2 (None-One-Tons).

$\rho{+}\pi$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\rho\pi$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

The 0 value represents the resource required for usage only in types; 1 resources linear runtime usage; ω indicates relevant usage with no upper limit, or with weakening for ω (as treated in Sect. 12), *arbitrary* usage. With the latter in place, we get three recognizable quantifiers,

informally	$\forall x{:}S.\,T$	$(x{:}S) \multimap T$	$\Pi x{:}S.\,T$
formally	$(0\,x{:}S) \to T$	$(1\,x{:}S) \to T$	$(\omega\;x{:}S) \to T$

where $\forall$ is parametric, effectively an intersection rather than a product, with abstractions and applications erasable at runtime, but Π is *ad hoc* and thus runtime persistent. In making that valuable distinction, I learn from Miquel's Implicit Calculus of Constructions [20]: a dependent intersection collects only the functions which work uniformly for all inputs; a dependent product allows distinct outputs for distinguishable inputs.

Unlike Miquel, I shall remain explicit in the business of typechecking, but once a non-zero term has been checked, we can erase its zero-resourced substructures to obtain an untyped (for types are zero-resourced) λ-term which is the runtime version of it and computes in simulation with the unerased original, as we shall see in Sect. 11. Note that erasure relies on the absence of negation: zero-resourcing should not be an accident of 'cancellation'.

Although I give only the bare functional essentials, one can readily imagine an extension with datatypes and records, where the type of an *in place* sorting algorithm might be $(1\,xs{:}\mathsf{List\ Int}) \to \{1\,ys{:}\mathsf{OrderedList\ Int};\ 0\,p{:}ys \sim xs\}$.

We can, of course, contemplate many other variations, whether for usage analysis in the style of Wadler [30], or for modelling partial knowledge as Gaboardi and coauthors have done [12].

3 Syntax and Computation

The type theory I present is parametrised by its system of sorts (or ‘universes’). I keep it predicative: each sort is closed under quantification, but quantifying over ‘higher’ sorts is forbidden. My aim (for further work) is to support a straightforward normalization proof in the style championed by Abel [2]. Indeed, the resource annotations play no role in normalization, so erasing them and embedding this system into the predicative type theory in Abel’s habilitationsschrift [1] may well deliver the result, but a direct proof would be preferable.

Definition 3 (Sort Ordering). *Let $\mathcal{S}$ be a set of sorts (i, j, k) with a well-founded ordering $j \succ i$:* $\dfrac{k \succ j \quad j \succ i}{k \succ i}$ $\quad \dfrac{\forall j.\,(\forall i.\, j \succ i \to P[i]) \to P[j]}{\forall k.\, P[k]}$.

Asperti and the Matita team give a bidirectional ‘external syntax’ for the Calculus of Constructions [4], exploiting the opportunities it offers to omit type annotations. I have been working informally with bidirectional kernel type theories for about ten years, so a precise treatment is overdue. Here, the very syntax of the theory is defined by mutual induction between *terms*, with types specified in advance, and *eliminations* with types synthesized. Types are terms, of course.

Definition 4 (Term, Elimination).

$$\begin{array}{rll@{\qquad}rll}
R, S, T, s, t ::= & *_i & \textit{sort } i & e, f ::= & x & \textit{variable} \\
| & (\pi\, x\!:\!S) \to T & \textit{function type} & | & f\ s & \textit{application} \\
| & \lambda\, x.\, t & \textit{abstraction} & | & s\!:\!S & \textit{annotation} \\
| & \underline{e} & \textit{elimination} & & &
\end{array}$$

Sorts are explicit, and the function type is annotated with a ‘price’ $\pi \in \mathcal{R}$: the number of copies of the input required to compute each copy of the output. This bidirectional design ensures that λ need not have a domain annotation—that information always comes from the prior type. An elimination becomes a term without annotation: indeed, we shall have *two* candidates for its type.

In the other direction, a term can be used as an elimination only if we give a type at which to check it. Excluding type annotations from the syntax would force terms into β-normal form. Effectively, type annotations mark places where computation is unfinished—the ‘cuts’, in the language of logical calculi: we see the types of the ‘active’ terms. I plan neither to write nor to read mid-term type annotations, but rather to work with β-normal forms and typed *definitions*.

Syntactically, terms do not fit where variables go: we must either compute the β-redexes after substitution, as in the hereditary method introduced by Watkins and coauthors [32] and deployed to great effect by Adams [3], or we must find some other way to suspend computation in a valid form. By adding cuts to the syntax of eliminations, I admit a small-step characterization of reduction which allows us to approach the question of type preservation without first establishing β-normalization, which is exactly cut elimination in the sense of Gentzen [14].

The syntax is set up so that a redex pushes a cut towards its elimination. The β-rule replaces a redex *elimination* at a function type by redexes at the domain and range. The υ-rule strips the annotation from a fully computed *term*.

Definition 5 (Contraction, Reduction, Computation). Contraction *schemes are as follows:*

$$(\lambda\, x.\, t : (\pi\, x\!:\!S) \to T)\; s \leadsto_\beta (t\!:\!T)[s\!:\!S/x] \qquad\qquad \underline{t : T} \leadsto_\upsilon t$$

Closing $\leadsto_\beta$ *and* $\leadsto_\upsilon$ *under all one-hole contexts yields* reduction*,* $s \leadsto t$*. The reflexive-transitive closure of reduction is* computation*:* $\twoheadrightarrow = \leadsto^*$.

Let us defer the metatheory of computation and build more of the type theory.

4 Typing Rules

What we might be used to calling a *context* is here a *pre*context, Γ.

Definition 6 (Precontext, Context). $\Gamma ::= \cdot \mid \Gamma, x\!:\!S$.

A Γ*-context is a marking-up of* Γ *with a rig element for each variable.*

$$\frac{}{\cdot \textit{ is a } \mathrm{Cx}(\cdot)} \qquad \frac{\Delta \textit{ is a } \mathrm{Cx}(\Gamma)}{\Delta, \rho\, x\!:\!S \textit{ is a } \mathrm{Cx}(\Gamma, x\!:\!S)}$$

If Δ *is a* Γ*-context, we may take* $\lfloor\Delta\rfloor = \Gamma$.

It is my tidy habit to talk of "Γ-contexts", rather than slicing "contexts" by $\lfloor - \rfloor$, after the fact: Γ-contexts form an $\mathcal{R}$-module: addition $\Delta + \Delta'$ is pointwise, and multiplication $\phi\Delta$ premultiplies all the rig-annotations on variables in Δ by ϕ, keeping types the same. It is another (similar) habit to suppress the conditions required for wellformedness of expressions like $\Delta + \Delta'$: that is how I am telling you that Δ and Δ' are both Γ-contexts for some Γ.

Definition 7 (Prejudgment). *The prejudgment forms,* $\mathcal{P}$*, are as follows.*

type checking	$\Gamma \vdash T \ni t$
type synthesis	$\Gamma \vdash e \in S$

As with contexts, judgments decorate prejudgments with resources from $\mathcal{R}$. We may define a forgetful map $\lfloor - \rfloor$ from judgments to prejudgments.

Definition 8 (Judgment). *The judgment forms,* $\mathcal{J}$*, and* $\lfloor - \rfloor : \mathcal{J} \to \mathcal{P}$ *are given as follows.*

type checking	$\lfloor \Delta \vdash \rho\, T \ni t \rfloor = \lfloor\Delta\rfloor \vdash T \ni t$
type synthesis	$\lfloor \Delta \vdash \rho\, e \in S \rfloor = \lfloor\Delta\rfloor \vdash e \in S$

Let us say that t and e are, respectively, the *subjects* of these judgments. Let us further designate Δ and T *inputs* and S an *output*. My plan is to arrange the rules so that, presupposing input to a judgment makes sense, the subjects are validated and sensible output (if any) is synthesized. My policy is "garbage in; garbage out" and its benefit is that information flows as typing unfolds from subjects (once checked) to inputs and outputs, but never from inputs back to subjects. By restricting the interaction between the parts of the judgments in this way, I arrange a straightforward inductive proof of type preservation.

Readers familiar with traditional presentations may be surprised by the lack of a *context validity* judgment: I was surprised, too. Rather, we must now maintain the invariant that *if* a derivable judgment has valid input, every judgment in its derivation has valid input and output, and its own output is valid.

Any assignment of resources to a valid precontext Γ gives a context Δ. The typing judgments indicate that the resources in Δ are precisely enough to construct the given t or e with multiplicity ρ: it may help to think of ρ as 'how many runtime copies'. The types S and T consume no resources. We may use the derived judgment form $\Gamma \vdash_0 T \ni t$ to abbreviate the all-resources-zero judgment $0\Gamma \vdash 0\; T \ni t$ which is common when checking that types are well formed.

Definition 9 (Type Checking and Synthesis). *Type checking and synthesis are given by the following mutually inductive definition.*

$$\textsc{pre}\quad \frac{\Delta \vdash \rho\; R \ni t}{\Delta \vdash \rho\; T \ni t}\; T \leadsto R \qquad\qquad \textsc{post}\quad \frac{\Delta \vdash \rho\; e \in S}{\Delta \vdash \rho\; e \in R}\; S \leadsto R$$

$$\textsc{sort}\quad \frac{}{\Gamma \vdash_0 *_j \ni *_i}\; j \succ i \qquad\qquad \textsc{var}\quad \frac{}{0\Gamma, \rho\, x\!:\!S, 0\Gamma' \vdash \rho\; x \in S}$$

$$\textsc{fun}\quad \frac{\Gamma \vdash_0 *_i \ni S \qquad \Gamma, x\!:\!S \vdash_0 *_i \ni T}{\Gamma \vdash_0\; *_i \ni (\pi\, x\!:\!S) \to T}$$

$$\textsc{lam}\quad \frac{\Delta, \rho\pi\, x\!:\!S \vdash \rho\; T \ni t}{\Delta \vdash \rho\; (\pi\, x\!:\!S) \to T \ni \lambda\, x.\, t} \qquad\qquad \textsc{app}\quad \frac{\Delta_0 \vdash \rho\; f \in (\pi\, x\!:\!S) \to T \qquad \Delta_1 \vdash \rho\pi\; S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\; f\; s \in T[s\!:\!S/x]}$$

$$\textsc{elim}\quad \frac{\Delta \vdash \rho\; e \in S}{\Delta \vdash \rho\; T \ni \underline{e}}\; S \preceq T \qquad\qquad \textsc{cut}\quad \frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \qquad \Delta \vdash \rho\; S \ni s}{\Delta \vdash \rho\; s\!:\!S \in S}$$

Introduction forms require a type proposed in advance. It may be necessary to PRE-compute that type, e.g., to see it as a function type when checking a lambda. The SORT rule is an axiom, reflecting the presupposition that the context is valid already. Wherever the context is extended, the rules guarantee that the new variable has a valid type: in FUN, it comes from the subject and is checked; in LAM, it is extracted from an input. The FUN rule pushes sort demands inward rather than taking the max of inferred sorts, so it is critical that ELIM allows cumulative inclusion of sorts as we switch from construction to usage.

The VAR or CUT at the heart of an elimination drives the synthesis of its type, which we may need to POST-compute if it is to conform to its use. E.g., to use APP, we must first compute the type of the function to a function type, exposing its domain (to check the argument) and range (whence we get the output type).

Presuming context validity, VAR is an axiom enforcing the resource policy. Note how APP splits our resources between function and argument, but with the same *pre*context for both. The usual 'conversion rule' is replaced by ELIM sandwiched between POSTs and PREs computing two types to compatible forms.

Valid typing derivations form $\mathcal{R}$-modules. Whatever you make some of with some things, you can make none of with nothing—plenty for making types.

5 Dependent Iterative Demolition of Lists

Thus far, I have presented a bare theory of functions. Informally, let us imagine some data by working in a context. Underlines $\underline{e}$ help readability only of *meta*theory: I omit them, and sorts to boot. I telescope some function types for brevity. Let us have zero type constructors and plenty of value constructors.

$$0\,\mathsf{List} : (0\,X\!:\!*) \to *, \qquad \omega\ \mathsf{nil} : (0\,X\!:\!*) \to \mathsf{List}\,X,$$
$$\omega\ \mathsf{cons} : (0\,X\!:\!*, 1\,x\!:\!X, 1\,xs\!:\!\mathsf{List}\,X) \to \mathsf{List}\,X$$

Let us add a traditional *dependent* eliminator, resourced for iterative demolition.

$$\begin{aligned}\omega\ \mathsf{lit} : {} & (0\,X\!:\!*, 0\,P\!:\!(0\,x\!:\!\mathsf{List}\,X) \to *) \to \\ & (1\,n\!:\!P\,(\mathsf{nil}\,X)) \to \\ & (\omega\ c\!:\!(1\,x\!:\!X, 0\,xs\!:\!\mathsf{List}\,X, 1\,p\!:\!P\,xs) \to P\,(\mathsf{cons}\,X\,x\,xs)) \to \\ & (1\,xs\!:\,\mathsf{List}\,X) \to P\,xs\end{aligned}$$

The nil case is used once. In the cons case the tail xs has been reconsituted as p but remains contemplated in types. The intended computation rules conform.

$$\mathsf{lit}\,X\ P\ n\ c\,(\mathsf{nil}\,X) \rightsquigarrow n \qquad \mathsf{lit}\,X\ P\ n\ c\,(\mathsf{cons}\,X\ x\ xs) \rightsquigarrow c\,x\ xs\ (\mathsf{lit}\,X\ P\ n\ c\,xs)$$

Typechecking both sides of each rule, we see that n occurs once on both sides of each, but that c is dropped in one rule and duplicated in the other—some weakening is needed. Meanwhile, for cons, the tail xs has its one use in the recursive call but can still be given as the zero-resourced tail argument of c.

Some familiar list operations can be seen as simple demolitions:

$$\begin{aligned}\omega\ \mathsf{append}\ & : (0\,X\!:\!*, 1\,xs\!:\!\mathsf{List}\,X, 1\,ys\!:\!\mathsf{List}\,X) \to \mathsf{List}\,X \\ \mathsf{append}\ & = \lambda X.\ \lambda xs.\ \lambda ys.\,\mathsf{lit}\,X\ (\lambda\_.\ \mathsf{List}\,X)\ ys\ (\lambda x.\ \lambda\_.\ \lambda xs'.\,\mathsf{cons}\,X\ x\ xs')\ xs \\ \omega\ \mathsf{reverse}\ & : (0\,X\!:\!*, 1\,xs\!:\!\mathsf{List}\,X) \to \mathsf{List}\,X \\ \mathsf{reverse}\ & = \lambda X.\ \lambda xs.\,\mathsf{lit}\,X\ (\lambda\_.\,(1\,ys\!:\!\mathsf{List}\,X) \to \mathsf{List}\,X) \\ & \qquad (\lambda ys.\,ys)\ (\lambda x.\ \lambda\_.\ \lambda f.\ \lambda ys.\,f\ (\mathsf{cons}\,X\ x\ ys))\ xs\ (\mathsf{nil}\,X)\end{aligned}$$

Now let us have unary numbers for contemplation only, e.g. of vector length.

$$0\,\mathsf{Nat} : *, \qquad 0\,\mathsf{zero} : \mathsf{Nat}, \qquad 0\,\mathsf{suc}\!:\!(0\,n\!:\!\mathsf{Nat}) \to \mathsf{Nat}$$
$$0\,\mathsf{Vector} : (0\,X\!:\!*, 0\,n\!:\!\mathsf{Nat}) \to *, \qquad \omega\ \mathsf{vnil}\!:\!(0\,X\!:\!*) \to \mathsf{Vector}\,X\ \mathsf{zero},$$
$$\omega\ \mathsf{vcons}\!:\!(0\,X\!:\!*, 0\,n\!:\!\mathsf{Nat}, 1\,x\!:\!X, 1\,xs\!:\!\mathsf{Vector}\,X\ n) \to \mathsf{Vector}\,X\ (\mathsf{suc}\,n)$$

We can measure length statically, so it does not matter that we drop heads. Then, assuming length computes as above, let us turn lists to $\mathsf{vectors}$.

$$\begin{array}{rcl}
0\ \mathsf{length} & : & (0\, X : *, 0\, xs : \mathsf{List}\ X) \to \mathsf{Nat} \\
\mathsf{length} & = & \lambda\, X.\ \mathsf{lit}\ X\ (\lambda\, \_.\ \mathsf{Nat})\ \mathsf{zero}\ (\lambda\, x.\ \lambda\, \_.\ \mathsf{suc}) \\
\omega\ \mathsf{vector} & : & (0\, X : *, 1\, xs : \mathsf{List}\ X) \to \mathsf{Vector}\ X\ (\mathsf{length}\ xs) \\
\mathsf{vector} & = & \lambda\, X.\ \mathsf{lit}\ X\ (\lambda\, xs.\ \mathsf{Vector}\ X\ (\mathsf{length}\ xs)) \\
& & \qquad (\mathsf{vnil}\ X)\ (\lambda\, x.\ \lambda\, xs.\ \lambda\, xs'.\ \mathsf{vcons}\ X\ (\mathsf{length}\ xs)\ x\ xs')
\end{array}$$

In the step case, the zero-resourced list tail, xs, is used only to compute a zero-resourced length: the actual runtime tail is recycled as the vector xs'. The impact is to demolish a list whilst constructing a vector whose length depends on what the list used to be: that is the dependent lollipop in action.

6 Confluence of Computation

We shall need to establish the *diamond property* for computation.

Definition 10 (Diamond Property). *A binary relation R has the* diamond property *if* $\forall s, p, q.\ sRp \wedge sRq \Rightarrow \exists r.\ pRr \wedge qRr$.

Appealing to visual intuition, I will typically depict such propositions, $\begin{array}{ccc} s & R & p \\ R & \exists & R \\ q & R & r \end{array}$ where the existential quantifier governs points and proofs below and right of it.

The Tait–Martin-Löf–Takahashi method [26] is adequate to the task. We introduce a 'parallel reduction', $\rhd$, which amounts to performing none, any or all of the available contractions in a term, but no further redexes thus arising. Proving the diamond property for $\rhd$ will establish it for $\twoheadrightarrow$. Here is how.

Lemma 11 (Parallelogram). *Let R, P be binary relations with $R \subseteq P \subset R^*$. If P has the diamond property, then so has R^*.*

Proof. If sR^*p then for some m, sR^mp, hence also sP^mp. Similarly, for some n, sP^nq. We may now define a 'parallelogram' t_{ij} for $0 \le i \le m, 0 \le j \le n$, first taking two sides to be the P-sequences we have, $s = t_{00}Pt_{10}P \ldots Pt_{m0} = p$ and $s = t_{00}Pt_{01}P \ldots Pt_{0n} = q$, then applying the diamond property

$$\text{for } 0 \le i < m, 0 \le j < n \qquad \begin{array}{ccc} t_{ij} & P & t_{(i+1)j} \\ P & \exists & P \\ t_{i(j+1)} & P & t_{(i+1)(j+1)} \end{array}$$

Let $r = t_{mn}$. The lower two sides give pP^nr and qP^mr, so pR^*r and qR^*r. □

By design, parallel reduction fits Lemma 11. We have structural rules for each construct, together with υ and β rules.

Definition 12 (Parallel Reduction). *Let parallel reduction,* $\triangleright$*, be defined by mutual induction for terms and eliminations, as follows.*

$$\frac{}{*_i \triangleright *_i} \quad \frac{S \triangleright S' \quad T \triangleright T'}{(\pi\, x\!:\!S) \to T \triangleright (\pi\, x\!:\!S') \to T'} \quad \frac{t \triangleright t'}{\lambda\, x.\, t \triangleright \lambda\, x.\, t'} \quad \frac{e \triangleright e'}{\underline{e} \triangleright \underline{e'}} \quad \frac{t \triangleright t' \quad T \triangleright T'}{\underline{t\!:\!T} \triangleright t'}$$

$$\frac{}{x \triangleright x} \quad \frac{f \triangleright f' \quad s \triangleright s'}{f\ s \triangleright f'\ s'} \quad \frac{t \triangleright t' \quad T \triangleright T'}{t\!:\!T \triangleright t'\!:\!T'} \quad \frac{t \triangleright t' \quad S \triangleright S' \quad T \triangleright T' \quad s \triangleright s'}{(\lambda\, x.\, t : (\pi\, x\!:\!S) \to T)\ s \triangleright (t'\!:\!T')[s'\!:\!S'/x]}$$

Lemma 13 (Parallel Reduction Computes). $\rightsquigarrow \subseteq \triangleright \subseteq \twoheadrightarrow$.

Proof. Both inclusions are easy inductions on derivations. □

Crucially, parallel reduction commutes with substitution, because the latter duplicates or drops redexes, but never subdivides them.

Lemma 14 (Vectorized Substitution). *Admissibly,* $\dfrac{t \triangleright t' \quad \boldsymbol{e} \triangleright \boldsymbol{e}'}{t[\boldsymbol{e}/\boldsymbol{y}] \triangleright t'[\boldsymbol{e}'/\boldsymbol{y}]}$.

Proof. Proceed by structural induction on the derivation of $t \triangleright t'$. Effectively we are lifting a substitution on terms to a substitution on parallel reduction derivations. The only interesting cases are for variables and β-contraction.

For $y_i \triangleright y_i$ where e_i/y_i and e'_i/y_i are in the substitutions 'before' and 'after': here, we substitute the given derivation of $e_i \triangleright e'_i$.

For $(\lambda\, x.\, t : (\pi\, x\!:\!S) \to T)s \triangleright (t'\!:\!T')[s'\!:\!S'/x]$, where our 'argument' inductive hypotheses yield $(s\!:\!S)[\boldsymbol{e}/\boldsymbol{y}] \triangleright (s'\!:\!S')[\boldsymbol{e}'/\boldsymbol{y}]$, we may extend the term and derivation substitutions, then use 'body' hypotheses to deduce $t[(s\!:\!S)[\boldsymbol{e}/\boldsymbol{y}]/x, \boldsymbol{e}/\boldsymbol{y}] \triangleright t'[(s'\!:\!S')[\boldsymbol{e}'/\boldsymbol{y}]/x, \boldsymbol{e}'/\boldsymbol{y}]$ and similarly for T; the usual composition laws allow us to substitute in phases $-[(s\!:\!S)[\boldsymbol{e}/\boldsymbol{y}]/x, \boldsymbol{e}/\boldsymbol{y}] = -[s'\!:\!S'/x][\boldsymbol{e}/\boldsymbol{y}]$ and the β rule then yields $((\lambda\, x.\, t : (\pi\, x\!:\!S) \to T)\ s)[\boldsymbol{e}/\boldsymbol{y}] \triangleright (t' : T')[s'\!:\!S'/x][\boldsymbol{e}/\boldsymbol{y}]$. □

Iterating Lemma 14, we get that if $e \twoheadrightarrow e'$ and $t \twoheadrightarrow t'$, then $t[e/x] \twoheadrightarrow t'[e'/x]$.

Lemma 15 (Parallel Reduction Diamond).

$$\begin{array}{ccc} s & \triangleright & p \\ \triangledown & \exists & \triangledown \\ q & \triangleright & r \end{array}$$

Proof. We use induction on the derivation of $s \triangleright p$, then case analysis on the derivation of $s \triangleright q$. Where both use the same rule, the inductive hypotheses yield common parallel reducts. The only interesting cases are where computation is possible but one side declines the opportunity.

For υ, we have $\dfrac{s \triangleright p \quad S \triangleright P}{\underline{s\!:\!S} \triangleright p} \quad \dfrac{\dfrac{s \triangleright q \quad S \triangleright Q}{s\!:\!S \triangleright q\!:\!Q}}{\underline{s\!:\!S} \triangleright \underline{q\!:\!Q}}$.

Inductively, we obtain $\begin{array}{c} s \rhd p \\ \triangledown \; \exists \triangledown \\ q \rhd r \end{array}$ and $\begin{array}{c} S \rhd P \\ \triangledown \; \exists \; \triangledown \\ Q \rhd R \end{array}$, then also $\dfrac{q \rhd r \quad Q \rhd R}{\underline{q : Q} \rhd r}$.

For β, we have
$$\frac{s' \rhd p' \quad S \rhd P \quad S' \rhd P' \quad s \rhd p}{(\lambda\, x.\, s' : (\pi\, x{:}S) \to S')\; s \rhd (p' : P')[p : P/x]}$$

$$\frac{\begin{array}{cccc} s' \rhd q' & S \rhd Q & S' \rhd Q' & \\ \vdots & \vdots & \vdots & s \rhd q \end{array}}{(\lambda\, x.\, s' : (\pi\, x{:}S) \to S')\; s \rhd (\lambda\, x.\, q' : (\pi\, x{:}Q) \to Q')\; q}$$

Induction yields common reducts r', R, R', r, so $(p' : P')[p : P/x] \rhd (r' : R')[r : R/x]$ by Lemma 14, then the β rule gives us the required

$$\frac{q' \rhd r' \quad Q \rhd R \quad Q' \rhd R' \quad q \rhd r}{(\lambda\, x.\, q' : (\pi\, x{:}Q) \to Q')\; q \rhd (r' : R')[r : R/x]}$$ □

Corollary 16 (Confluence).

$$\begin{tikzcd} s \arrow[r, twoheadrightarrow] \arrow[d, twoheadrightarrow] \arrow[dr, phantom, "\exists"] & p \arrow[d, twoheadrightarrow] \\ q \arrow[r, twoheadrightarrow] & r \end{tikzcd}$$

Proof. Apply Lemma 11 with Lemmas 13 and 15. □

7 Subtyping and Its Metatheory

Subtyping is the co- and contravariant lifting of the universe ordering through the function type, where Luo's Extended Calculus of Constructions is covariant in the codomain and *equi*variant in the domain [17]. It is not merely convenient, but crucial in forming polymorphic types like $*_1 \ni (0\, X : *_0) \to (1\, x : X) \to X$, as the FUN rule demands $X : *_1$. To establish that cut elimination preserves typing, we must justify the subtyping in the ELIM rule with a proof of subsumption, i.e., that *term* can be lifted from sub- to supertype. While developing its proof, below, I had to adjust the definition to allow a little wiggle room where the application rule performs substitution: type annotations obstruct the proof. Rather than demanding cut elimination to get rid of them, I deploy wilful ignorance.

Definition 17 (Subtyping). *Let $\sim$ ('similarity') identify terms and eliminations structurally up to $s : S \sim s : S'$. Define subtyping inductively thus.*

$$\frac{S \sim T}{S \preceq T} \qquad \frac{j \succ i}{*_i \preceq *_j} \qquad \frac{S' \preceq S \quad T \preceq T'}{(\pi\, x{:}S) \to T \preceq (\pi\, x{:}S') \to T'}$$

In particular, subtyping is reflexive, so a term $\underline{e}$ is accepted if its *synthesized* type S and its checked type T have a common reduct.

Note that computation plays no role in subtyping: given that it is deployed at the 'change of direction', we can always use POST and PRE to compute as much as is needed to make this rather rigid syntactic definition apply. The rigidity then makes it easier to establish the crucial metatheoretic properties of subtyping.

Lemma 18 (Subtyping Transitivity). *Admissibly,* $\dfrac{R \preceq S \quad S \preceq T}{R \preceq T}$.

Proof. Contravariance in the rule for function types obliges us to proceed by induction on the maximum of the heights of the derivations (or, in effect, the 'size change' principle, for which subtyping is a paradigmatic example). If both derivations are by similarity, the result is by similarity. Otherwise, we have either sorts, in which case the transitivity of $\succ$ suffices, or function types, in which case the result follows from the inductive hypotheses. $\square$

We need two results about the interaction between subtyping and computation. If we compute one side, we can compute the other side to match, and if we compute both sides independently, we can compute further to reestablish subtyping. Both will depend on the corresponding fact about similarity.

Lemma 19 (Similarity Preservation). *Again, with $\exists$ applying below and right,*

$$\begin{tikzcd} S \arrow[r, phantom, "\sim"] \arrow[d, phantom, "\triangledown"] \arrow[dr, phantom, "\exists"] & T \arrow[d, phantom, "\triangledown"] \\ S' \arrow[r, phantom, "\sim"] & T' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, phantom, "\sim"] \arrow[d, phantom, "\triangledown"] & T \arrow[d, phantom, "\triangledown"] \\ S' \arrow[d, phantom, "\exists\ \triangledown"] & T' \arrow[d, phantom, "\triangledown"] \\ S'' \arrow[r, phantom, "\sim"] & T'' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, phantom, "\sim"] \arrow[d, two heads] \arrow[dr, phantom, "\exists"] & T \arrow[d, two heads] \\ S' \arrow[r, phantom, "\sim"] & T' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, phantom, "\sim"] \arrow[d, two heads] & T \arrow[d, two heads] \\ S' \arrow[d, two heads, "\exists"'] & T' \arrow[d, two heads] \\ S'' \arrow[r, phantom, "\sim"] & T'' \end{tikzcd}$$

Proof. For $\triangleright$, use copycat induction on derivations. If just one side computes, we need only compute the other. When both compute apart, we need to compute both back together, hence the far left $\exists$. For $\twoheadrightarrow$, we iterate the results for $\triangleright$. $\square$

Lemma 20 (Subtyping Preservation).

$$\begin{tikzcd} S \arrow[r, phantom, "\preceq"] \arrow[d, two heads] \arrow[dr, phantom, "\exists"] & T \arrow[d, two heads] \\ S' \arrow[r, phantom, "\preceq"] & T' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, phantom, "\preceq"] \arrow[d, two heads, "\exists"'] & T \arrow[r, two heads] & T' \arrow[d, equal] \\ S' \arrow[rr, phantom, "\preceq"] & & T' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, phantom, "\preceq"] \arrow[d, two heads] & T \arrow[d, two heads] \\ S' \arrow[d, two heads, "\exists"'] & T' \arrow[d, two heads] \\ S'' \arrow[r, phantom, "\preceq"] & T'' \end{tikzcd}$$

Proof. Induction on the derivation of $S \preceq T$. Lemma 19 covers similarity. For sorts, there is no way to compute. For function types, computation occurs only within sources and targets, so the inductive hypotheses deliver the result. $\square$

Lemma 21 (Subtyping Stability). $S \preceq T \Rightarrow S[r\!:\!R/x] \preceq T[r\!:\!R'/x]$.

Proof. Induction on derivations. Wherever R and R' occur distinctly, $\sim$ ignores. $\square$

The key result about subtyping is that it is justified by the admissibility of subsumption, pushing terms up the ordering. We may extend subtyping pointwise to contexts and establish the following rule, contravariant in contexts.

Theorem 22 (Subsumption). *If $\Delta' \preceq \Delta$, then admissibly,*

$$\frac{\Delta \vdash \rho\ S \ni s}{\Delta' \vdash \rho\ T \ni s}\ S \preceq T \qquad \frac{\Delta \vdash \rho\ e \in S}{\exists S'.S' \preceq S \wedge \Delta' \vdash \rho\ e \in S'}$$

Proof. We proceed by induction on typing derivations. For PRE, we make use of Lemma 20. We may clearly allow iterated PRE to advance types by $\twoheadrightarrow$, not just $\rightsquigarrow$. I write $\therefore$ to mark an appeal to the inductive hypothesis.

$$\frac{\Delta \vdash \rho\ R \ni t}{\Delta \vdash \rho\ S \ni t}\ S \rightsquigarrow R \qquad \begin{tikzcd} S \arrow[d, twoheadrightarrow] \arrow[r, phantom, "\preceq"] \arrow[dr, phantom, "\exists"] & T \arrow[d, twoheadrightarrow] \\ R \arrow[r, phantom, "\preceq"] & R' \end{tikzcd} \qquad \frac{\therefore \Delta' \vdash \rho\ R' \ni t}{\Delta' \vdash \rho\ T \ni t}$$

For SORT, transitivity of $\succ$ suffices. For FUN, we may pass the inflation of the desired sort through to the premises and appeal to induction.

For LAM, we have $\dfrac{S' \preceq S \quad T \preceq T'}{(\pi\, x\!:\!S) \to T \preceq (\pi\, x\!:\!S') \to T'} \quad \dfrac{\Delta, \rho\pi\, x\!:\!S \vdash \rho\ T \ni t}{\Delta \vdash \rho\ (\pi\, x\!:\!S) \to T \ni \lambda\, x.\, t}$. The contravariance of function subtyping allows us to extend the context with the source subtype and check the target supertype, $\dfrac{\therefore \Delta', \rho\pi\, x\!:\!S' \vdash \rho\ T' \ni t}{\Delta' \vdash \rho\ (\pi\, x\!:\!S') \to T' \ni \lambda\, x.\, t}$.

For ELIM, we have $\dfrac{\Delta \vdash \rho\ e \in R}{\Delta \vdash \rho\ S \ni \underline{e}}\ R \preceq S \quad S \preceq T$. Inductively, for some $R' \preceq R$ we have $\Delta' \vdash \rho\ e \in R'$ and by Lemma 18, we get $R' \preceq T$ and apply ELIM.

For POST, we may again appeal to Lemma 20. For VAR, we look up the subtype given by the contextual subtyping.

For APP, we have $\dfrac{\Delta_0 \vdash \rho\ f \in (\pi\, x\!:\!S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ f\ s \in T[s\!:\!S/x]}$. Given that Δ_0 and Δ_1 share a precontext, we have $\Delta'_0 \preceq \Delta_0$ and $\Delta'_1 \preceq \Delta_1$. Inductively, we may deduce in succession, $\therefore \exists S', T'.\ S \preceq S' \wedge T' \preceq T \wedge \quad \Delta'_0 \vdash \rho\ f \in (\pi\, x\!:\!S') \to T'$
$\therefore \Delta'_1 \vdash \rho\pi\ S' \ni s$
from which we obtain $\Delta'_0 + \Delta'_1 \vdash \rho\ f\ s \in T'[s\!:\!S'/x]$ where Lemma 21 gives, as required, $T'[s\!:\!S'/x] \preceq T[s\!:\!S/x]$. □

8 Not *That* Kind of Module System

Above, I claimed that Γ-contexts and typing deriviations yield $\mathcal{R}$-modules. Let us make that formal. Firstly, what is an $\mathcal{R}$-module?

Definition 23 ($\mathcal{R}$-module). *An $\mathcal{R}$-module is a set M with*

$$\begin{array}{rll} \textit{zero} & 0 & : M \\ \textit{addition} & - + - & : M \times M \to M \\ \textit{scalar multiplication} & - - & : \mathcal{R} \times M \to M \end{array}$$

which make $(M, 0, +)$ a commutative monoid and are compatible $\mathcal{R}$ in that, for all $m \in M$ $\ 0m = m$ $(\rho + \pi)m = \rho m + \pi m$ $(\rho\pi)m = \rho(\pi m)$.

The obvious (and, for us, adequate) way to form $\mathcal{R}$-modules is pointwise.

Lemma 24 (Pointwise $\mathcal{R}$-modules). *$X \to \mathcal{R}$ is an $\mathcal{R}$-module with*

$$0x = 0 \qquad (f+g)\,x = f\,x + g\,x \qquad (\rho f)\,x = \rho(f\,x).$$

Proof. Calculation with rig laws for $\mathcal{R}$. □

By taking $X = 0$ and, we get that $0 \to \mathcal{R} \cong 1$ is an $\mathcal{R}$-module. By taking $X = 1$, we get that $1 \to R \cong \mathcal{R}$ itself is an $\mathcal{R}$-module.

Lemma 25 (Contexts $\mathcal{R}$-modules). *Γ-contexts form an $\mathcal{R}$-module.*

Proof. The Γ-contexts, Δ are given by functions $\Delta|- : \mathrm{dom}(\Gamma) \to \mathcal{R}$, where $(\Delta, \rho\, x\!:\!S, \Delta')|x = \rho$. Lemma 24 applies. □

Where the 'coeffects' treatment of resources from Petricek and coauthors [21] retains the non-dependent mode of splitting the context—some variables into one premise, the rest in the other—the potential for type dependency forces more rigidity upon us. The module structure of Γ-contexts lets us send Γ to all premises, splitting up the resources the context gives each of Γ's variables.

What about typing derivations? We can play the same game. Let $\mathcal{T}(X)$ be the set of finitely branching trees whose nodes are labelled with elements of X. The typing rules tell us which elements D, of $\mathcal{T}(\mathcal{J})$ constitute valid deductions of the judgment at the root. We can separate such a tree into a *shape* and a set of *positions*. The elements of $\mathcal{T}(\mathcal{J})$ with a given shape form a module by lifting $\mathcal{R}$ pointwise over positions. That is but a dull fact about syntax, but more interesting is that the module can then be restricted to *valid* derivations.

Definition 26 (Shape and Positions). *Shapes, d, of derivations inhabit trees $\mathcal{T}(\mathcal{P})$ of prejudgments. The shape of a given derivation is given by taking $\lfloor - \rfloor : \mathcal{T}(\mathcal{J}) \to \mathcal{T}(\mathcal{P})$ to be the functorial action $\mathcal{T}(\lfloor - \rfloor)$ which replaces each judgment with its corresponding prejudgment. Position sets,* $\mathrm{Pos} : \mathcal{T}(\mathcal{P}) \to \mathbf{Set}$ *and prejudgment positions* $\mathrm{Pos}' : \mathcal{P} \to \mathbf{Set}$ *are given structurally:*

$$\mathrm{Pos}\left(\frac{d_1 \ldots d_n}{P}\right) = \mathrm{Pos}'(P) + \sum_i \mathrm{Pos}(d_i) \qquad \begin{aligned} \mathrm{Pos}'(\Gamma \vdash) &= 0 \\ \mathrm{Pos}'(\Gamma \vdash T \ni t) &= \mathrm{dom}(\Gamma) + 1 \\ \mathrm{Pos}'(\Gamma \vdash e \in S) &= \mathrm{dom}(\Gamma) + 1 \end{aligned}$$

where 1 is the unit type with element $\star$.

That is, each typing prejudgment has a resource position for each quantity in its context and for the number of things to be constructed. A straightforward recursive labelling strategy then yields the following.

Lemma 27 (Representing Derivations).

$$\forall d \in \mathcal{T}(\mathcal{P})$$

$$\{D : \mathcal{T}(\mathcal{J}) \mid \lfloor D \rfloor = d\} \cong \mathrm{Pos}(d) \to \mathcal{R}.$$

Proof. Structural induction. At each node, $\{J : \mathcal{J} \mid \lfloor J \rfloor = P\} \cong \mathrm{Pos}'(P) \to \mathcal{R}$.

P	J	$\mathrm{Pos}'(P) \to \mathcal{R}$
$\Gamma \vdash$	$\Gamma \vdash$	$\_$
$\Gamma \vdash T \ni t$	$\Delta \vdash \rho\, T \ni t$	$x \mapsto \Delta\vert x$; $\star \mapsto \rho$
$\Gamma \vdash e \in S$	$\Delta \vdash \rho\, e \in S$	$x \mapsto \Delta\vert x$; $\star \mapsto \rho$

□

The derivation trees of shape d thus form an unremarkable $\mathcal{R}$-module. Let us establish something a touch more remarkable. In fact it is obvious, because when designing the system, I took care to ensure that any nonzero resource demand in the conclusion of each rule is linearly a factor of the demands in the premises.

Theorem 28 (Valid Derivation Modules). *For any* valid *derivation tree D of shape d, the $\mathcal{R}$-module on $\{D' : \mathcal{T}(\mathcal{J}) \mid \lfloor D' \rfloor = d\}$ refines to an $\mathcal{R}$-module on $\{D' : \mathcal{T}(\mathcal{J}) \mid \lfloor D' \rfloor = d, D'$* valid*$\}$.*

Proof. It is necessary and sufficient to check closure under addition and scalar multiplication as the latter gives us that $0D$ is a valid zero. The proof is a straightforward induction on d, then inversion of the rules yielding the conclusion. For scalar multiplication, I give the cases for var, app and cut, as they give the pattern for the rest, showing local module calculuations by writing true equations in places where one side is given and the other is needed.

$$\phi\left(\frac{\Gamma, x\!:\!S, \Gamma' \vdash}{0\Gamma, \rho\, x\!:\!S, 0\Gamma' \vdash \rho\; x \in S}\right) = \frac{\Gamma, x\!:\!S, \Gamma' \vdash}{0\Gamma, \phi\rho\, x\!:\!S, 0\Gamma' \vdash \phi\rho\; x \in S}$$

$$\phi\left(\frac{\begin{array}{c}\Delta \vdash \rho\; f \in (\pi\, x\!:\!S) \to T \\ \Delta' \vdash \rho\pi\; S \ni s\end{array}}{\Delta + \Delta' \vdash \rho\; f\; s \in T[s\!:\!S/x]}\right) = \frac{\begin{array}{c}\phi\Delta \vdash \phi\rho\; f \in (\pi\, x\!:\!S) \to T \\ \phi\Delta' \vdash (\phi\rho)\pi\; S \ni s\end{array}}{\phi\Delta + \phi\Delta' \vdash \phi\rho\; f\; s \in T[s\!:\!S/x]}$$

$$\phi\left(\frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\; S \ni s}{\Delta \vdash \rho\; s\!:\!S \in S}\right) = \frac{\lfloor\phi\Delta\rfloor \vdash_0 *_i \ni S \quad \phi\Delta \vdash \phi\rho\; S \ni s}{\phi\Delta \vdash \phi\rho\; s\!:\!S \in S}$$

For addition, I give just the app case, which makes essential use of commutativity of the rig's addition and distributivity of multplication over addition.

$$\frac{\begin{array}{c}\Delta_0 \vdash \rho_0\; f \in (\pi\, x\!:\!S) \to T \\ \Delta_0' \vdash \rho_0\pi\; S \ni s\end{array}}{\Delta_0 + \Delta_0' \vdash \rho_0\; f\; s \in T[s\!:\!S/x]} + \frac{\begin{array}{c}\Delta_1 \vdash \rho_1\; f \in (\pi\, x\!:\!S) \to T \\ \Delta_1' \vdash \rho_1\pi\; S \ni s\end{array}}{\Delta_1 + \Delta_1' \vdash \rho_1\; f\; s \in T[s\!:\!S/x]}$$

$$= \frac{\begin{array}{c}\Delta_0 + \Delta_1 \vdash (\rho_0 + \rho_1)\; f \in (\pi\, x\!:\!S) \to T \\ \Delta_0' + \Delta_1' \vdash (\rho_0 + \rho_1)\pi\; S \ni s\end{array}}{(\Delta_0 + \Delta_1) + (\Delta_0' + \Delta_1') \vdash (\rho_0 + \rho_1)\; f\; s \in T[s\!:\!S/x]}$$

Hence, valid derivations form an $\mathcal{R}$-module. □

Not only can we multiply by scalars and add. We can also pull out common factors and split up our resources wherever they make multiples and sums.

Lemma 29 (Factorization). *If $\Delta \vdash \phi\rho\ T \ni t$ then for some context named Δ/ϕ, $\Delta = \phi(\Delta/\phi)$ and $\Delta/\phi \vdash \rho\ T \ni t$. Similarly, if $\Delta \vdash \phi\rho\ e \in S$ then for some Δ/ϕ, $\Delta = \phi(\Delta/\phi)$ and $\Delta/\phi \vdash \rho\ e \in S$.*

Proof. Induction on derivations. The only interesting case is APP.

$$\frac{\Delta_0 \vdash \phi\rho\ f \in (\pi\, x\!:\!S) \to T \quad \Delta_1 \vdash (\phi\rho)\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \phi\rho\ f\ s \in T[s\!:\!S/x]}$$

Inductively, $\Delta_0/\phi \vdash \rho\ f \in (\pi\, x\!:\!S) \to T$, and reassociating, $\Delta_1/\phi \vdash \rho\pi\ S \ni s$, so distribution gives us $\phi(\Delta_0/\phi + \Delta_1/\phi) \vdash \phi\rho\ f\ s \in T[s\!:\!S/x]$. □

This result does not mean $\mathcal{R}$ has multiplicative inverses, just that to make ϕ things at once, our supplies must come in multiples of ϕ, especially when $\phi = 0$.

Corollary 30 (Nothing from Nothing). *If $\Delta \vdash 0\ T \ni t$ then $\Delta = 0\lfloor\Delta\rfloor$.*

Proof. Lemma 29 with $\phi = \rho = 0$. □

Lemma 31 (Splitting). *If $\Delta \vdash (\phi + \rho)\ T \ni t$ then for some $\Delta = \Delta' + \Delta''$, $\Delta' \vdash \phi\ T \ni t$ and $\Delta' \vdash \rho\ T \ni t$. Similarly, if $\Delta \vdash (\phi + \rho)\ e \in S$ then for some $\Delta = \Delta' + \Delta''$, $\Delta' \vdash \phi\ e \in S$ and $\Delta'' \vdash \rho\ e \in S$.*

Proof. Induction on derivations. The only interesting case is APP.

$$\frac{\Delta_0 \vdash (\phi + \rho)\ f \in (\pi\, x\!:\!S) \to T \quad \Delta_1 \vdash (\phi + \rho)\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash (\phi + \rho)\ f\ s \in T[s\!:\!S/x]}$$

Inductively, $\Delta'_0 \vdash \phi\ f \in (\pi\, x : S) \to T$ and $\Delta''_0 \vdash \rho\ f \in (\pi\, x : S) \to T$, and distributing, $\Delta'_1 \vdash \phi\pi\ S \ni s$ and $\Delta''_1 \vdash \rho\pi\ S \ni s$, so $\Delta'_0 + \Delta'_1 \vdash \phi\pi\ f\ s \in T[s\!:\!S/x]$ and $\Delta''_0 + \Delta''_1 \vdash \rho\pi\ f\ s \in T[s\!:\!S/x]$. □

9 Resourced Stability Under Substitution

Let us establish that basic thinning and substitution operations lift from syntax (terms and eliminations) to judgments (checking and synthesis). It may seem peculiar to talk of thinning in a precisely resourced setting, but as the *pre*context grows, the context will show that we have *zero* of the extra things.

Notationally, it helps to define *localization* of judgments to contexts, in that it allows us to state properties of derivations more succinctly.

Definition 32 (Localization). *Define*

$$\begin{array}{ll} - \vdash - : \mathrm{Cx}(\Gamma) \times \mathcal{J} \to \mathcal{J} & \\ \Delta \vdash \ \Delta' \vdash \rho\, T \ni t & = \Delta, \Delta' \vdash \rho\, T \ni t \\ \Delta \vdash \ \Delta' \vdash \rho\, e \in S & = \Delta, \Delta' \vdash \rho\, e \in S \end{array}$$

Strictly speaking, I should take care when localizing $\Delta \vdash \mathcal{J}$ to freshen the names in $\mathcal{J}$'s local context relative to Δ. For the sake of readability, I shall presume that accidental capture does not happen, rather than freshening explicitly.

Lemma 33 (Thinning). *Admissibly,* $\dfrac{\Delta \vdash \mathcal{J}}{\Delta, 0\Gamma \vdash \mathcal{J}}$.

Proof. Induction on derivations, with $\mathcal{J}$ absorbing local extensions to the context, so the inductive hypothesis applies under binders. We can thus replay the input derivation with 0Γ inserted. In the APP case, we need that $0\Gamma + 0\Gamma = 0\Gamma$. In the VAR case, inserting 0Γ preserves the applicability of the rule. □

Lemma 34 (Substitution Stability). *Admissibly,* $\dfrac{\Delta, \phi\, x\!:\!S \vdash \mathcal{J} \Delta' \vdash \phi\, e \in S}{\Delta + \Delta' \vdash \mathcal{J}[e/x]}$.

Proof. Induction on derivations, effectively substituting a suitable $\Delta', 0\Gamma \vdash \phi\ e \in S$ for every usage of the VAR rule at some $\lfloor \Delta \rfloor, \phi\, x : S, 0\Gamma \vdash \phi\ x \in S$. Most cases are purely structural, but the devil is in the detail of the resourcing, so let us take account. For PRE, LAM, ELIM and POST, the resources in Δ are undisturbed and the induction goes through directly. For SORT and FUN, $\Delta = 0\Gamma$, and Corollary 30 tells us that $\phi = 0$ and hence $\Delta' = 0\Gamma$, pushing the induction through. For VAR with variables other than x, again, $\phi = 0$ and the induction goes through. For VAR at x itself, Lemma 33 delivers the correct resource on the nose. For CUT, we may give all of the es to the term and (multiplying by 0, thanks to Theorem 28) exactly none of them to its type. In the APP case, Lemma 31 allows us to share out our es in exactly the quantities demanded in the premises. □

10 Computation Preserves Typing

The design of the system allows us to prove that computation in all parts of a judgment preserves typing: inputs never become *subjects* at any point in the derivation. While following the broad strategy of 'subject reduction' proofs, exemplified by McKinna and Pollack [19], the result comes out in one delightfully unsubtle dollop exactly because information flows uniformly through the rules.

We can lift $\twoheadrightarrow$ to contexts, simply by permitting computation in types, and we can show that any amount of computation in judgment inputs, and a parallel reduction in the subject, preserves the derivability of judgments upto computation in outputs. We should not expect computation to preserve the types we can synthesize: if you reduce a variable's type in the context, you should not expect to *synthesize* the unreduced type, but you can, of course, still *check* it.

Theorem 35 (Parallel Preservation).

$$\begin{array}{ccc}\Delta & T & t\\ \downarrow & \downarrow & \triangledown\\ \Delta' & T' & t'\end{array} \Rightarrow \frac{\Delta\vdash\rho\,T\ni t}{\Delta'\vdash\rho\,T'\ni t'} \qquad \begin{array}{cc}\Delta & e\\ \downarrow&\triangledown\\ \Delta'&e'\end{array}\Rightarrow \exists\begin{array}{c}S\\ \downarrow\\ S'\end{array}\wedge\frac{\Delta\vdash\rho\,e\in S}{\Delta'\vdash\rho\,e'\in S'}$$

Proof. We proceed by induction on derivations and inversion of $\triangleright$. Let us work through the rules in turn.

Type Checking. For pre, we have

$$\frac{\Delta\vdash\rho\,R\ni t}{\Delta\vdash\rho\,T\ni t}\qquad \begin{array}{cc}\Delta & t\\ \downarrow&\triangledown\\ \Delta'&t'\end{array}\quad\begin{array}{ccc}T & \rightsquigarrow & R\\ \downarrow&\exists&\downarrow\\ T'&\twoheadrightarrow&R'\end{array}\qquad \frac{\therefore\ \Delta'\vdash\rho\,R'\ni t'}{\Delta'\vdash\rho\,T'\ni t'}$$

with the confluence of computation telling me how much computation to do to R if I want T' to check t'. For sort, subject and checked type do not reduce, but one axiom serves as well as another.

$$\text{given } \Gamma\vdash_0 *_j\ni *_i \quad j\succ i \quad \Gamma\twoheadrightarrow\Gamma' \quad \text{deduce } \Gamma'\vdash_0 *_j\ni *_i$$

For fun and lam, respectively, we must have had

$$\frac{\begin{array}{c}\Gamma\vdash_0 *_i\ni S\\ \Gamma,x\!:\!S\vdash_0 *_i\ni T\end{array}}{\Gamma\vdash_0\ *_i\ni(\pi\,x\!:\!S)\to T}\quad \begin{array}{ccc}\Gamma & S & T\\ \downarrow&\triangledown&\triangledown\\ \Gamma'&S'&T'\end{array}\quad \frac{\begin{array}{c}\therefore\ \Gamma'\vdash_0 *_i\ni S'\\ \therefore\ \Gamma',x\!:\!S'\vdash_0 *_i\ni T'\end{array}}{\Gamma'\vdash_0\ *_i\ni(\pi\,x\!:\!S')\to T'}$$

$$\frac{\Delta,\rho\pi\,x\!:\!S\vdash\rho\,T\ni t}{\Delta\vdash\rho\,(\pi\,x\!:\!S)\to T\ni\lambda\,x.\,t}\quad \begin{array}{cccc}\Gamma & S & T & t\\ \downarrow&\downarrow&\downarrow&\triangledown\\ \Gamma'&S'&T'&t'\end{array}\quad \frac{\therefore\ \Delta',\rho\pi\,x\!:\!S'\vdash\rho\,T'\ni t'}{\Delta'\vdash\rho\,(\pi\,x\!:\!S')\to T'\ni\lambda\,x.\,t'}$$

For elim, we have two cases. For the structural case, we must compute.

$$\frac{\Delta\vdash\rho\,e\in S}{\Delta\vdash\rho\,T\ni\underline{e}}\ S\preceq T\quad \begin{array}{ccc}\Delta & T & e\\ \downarrow&\downarrow&\triangledown\\ \Delta'&T'&e'\end{array}\quad \therefore\ \exists\begin{array}{c}S\\ \downarrow\\ S'\end{array}\wedge\ \Delta'\vdash\rho\,e'\in S'$$

$$\begin{array}{ccc} & S\ \preceq & T\\ & \downarrow & \downarrow\\ & S' & T'\\ \exists & \downarrow & \downarrow\\ & S''\preceq & T''\end{array}\qquad \cfrac{\cfrac{\cfrac{\Delta'\vdash\rho\,e'\in S'}{\Delta'\vdash\rho\,e'\in S''}\ S'\twoheadrightarrow S''}{\Delta'\vdash\rho\,T''\ni\underline{e'}}\ S''\preceq T''}{\Delta'\vdash\rho\,T'\ni\underline{e'}}\ T'\twoheadrightarrow T''$$

Lemma 20 reestablishes subtyping after computation.

For υ-contraction, we have a little more entertainment. We start with

$$\frac{\begin{array}{c}\Delta\vdash\rho\,S\ni s\\ \cdots\\ \Delta\vdash\rho\,s\!:\!S\in S'\end{array}}{\Delta\vdash\rho\,T\ni\underline{s\!:\!S}}\quad \begin{array}{cc}\Delta & s\\ \downarrow&\triangledown\\ \Delta'&s'\end{array}\quad \begin{array}{cccc} & S & \preceq & T\\ & \downarrow & & \downarrow\\ & S' & & T'\\ \exists & \downarrow & & \downarrow\\ & S'' & \preceq & T''\end{array}\quad \therefore\ \Delta'\vdash\rho\,S''\ni s'$$

Then by Theorem 22 (subsumption), we obtain $\dfrac{\dfrac{\Delta' \vdash \rho\, S'' \ni s'}{\Delta' \vdash \rho\, T'' \ni s'}\; S'' \preceq T''}{\Delta' \vdash \rho\, T' \ni s'}\; T' \twoheadrightarrow T''$.

Type Synthesis. For POST, we have

$$\frac{\Delta \vdash \rho\; e \in S}{\Delta \vdash \rho\; e \in R} \qquad \begin{array}{cccc} \Delta & e & S \rightsquigarrow & R \\ \downarrow & \triangledown & \exists\downarrow & \exists\downarrow \\ \Delta' & e' & S' \twoheadrightarrow & R' \end{array} \qquad \frac{\therefore \Delta' \vdash \rho\; e' \in S'}{\Delta' \vdash \rho\; e' \in R'}$$

For VAR, again, one axiom is as good as another

$$\frac{}{0\Gamma_0, \rho\, x\!:\!S, 0\Gamma_1 \vdash \rho\; x \in S} \qquad \begin{array}{ccc} \Gamma_0 & S & \Gamma_1 \\ \downarrow & \downarrow & \downarrow \\ \Gamma_0' & S' & \Gamma_1' \end{array} \qquad \frac{}{0\Gamma_0', \rho\, x\!:\!S', 0\Gamma_1' \vdash \rho\; x \in S'}$$

The case of CUT is just structural.

$$\frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\; S \ni s}{\Delta \vdash \rho\; s\!:\!S \in S} \qquad \begin{array}{ccc} \Delta & S & s \\ \downarrow & \triangledown & \triangledown \\ \Delta' & S' & s' \end{array} \qquad \frac{\therefore \lfloor\Delta'\rfloor \vdash_0 *_i \ni S' \quad \therefore \Delta' \vdash \rho\; S' \ni s'}{\Delta' \vdash \rho\; s'\!:\!S' \in S'}$$

For APP, we have two cases. In the structural case, we begin with

$$\frac{\Delta_0 \vdash \rho\; f \in (\pi\, x\!:\!S) \to T \quad \Delta_1 \vdash \rho\pi\; S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\; f\; s \in T[s\!:\!S/x]} \qquad \begin{array}{ccccc} \Delta_0 & + & \Delta_1 & f & s \\ \downarrow & & \downarrow & \triangledown & \triangledown \\ \Delta_0' & + & \Delta_1' & f' & s' \end{array}$$

and we should note that the computed context $\Delta_0' + \Delta_1'$ continue to share a common (but more computed) precontext. The inductive hypothesis for the function tells us the type at which to apply the inductive hypothesis for the argument. We obtain $\begin{array}{cc} S & T \\ \exists\downarrow & \exists\downarrow \\ S' & T' \end{array}$ $\dfrac{\therefore \Delta_0' \vdash \rho\; f \in (\pi\, x\!:\!S') \to T' \quad \therefore \Delta_1' \vdash \rho\pi\; S' \ni s}{\Delta_0' + \Delta_1' \vdash \rho\; f'\; s' \in T'[s'\!:\!S'/x]}$

where Lemma 14 tells us that $T[s : S/x] \twoheadrightarrow T'[s' : S'/x]$. This leaves only the case where application performs β-reduction, the villain of the piece. We have

$$\frac{\begin{array}{l}\Delta_0 \vdash \rho\; (\lambda\, x.\, t : (\pi\, x\!:\!S_0) \to T_0) \in (\pi\, x\!:\!S_1) \to T_1 \\ \Delta_1 \vdash \rho\pi\; S_1 \ni s\end{array}}{\Delta_0 + \Delta_1 \vdash \rho\; (\lambda\, x.\, t : (\pi\, x\!:\!S_0) \to T_0)\; s \in T_1[s\!:\!S_1/x]} \qquad \begin{array}{cccccccccc} \Delta_0 & + & \Delta_1 & t & S_0 & T_0 & s & S_0 & T_0 \\ \downarrow & & \downarrow & \triangledown & \triangledown & \triangledown & \triangledown & \downarrow & \downarrow \\ \Delta_0' & + & \Delta_1' & t' & S_0' & T_0' & s' & S_1 & T_1 \end{array}$$

noting that POST might mean we apply at a function type computed from that given. Let us first interrogate the type checking of the function. There will have been some FUN, and after PRE computing $S_0 \twoheadrightarrow S_2$ and $T_0 \twoheadrightarrow T_2$, some LAM:

$$\frac{\lfloor\Delta_0\rfloor \vdash_0 *_i \ni S_0 \quad \lfloor\Delta_0\rfloor, x\!:\!S_0 \vdash_0 *_i \ni T_0}{\lfloor\Delta_0\rfloor \vdash_0\; *_i \ni (\pi\, x\!:\!S_0) \to T_0} \qquad \frac{\Delta_0, \rho\pi\, x\!:\!S_2 \vdash \rho\; T_2 \ni t}{\Delta_0 \vdash \rho\; (\pi\, x\!:\!S_2) \to T_2 \ni \lambda\, x.\, t}$$

We compute a common reduct $S_0 \twoheadrightarrow \{S'_0, S_1, S_2\} \twoheadrightarrow S'_1$, and deduce inductively

$$\begin{array}{ll} \therefore \lfloor \Delta'_0 \rfloor, x : S'_1 \vdash_0 *_i \ni T'_0 & \therefore \lfloor \Delta'_1 \rfloor \vdash_0 *_i \ni S'_0 \\ \therefore \Delta'_0, \rho\pi\, x : S'_1 \vdash \rho\, T'_0 \ni t' & \therefore \Delta'_1 \vdash \rho\pi\, S'_0 \ni s' \end{array}$$

so that CUT and POST give us $\Delta'_1 \vdash \rho\pi\ s' : S'_0 \in S'_1$. Now, Lemma 34 (stability under substitution) and CUT give us

$$\frac{\lfloor \Delta'_0 + \Delta'_1 \rfloor \vdash_0 *_i \ni T'_0[s' : S'_0/x] \quad \Delta'_0 + \Delta'_1 \vdash \rho\, T'_0[s' : S'_0/x] \ni t'[s' : S'_0/x]}{\Delta'_0 + \Delta'_1 \vdash \rho\ (t' : T'_0)[s' : S'_0/x] \in T'_0[s' : S'_0/x]}$$

so POST can compute our target type to a common reduct $T_0 \twoheadrightarrow \{T'_0, T_1, T_2\} \twoheadrightarrow T'_1$, and deliver $\Delta'_0 + \Delta'_1 \vdash \rho\ (t' : T'_0)[s' : S'_0/x] \in T'_1[s' : S'_1/x]$. □

Corollary 36 (Preservation).

$$\begin{array}{ccc} \Delta & T & t \\ \downarrow & \downarrow & \downarrow \\ \Delta' & T' & t' \end{array} \Rightarrow \frac{\Delta \vdash \rho\, T \ni t}{\Delta' \vdash \rho\, T' \ni t'} \qquad \begin{array}{cc} \Delta & e \\ \downarrow & \downarrow \\ \Delta' & e' \end{array} \Rightarrow \exists \begin{array}{c} S \\ \downarrow \\ S' \end{array} \wedge \frac{\Delta \vdash \rho\, e \in S}{\Delta' \vdash \rho\, e' \in S'}.$$

Proof. Iteration of Theorem 35. □

11 Erasure to an Implicit Calculus

Runtime programs live in good old lambda calculus and teletype font, to boot.

Definition 37 (Programs). `p` $::= x \mid$ `\`x `->` `p` $\mid$ `p p`.

I introduce two new judgment forms for programs, which arise as the erasure of our existing fully explicit terms.

Definition 38 (Erasure Judgments). *When $\rho \neq 0$, we may form judgments as follows:* $\Delta \vdash \rho\, T \ni t\ \blacktriangleright \texttt{p} \quad \Delta \vdash \rho\, e \in S\ \blacktriangleright \texttt{p}$.

That is, programs are nonzero-resourced. Such judgments are derived by an elaborated version of the existing rules which add programs as outputs. For checking, we must omit the type formation rules, but we obtain implicit and explicit forms of abstraction. For synthesis, we obtain implicit and explicit forms of application. In order to ensure that contemplation never involves consumption, we must impose a condition on the rig $\mathcal{R}$ that not only is the presence of negation unnecessary, but also its absence is vital: $\dfrac{\rho + \pi = 0}{\rho = \pi = 0}$.

Definition 39 (Checking and Synthesis with Erasure).

$$\textsc{pre+}\quad \frac{\Delta \vdash \rho\ R \ni t\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}}\ T \rightsquigarrow R$$

$$\textsc{lam0}\quad \frac{\Delta, 0\,x\!:\!S \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ (\phi\, x\!:\!S) \to T \ni \lambda\, x.\, t\ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

$$\textsc{lam+}\quad \frac{\Delta, \rho\pi\, x\!:\!S \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ (\pi\, x\!:\!S) \to T \ni \lambda\, x.\, t\ \blacktriangleright \mathtt{\backslash} x\ \mathtt{->}\ \mathtt{p}}\ \rho\pi \neq 0$$

$$\textsc{elim+}\quad \frac{\Delta \vdash \rho\ e \in S\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni \underline{e}\ \blacktriangleright \mathtt{p}}\ S \preceq T$$

$$\textsc{post+}\quad \frac{\Delta \vdash \rho\ e \in S\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ e \in R\ \blacktriangleright \mathtt{p}}\ S \rightsquigarrow R$$

$$\textsc{var+}\quad \frac{}{0\Gamma, \rho\, x\!:\!S, 0\Gamma' \vdash \rho\ x \in S\ \blacktriangleright x}$$

$$\textsc{app0}\quad \frac{\Delta \vdash \rho\ f \in (\phi\, x\!:\!S) \to T\ \blacktriangleright \mathtt{p} \quad \lfloor\Delta\rfloor \vdash_0 S \ni s}{\Delta \vdash \rho\ f\ s \in T[s\!:\!S/x]\ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

$$\textsc{app+}\quad \frac{\Delta \vdash \rho\ f \in (\pi\, x\!:\!S) \to T\ \blacktriangleright \mathtt{p} \quad \Delta' \vdash \rho\pi\ S \ni s\ \blacktriangleright \mathtt{p}'}{\Delta + \Delta' \vdash \rho\ f\ s \in T[s\!:\!S/x]\ \blacktriangleright \mathtt{p}\,\mathtt{p}'}\ \rho\pi \neq 0$$

$$\textsc{cut+}\quad \frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\ S \ni s\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ s\!:\!S \in S\ \blacktriangleright \mathtt{p}}$$

We can be sure that in the LAM0 rule, the variable x bound in t occurs nowhere in the corresponding $\mathtt{p}$, because it is bound with resource 0, and it will remain with resource 0 however the context splits, so the rule VAR+ cannot *consume* it, even though VAR can still contemplate it. Accordingly, no variable escapes its scope. We obtain without difficulty that erasure can be performed.

Lemma 40 (Erasures Exist Uniquely and Elaborate). *If* $\rho \neq 0$, *then*

$$\frac{\Delta \vdash \rho\ T \ni t}{\exists!\mathtt{p}.\ \Delta \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}} \qquad \frac{\Delta \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni t}$$

$$\frac{\Delta \vdash \rho\ e \in S}{\exists!\mathtt{p}.\ \Delta \vdash \rho\ e \in S\ \blacktriangleright \mathtt{p}} \qquad \frac{\Delta \vdash \rho\ e \in S\ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ e \in S}.$$

Proof. Induction on derivations. □

The unerased forms may thus be used to form types, as Theorem 28 then gives us that $\frac{\Delta \vdash \rho\ T \ni t\ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 T \ni t}$ $\frac{\Delta \vdash \rho\ e \in S\ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 e \in S}$.

How do programs behave? They may compute by β reduction, liberally.

Definition 41 (Program Computation).

$$\frac{}{(\mathtt{\backslash} x\ \mathtt{->}\ \mathtt{p})\ \mathtt{p}' \rightsquigarrow \mathtt{p}[\mathtt{p}'/x]} \qquad \frac{\mathtt{p} \rightsquigarrow \mathtt{p}'}{\mathtt{\backslash} x\ \mathtt{->}\ \mathtt{p} \rightsquigarrow \mathtt{\backslash} x\ \mathtt{->}\ \mathtt{p}'} \qquad \frac{\mathtt{p} \rightsquigarrow \mathtt{p}'}{\mathtt{p}\,\mathtt{p}_a \rightsquigarrow \mathtt{p}'\,\mathtt{p}_a} \qquad \frac{\mathtt{p} \rightsquigarrow \mathtt{p}'}{\mathtt{p}_f\,\mathtt{p} \rightsquigarrow \mathtt{p}_f\,\mathtt{p}'}$$

The key to understanding the good behaviour of computation is to observe that any term which erases to some `\`x `->` `p` must contain a subterm on its left spine typed with the LAM+ rule. On the way to that subterm, appeals to APP0 will be bracketed by appeals to LAM0, ensuring that we can dig out the non-zero λ by computation. Let us now show that we can find the LAM+.

Definition 42 (n-to-ρ-Function Type). *Inductively, $(\pi x\!:\!S) \to T$ is a 0-to-ρ-function type if $\rho\pi \neq 0$; $(\phi x\!:\!S) \to T$ is an $n+1$-to-ρ-function type if $\rho\phi = 0$ and T is an n-to-ρ-function type.*

Note that n-to-ρ-function types are stable under substitution and subtyping. Let λ^n denote λ-abstraction iterated n times.

Lemma 43 (Applicability). *If $\Delta \vdash \rho\ T \ni t\ \blacktriangleright$ `\`x `->` `p`, then $T \twoheadrightarrow$ some n-to-ρ-function type T' and $t \twoheadrightarrow$ some $\lambda^n\,\boldsymbol{y}.\ \lambda\,x.\,t'$ such that*

$$\Delta \vdash \rho\, T' \ni \overset{n}{\lambda}\,\boldsymbol{y}.\ \lambda\,x.\,t'\ \blacktriangleright \texttt{\textbackslash}x\ \texttt{->}\ \texttt{p}$$

If $\Delta \vdash \rho\ e \in S\ \blacktriangleright$ `\`x `->` `p`, then $S \twoheadrightarrow$ some n-to-ρ-function type S' and $e \twoheadrightarrow$ some $\lambda^n\,\boldsymbol{y}.\ \lambda\,x.\,t' : S'$ such that

$$\Delta \vdash \rho\ \overset{n}{\lambda}\,\boldsymbol{y}.\ \lambda\,x.\,t' : S' \in S'\ \blacktriangleright \texttt{\textbackslash}x\ \texttt{->}\ \texttt{p}.$$

Proof. Proceed by induction on derivations. Rules VAR+ and APP+ are excluded by the requirement to erase to `\`x`->p`. For PRE+, the inductive hypothesis applies and suffices. For LAM0, the inductive hypothesis tells us how to compute t and T to an abstraction in an n-to-ρ-function type, and we glue on one more $\lambda\,y.\,-$ and one more $(\phi\, y\!:\!S) \to -$, respectively. At LAM+, we can stop. For POST+, the inductive hypothesis gives us a cut at type S', where $S \twoheadrightarrow S'$, so we can take the common reduct $S \twoheadrightarrow \{S', R\} \twoheadrightarrow R'$ and deliver the same cut at type R'. For CUT+, we again proceed structurally. This leaves only the entertainment.

For ELIM+, we had $e \in S$ with $S \preceq T$. Inductively, $e \twoheadrightarrow \lambda\,\boldsymbol{y}.\ \lambda\,x.\,t' : S'$ with $S \twoheadrightarrow S'$, some n-to-ρ-function type. Lemma 20 gives us that $T \twoheadrightarrow T'$ with $S' \preceq T'$, also an n-to-ρ-function type. Hence $T' \ni \underline{\lambda\,\boldsymbol{y}.\ \lambda\,x.\,t' : S'}$ and then Theorem 35 (preservation) allows the υ-reduction to $T' \ni \lambda\,\boldsymbol{y}.\ \lambda\,x.\,t'$.

For APP0, the inductive hypothesis gives us $f \twoheadrightarrow \lambda\,y.\ \lambda^n\,\boldsymbol{y}.\ \lambda\,x.\,t' : (\phi x : S') \to T'$ with $S \twoheadrightarrow S'$ and $T \twoheadrightarrow T'$, and T' an n-to-ρ-function type. Hence $f\ s \twoheadrightarrow (\lambda^n\,\boldsymbol{y}.\ \lambda\,x.\,t' \!:\! T')[s\!:\!S'/x]$. Preservation tells us the reduct is well typed at some other reduct of $T[s\!:\!S/x]$, but the common reduct is the type we need. □

Theorem 44 (Step Simulation). *The following implications hold.*

$$\frac{\Delta \vdash \rho\, T \ni t\ \blacktriangleright \texttt{p}}{\exists t'.\ t \twoheadrightarrow t' \wedge\ \Delta \vdash \rho\, T \ni t'\ \blacktriangleright \texttt{p}'}\ \texttt{p} \rightsquigarrow \texttt{p}'$$

$$\frac{\Delta \vdash \rho\, e \in S\ \blacktriangleright \texttt{p}}{\exists e', S'.\ e \twoheadrightarrow e' \wedge S \twoheadrightarrow S' \wedge\ \Delta \vdash \rho\, e' \in S'\ \blacktriangleright \texttt{p}'}\ \texttt{p} \rightsquigarrow \texttt{p}'.$$

Proof. Induction on derivations and inversion of program computation. The only interesting case is the APP+ case when the computation takes a β-step.

$$\frac{\Delta \vdash \rho\ f \in (\pi\, x\!:\!S) \to T\ \blacktriangleright \texttt{\textbackslash}x\ \texttt{->}\ \mathtt{p} \quad \Delta' \vdash \rho\pi\ S \ni s\ \blacktriangleright \mathtt{p}'}{\Delta + \Delta' \vdash \rho\ f\ s \in T[s\!:\!S/x]\ \blacktriangleright (\texttt{\textbackslash}x\ \texttt{->}\ \mathtt{p})\ \mathtt{p}' \rightsquigarrow \mathtt{p}[\mathtt{p}'/x]}\ \rho\pi \neq 0$$

We have some f whose type is a 0-to-ρ function type and which erases to some $\texttt{\textbackslash}x$ `->` `p`, so we may invoke Lemma 43 to get that $f \twoheadrightarrow \lambda x.\, t$ at some reduced function type, $(\pi\, x\!:\!S') \to T'$, where S' still accepts the argument s, by preservation, erasing to $\mathtt{p}'$. Accordingly, $f\ s \twoheadrightarrow (t\!:\!T')[s\!:\!S'/x]$, where the latter is still typed at a reduct of $T[s\!:\!S/x]$ and erases to $\mathtt{p}[\mathtt{p}'/x]$. □

Accordingly, once we know terms are well typed, contemplation has done its job and we may erase to virtuous and thrifty 0-free programs.

12 Take It or Leave It

Let us consider liberalising our rig-based resource management. At present, the $\{0, 1, \omega\}$ rig imposes a kind of *relevance*, but not traditional relevance, in that it takes at least two uses at multiplicity 1 or one at ω to discharge our spending needs: what if we want traditional relevance, or even the traditional intuitionistic behaviour? Similarly, we might sometimes want to weaken the linear discipline to affine typing, where data can be dropped but not duplicated.

One option is to impose an 'order', $\leq$ on the rig. We can extend it pointwise to Γ-contexts, so $\Delta \leq \Delta'$ if Δ' has at least as many of each variable in Γ as Δ.

The $\leq$ relation should be reflexive and transitive, and at any rate we shall need at least that the order respects the rig operations

$$\frac{}{\rho \leq \rho} \qquad \frac{\rho \leq \pi \quad \pi \leq \phi}{\rho \leq \phi} \qquad \frac{\pi \leq \phi}{\rho + \pi \leq \rho + \phi} \qquad \frac{\pi \leq \phi}{\rho\pi \leq \rho\phi} \qquad \frac{\pi \leq \phi}{\pi\rho \leq \phi\rho}$$

to ensure that valid judgments remain an $\mathcal{R}$-module when we add weakening:

$$\text{WEAK}\quad \frac{\Delta \vdash \rho\ T \ni t}{\Delta' \vdash \rho\ T \ni t}\ \Delta \leq \Delta'$$

To retain Lemmas 29 and 31 (factorization and splitting), we must also be able to factorize and split the ordering, so two more conditions on $\leq$ emerge: factorization and additive splitting.

$$\frac{\rho\pi \leq \rho\phi}{\pi \leq \phi} \qquad \frac{\phi + \rho \leq \pi}{\exists \phi', \rho'.\ \phi \leq \phi' \wedge \rho \leq \rho' \wedge \pi = \phi' + \rho'}$$

Stability under substitution requires no more conditions but a little more work. The following lemma is required to deliver the new case for WEAK.

Lemma 45 (Weakening). *If $\rho \leq \rho'$ then*

$$\frac{\Delta' \vdash \rho'\pi\ T \ni t}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\ T \ni t} \qquad \frac{\Delta' \vdash \rho'\pi\ e \in S}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\ e \in S}.$$

Proof. Induction on derivations, with the interesting cases being VAR, WEAK and APP: the rest go through directly by inductive hypothesis and replay of the rule. For VAR, form Δ by replacing the $\rho'\pi$ in Δ' by $\rho\pi$, which is smaller.

For WEAK, we must have delivered $\Delta'' \vdash \rho'\pi T \ni t$ from some Δ' with $\Delta' \leq \Delta''$ and $\Delta' \vdash \rho'\pi\ T \ni t$. By the inductive hypothesis, there is some $\Delta \leq \Delta'$ with $\Delta \vdash \rho\pi\ T \ni t$ and transitivity gives us $\Delta \leq \Delta''$.

For APP, the fact that $\leq$ respects multiplication allows us to invoke the inductive hypothesis for the argument, and then we combine the smaller contexts delivered by the inductive hypotheses to get a smaller sum. □

As a consequence, we retain stability of typing under substitution and thence type preservation. To retain safe erasure, we must prevent WEAK from bringing a contemplated variable back into a usable position by demanding $\frac{\rho \leq 0}{\rho = 0}$.

If we keep $\leq$ discrete, nobody will notice the difference with the rigid system. However, we may now add, for example, $1 \leq \omega$ to make ω capture run-time relevance, or $0 \leq 1 \leq \omega$ for affine typing, or $0, 1 \leq \omega$ (but not $0 \leq 1$) to make $(\omega\ x : S) \to T$ behave like an ordinary intuitionistic function type whilst keeping $(1\, x : S) \to T$ linear: you can throw only plenty away.

13 Contemplation

Our colleagues who have insisted that dependent types should depend only on replicable things have been right all along. The $\mathcal{R}$-module structure of derivations ensures that every construction fit for consumption has an intuitionistic counterpart fit for contemplation, replicable because it is made from nothing.

In a dependent type theory, the variables in types stand for things: which things? It is not always obvious, because types may be used to classify more than one kind of thing. The things that variables in types stand for are special: when substituted for variables, they appear in types and are thus contemplated. A fully dependent type theory demands that everything classified by type has a contemplatable image, not that everything is contemplatable.

Here, we use the same types to classify terms and eliminations in whatever quantity, and also to classify untyped programs after erasure, but it is the *eliminations of quantity zero* which get contemplated when the application rule substitutes them for a variable, and everything classified by a type in one way or another corresponds to just such a thing.

Such considerations should warn us to be wary of jumping to conclusions, however enthusiastic we may feel. We learned from Kohei Honda that session types are linear types, in a way which has been made precise intuitionistically by Caires and Pfenning [7], and classically by Wadler [31], but we should not expect a linear dependent type theory, to be a theory of dependent session types *per se*. The linear dependent type theory in this paper is *not* a theory of session types because contemplated terms give too much information: they represent the *participants* which input and output values according to the linear types.

Dependency in protocol types should concern only the *signals* exchanged by the participants, not the participants' private strategies for generating those signals. In fact, the *signal traffic*, the *participants* and the *channels* are all sorts of things classified by session types, but it is only the signal traffic which must inform dependency. That is another story, and I will tell it another time, but it is based on the same analysis of how dependent type theories work. It seems realistic to pursue programming in the style of Gay and Vasconcelos [13] with dependent session types and linearly managed channels.

What we do have is a basis for a propositions-as-types account of certifying linearly typed programs, where the idealised behaviour of the program can be contemplated in propositions. When proving theorems about functions with unit-priced types, we know to expect uniformity properties when the price is zero: from parametricity, we obtain 'theorems for free' [23,29]. What might we learn when the price is small but not zero? Can we learn that a function from lists to lists which is parametric in the element type and linear in its input necessarily delivers a permutation? If so, the mission to internalise such results in type theory, championed by Bernardy, Jansson and Paterson [5] is still more crucial.

Looking nearer, I have mentioned the desirability of a normalization proof, and its orthogonality to resourcing. We must also look beyond $\multimap$ and give dependent accounts of other linear connectives: dependent $\otimes$ clearly makes sense with a pattern matching eliminator; dependent $(x\!:\!S)\,\&\,T[x]$ offers the intriguing choice to select an S or a $T[x]$ whose type mentions what the S used to be, like money or goods to the value of the money. But what are the duals?

It would be good to study datatypes supporting mutation. We have the intriguing prospect of linear induction principles like

$$\begin{array}{l}\forall X\!:\!*.\ \forall P\!:\!\mathsf{List}\ X \to *.\\ (n\!:\!P\ [])\multimap(c\!:\!(x\!:\!X)\multimap(xsp\!:\!(xs\!:\!\mathsf{List}\ X)\,\&\,P\ xs)\multimap P\ (x :: \mathsf{fst}\ xsp))\to\\ (xs\!:\!\mathsf{List}\ X)\multimap P\ xs\end{array}$$

which allow us at each step in the list either to retain the tail or to construct a P from it, but not both. Many common programs exhibit this behaviour (insertion sort springs to mind) and they seem to fit the heuristic identified by Domínguez and Pardo for when the fusion of paramorphisms an optimisation [11].

What we can now bring to all these possibilities is the separation of contemplation from consumption, ensuring that contemplation requires no resource and can correspondingly be erased. More valuable, perhaps, than this particular technical answer to the challenge of fully integrating linear and dependent types is the learning of the question 'What are the contemplated images?'.

Acknowledgements. Writing was a substitute for sleep in a busy summer of talks in the Netherlands, Germany and England. I thank everyone who interacted with me in the daytimes at least for their forbearance, and some for very useful conversations and feedback, notably James McKinna and Shayan Najd. The ideas were incubated during my visit to Microsoft Research Cambridge in 2014: I'm grateful to Nick Benton for feedback and encouragement. My talk on this topic at SREPLS in Cambridge provoked a very helpful interaction with Andy Pitts, Dominic Orchard, Tomas Petricek, and Stephen Dolan—his semiring-sensitivity provoked the generality of the resource treatment here [10]. I apologize to and thank the referees of all versions of this paper. Sam Lindley and Craig McLaughlin also deserve credit for helping me get my act together.

The inspiration, however, comes from Phil. The clarity of the connection he draws between classical linear logic and session types [31] is what attracted me to this problem. One day he, Simon Gay and I bashed rules at my whiteboard, trying to figure out a dependent version of that 'Propositions As Sessions' story. The need to distinguish 'contemplation' from 'consumption' emerged that afternoon: it has not yet delivered a full higher-order theory of 'dependent session types', but in my mind, it showed me the penny I had to make drop.

References

1. Abel, A.: Normalization by Evaluation: Dependent Types and Impredicativity, Habilitationsschrift (2013)
2. Abel, A., Coquand, T., Dybjer, P.: Normalization by evaluation for Martin-Löf Type theory with typed equality judgements. In: Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10–12 July 2007, Wroclaw, Poland, pp. 3–12. IEEE Computer Society (2007)
3. Adams, R.: Pure type systems with judgemental equality. J. Funct. Program. **16**(2), 219–246 (2006)
4. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: A bi-directional refinement algorithm for the calculus of (co)inductive constructions. Logical Meth. Comput. Sci. **8**(1), 1–49 (2012)
5. Bernardy, J.-P., Jansson, P., Paterson, R.: Proofs for free - parametricity for dependent types. J. Funct. Program. **22**(2), 107–152 (2012)
6. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(5), 552–593 (2013)
7. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
8. Cervesato, I., Pfenning, F.: A linear logical framework. Inf. Comput. **179**(1), 19–75 (2002)
9. de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 201–218. Springer, Heidelberg (2008)
10. Dolan, S.: Fun with semirings: a functional pearl on the abuse of linear algebra. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, 25–27 September, pp. 101–110. ACM, Boston (2013)
11. Domínguez, F., Pardo, A.: Program fusion with paramorphisms. In: Proceedings of the International Conference on Mathematically Structured Functional Programming, MSFP 2006, pp. 6–6. British Computer Society, Swinton (2006)

12. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, 23–25 January, pp. 357–370. ACM, Rome (2013)
13. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010)
14. Gentzen, G.: Untersuchungen über das logische schließen. Math. Z. **29**(2–3), 176–210 (1935). 405–431
15. Girard, J.-Y.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
16. Krishnaswami, N.R., Pradic, P., Benton, N.: Integrating linear and dependent types. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, 15-17 January 2015, pp. 17–30. ACM, Mumbai (2015)
17. Luo, Z.: ECC, an extended calculus of constructions. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), 5–8 June 1989, pp. 386–395. IEEE Computer Society, Pacific Grove (1989)
18. Martin-Löf, P.: An Intuitionistic theory of types: predicative part. In: Rose, H.E., Shepherdson, J.C. (eds.) Logic Colloquium 1973. North-Holland Publishing Company, Amsterdam (1975)
19. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. J. Autom. Reasoning **23**(3–4), 373–409 (1999)
20. Miquel, A.: The implicit calculus of constructions. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 344–359. Springer, Heidelberg (2001)
21. Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, 1–3 September 2014, pp. 123–135. ACM, Gothenburg (2014)
22. Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1), 1–44 (2000)
23. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
24. Shi, R., Xi, H.: A linear type system for multicore programming in ATS. Sci. Comput. Program. **78**(8), 1176–1192 (2013)
25. Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. J. Funct. Program. **23**(4), 402–451 (2013)
26. Takahashi, M.: Parallel reductions in λ-calculus (revised version). Inf. Comput. **118**(1), 120–127 (1995)
27. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 20–22 July 2011, pp. 161–172. ACM, Odense (2011)
28. Vákár, M.: Syntax and Semantics of Linear Dependent Types. CoRR, abs/1405.0033 (2014)
29. Wadler, P.: Theorems for free! In: FPCA, pp. 347–359 (1989)
30. Wadler, P.: Is there a use for linear logic? In: Consel, C., Danvy, O. (eds.) Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 1991, Yale University, New Haven, Connecticut, USA, 17-19 June 1991, pp. 255–273. ACM (1991)

31. Wadler, P.: Propositions as sessions. In: Thiemann, P., Findler, R.B. (eds) ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, 9–15 September 2012, pp. 273–286. ACM, Copenhagen (2012)
32. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.W.: A concurrent logical framework: the propositional fragment. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 355–377. Springer, Heidelberg (2004)

Pointlessness is Better than Listlessness

John T. O'Donnell(✉) and Cordelia V. Hall

School of Computing Science, University of Glasgow, Glasgow, UK
john.odonnell@glasgow.ac.uk

Abstract. Extensible sparse functional arrays (ESFA) is a persistent data structure with an implementation that performs every operation in $O(1)$ time and space. There is no requirement for single threading of arrays, and no performance penalty for sharing. The implementation is an example of hardware/software co-design and also of active data structures. This makes the work interdisciplinary, and it builds on many ideas from functional programming. These include the definition of new array operations, observations about mutability and purity, monads, program transformations to remove unnecessary data structures, equational reasoning, and fixing space leaks in the garbage collector. This paper summarises a recently published exposition of the system, focusing on drawing connections with some of these foundational ideas. It also presents some new results on a simulation of ESFA on a GPU. A surprising result is that although the simulation has high overheads and limited parallelism, it can outperform a state of the art red black tree implementation when there is a large amount of sharing between arrays.

1 Introduction

Sometimes a program can be transformed to run faster by eliminating unnecessary work. An example is Wadler's "listless" transformation (Wadler 1984), which avoids the creation of unnecessary intermediate lists. This improves efficiency, but the remaining lists still limit performance because it takes time to follow chains of pointers in order to find a particular item.

By combining digital circuit design with algorithms — transforming the underlying machine as well as the software — we can get rid of some of the pointers too. The idea is to design a "smart memory" with a small amount of processing capability in each location. The benefit comes from the massive parallelism inherent in digital circuits: every logic gate is operating in parallel, all the time. Many of them can be made to do useful work much of the time, although conventional architectures waste nearly all of this processing power. Such machines are called associative memories or content addressable parallel processors, and the combination of parallel circuits and software is an example of hardware/software co-design. The general approach is sometimes called *active data structures* because a memory operation initiated by the CPU causes the memory to perform many small computations internally (O'Donnell, Hall, and Monro 2013).

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 234–248, 2016.
DOI: 10.1007/978-3-319-30936-1_13

A particularly challenging data structure is the pure functional array, which might better be called a persistent array. When an array a is updated with a mapping $i \mapsto v$ from an index i to a value v, the result is a new array a' which is identical to a except that it contains the new mapping. The original array a is unchanged — it is persistent. Functional arrays can be implemented using data structures related to binary search trees (Graefe 2010), giving logarithmic time access provided the trees are balanced. To be efficient, the algorithms need both to rebalance trees and maintain sharing between arrays (O'Neill 1994) and (O'Neill 2000).

This paper discusses the application of circuit parallelism to extensible sparse functional arrays (ESFA). The system consists of a machine (a digital circuit) called ESFM, along with algorithms running on ESFM that implement the operations for ESFA. Circuit parallelism enables the machine to perform a variety of data structure operations in unit time, while chains of pointers need to be followed on a conventional architecture. Each lookup and each update takes a small constant number of memory accesses, regardless of the past history of updates. There is no restriction on which arrays may be updated, and there is no loss of sharing.

The system has been described in a previous paper (O'Donnell 2015). In this paper we give a high level overview of the main ideas and provide a broader context. Then we present some new results about an implementation of ESFM using general purpose graphical processing units (GPUs) and compare the performance with that of a C library. Finally, we discuss work in progress on memory management.

There are several other approaches to functional arrays, and the choice depends on the nature of the application. We argue that functional arrays are simply a different data structure than imperative arrays. Both kinds of array can be used in an imperative language, and also in a functional language. The distinction between imperative and functional arrays lies in the API, not in the programming language.

Section 2 presents two APIs for functional arrays: a pure one and a stateful one using a monad. Section 3 explains the main idea: using a novel circuit to eliminate explicit tree structures and avoid following pointers. Section 4 shows two applications of equational reasoning in deriving or proving the correctness of the implementation: one application is in the hardware design, and the other is in the algorithm running on that hardware. In Sect. 5 we discuss using a multicore (a GPGPU) to simulate the ESFM circuit, and show how the GPU architecture suggests a new family of bulk array operations. Future directions are suggested in Sect. 6, where a space leak is identified along with a possible solution using a hardware garbage collector. Section 7 concludes with a discussion of the links between this work and some of the classic early research in functional programming.

2 Shall I Be Pure?

Imperative languages are based on the assignment statement, which calculates a new value, destroys the information stored in a variable, and replaces it with the new value. The new value of a variable is stored in the same place as the original one. This has several advantages: it enables circular structures to be traversed by marking nodes as they are encountered, and a small part of a large structure can be changed allowing efficient incremental updates to an array. However, variables vary over time, so equational reasoning is not valid.

Pure functional languages are based on the equation, which also calculates a new value and saves it but does not insist on destroying any existing information. The new value may be stored in a newly allocated memory location. Values are persistent and immutable, while destruction is left to the garbage collector. This approach is pure and retains referential transparency, but is incompatible with imperative arrays and makes some algorithms harder to implement efficiently (e.g. traversing circular data structures).

Effects introduce impurity, while purity leads to inefficiency. Wadler asked rhetorically "is there a way to combine the indulgences of impurity with the blessings of purity?" (Wadler 1992). Answering in the affirmative, he showed how to apply monads to allow computational effects in a pure functional language. This makes it possible to use imperative arrays in a functional language, retaining the benefits of both effects and purity.

But that is not the end of the story for functional arrays. When you update an imperative array, the old array contents are gone. If an algorithm needs access to the old array value as well as the new one, then monads don't help: the sharing of pure functional arrays is needed.

Imperative arrays and functional arrays are distinct data structures, with different APIs. They are not tied to particular programming paradigms. You can use imperative arrays in a functional language (with monads, or unique types), and you can also use functional arrays in an imperative language. The historical terminology can be misleading; the essential distinction is whether the arrays are persistent, not whether the host programming language is functional or imperative.

It has turned out to be useful to give two APIs for ESF arrays: a high level one where each array operation is a pure function (Table 1), and a low level one that makes the machine state explicit (Table 2). The high level interface allows the programmer to manipulate arrays with pure update and lookup functions, and requires storage management to be handled automatically. The low level interface gives access to operations needed by the storage manager, and requires performing all the array operations in a monad.

Table 1 defines the high level API, which consists of ordinary expressions and pure function applications. This API places no constraints on the structure of the program; there is no need to perform array operations in a monad. There is a constant *empty* which is an array containing no elements. Every array other than *empty* must be constructed using *update*, and arrays are accessed using *lookup*. The remaining operations support sparse access by traversing the defined elements of an array. Two laws state the relationship between *lookup* and *update*.

Table 1. Array operations: pure functional API

$empty$	$:: Array$
$update$	$:: Array \rightarrow (Idx \mapsto Val) \rightarrow Array$
$lookup$	$:: Array \rightarrow Idx \rightarrow Maybe\ Val$
$minDef, maxDef$	$:: Array \rightarrow Maybe\ (Idx \mapsto Val)$
$nextDef, prevDef$	$:: Array \rightarrow Idx \rightarrow Maybe\ (Idx \mapsto Val)$

Law 1 (Empty Array). *For all* $i :: Idx$*,*

$$lookup\ empty\ i = Nothing$$

Law 2 (Nonempty Array). *For all* $a :: Array$*, element* $(j \mapsto v) :: (Idx \mapsto Val)$ *and index* $i :: Idx$*,*

$$\begin{aligned}&lookup\ (update\ a\ (j \mapsto v))\ i\\ &\quad \mid i \equiv j = Just\ v\\ &\quad \mid i \not\equiv j = lookup\ a\ i\end{aligned}$$

Memory management is essential and is not expressible in the pure interface. It could be performed either by the user program or by an automatic garbage collector. In either case, it is better to write the memory management as an imperative functional program (Peyton Jones and Wadler 1993), rather than doing it all at the lowest level of programming. Therefore a lower level interface is needed to make memory management explicit. The crucial low level operation is to delete an array. This operation changes the state, so it needs to be provided in a monad.

$$\begin{aligned}&\textbf{type}\ SystemState\ s\ e\ a = StateT\ (CircuitState\ s, e)\ IO\ a\\ &\textbf{type}\ EsfaState\ a = SystemState\ Cell\ AuxState\ a\\ &\textbf{type}\ Result\ a = Either\ ErrMsg\ a\end{aligned}$$

Table 2 shows the lower level API. The array accesses (*updateS* etc.) are similar to the pure counterparts. The last two operations change the machine state. When an array is deleted, the array handle is reclaimed and all elements that have become inaccessible are identified (but see Sect. 6).

There is a special facility to enable a finaliser to be run when an element is reclaimed. This is necessary, for instance, if the element contains a pointer back into the heap in the CPU's memory. When an *updateS* creates a new array with a new element, a Boolean parameter *notify* indicates whether the main program wants to be notified when the element is reclaimed. Later, when the element is identified as garbage, it is reclaimed immediately if the Boolean is False. If *notify* is true, then the element is left in the ESFM memory but marked

Table 2. Array operations in the stateful API. The *deleteS* and *killZombieS* operations support memory management, and the others correspond to functions in the pure API in Table 1.

updateS	$:: Array \rightarrow (Idx \mapsto Val) \rightarrow Bool \rightarrow EsfaState\ (Result\ Array)$
lookupS	$:: Array \rightarrow Idx \rightarrow EsfaState\ (Result\ Val)$
minDefS,*maxDefS*	$:: Array \rightarrow EsfaState\ (Result\ (Idx \mapsto Val))$
nextDefS,*prevDefS*	$:: Array \rightarrow Idx \rightarrow EsfaState\ (Result\ (Idx \mapsto Val))$
deleteS	$:: Array \rightarrow EsfaState\ ()$
killZombieS	$:: EsfaState\ (Maybe\ Val)$

as a zombie. There is an operation *killZombie* that searches associatively for a zombie, reclaims it, and returns the value of the element for finalisation.

Both the *deleteS* and *killZombie* operations are single machine operations; they take $O(1)$ time.

Monads are not needed for pure functional arrays, but they do allow for explicit implementation of memory management. The programmer can choose which approach is suitable for an application.

3 Transforming Programs to Eliminate Trees

Wadler showed how to transform programs to eliminate trees (Wadler 1990). The reason functional arrays are tricky to implement on a conventional machine is that the resulting trees require time to traverse and to rebalance.

The essential idea behind ESFA is to eliminate *all* the trees used to represent arrays. To do this we need to do some circuit design, because the new algorithms do not run on conventional hardware. For the full details see (O'Donnell 2015); this section just introduces some of the ideas.

Ordinary memory hardware ("random access memory") accesses data by its address. Associative memory hardware accesses data by its contents. For example, suppose each memory word contains two fields: a key and a value. Instead of an instruction like `load Reg,address` — which requires finding the address of the data — the hardware provides an instruction like `loadval Reg,key`. This instruction broadcasts the key to each memory cell, which compares it with the local key value. If a cell finds a match, it returns the corresponding value.

The ESF Machine (ESFM) is related to associative memory but provides more general computational capabilities. Instead of memory words, it holds a set of *cells*. Each cell contains several fields, including the index and value of an array element, and some control bits indicating whether the cell is full or empty.

We can think of an imperative array as "knowing" where its elements are. That is, given the address of an array a, the element of $a[i]$ is just $a+k\times i$ where k is the size of an element. That approach is inconsistent with the massive sharing that ESF arrays may have.

The key idea is to reverse the perspective completely. We will represent an array by a *handle*, which is just a unique number that is unrelated to the locations of any of the array's elements. Instead, we will just place the elements (the mappings $i \mapsto v$ from index i to value v) in random locations anywhere in the machine. Associative searching rather than address is used to locate an element.

In this reverse perspective, an array doesn't know its elements, but instead *each element knows the set of arrays that contain the element.* This is called the element's inclusion set.

To perform all the operations in the API within $O(1)$ time and space — with no restrictions whatsoever on sharing — we need to be able to represent every inclusion set in constant space, and test for set membership in constant time. For arbitrary sets this would be impossible, but inclusion sets are not arbitrary: they satisfy a set of invariants that are maintained by all the ESFA operations.

The invariants ensure that (1) we can identify each array a by a code c and (2) associate a pair of natural numbers *low* and *high* with each cell x containing an element, such that a is in the inclusion set of the element in x if and only if $low\ x \leqslant c \leqslant high\ x$. For an explanation of why this is so, see (O'Donnell 2015). For now, the point is that every inclusion set can be represented by just two numbers, and set membership can be determined by two comparisons.

When a lookup takes place, the elements are found by parallel comparisons in each cell. When an update takes place, many (typically half) of the array codes and low/high values in the entire machine need to be incremented. When a delete takes place, many of these are decremented. This looks like a lot of overhead, but it is exactly the kind of computation that circuit parallelism is good at: each cell has enough logic gates to perform a comparison and an increment, and all the cells can do this in parallel.

4 Calculating is Better than Scheming

The implementation of ESF arrays contains two levels. The lower level is a digital circuit design that provides combinators for parallel maps, folds, and scans. The upper level implements each operation in the API using the parallel combinators. Both levels contain many details, are rather complex, and tricky to get right. For both, it has proved more successful to calculate the algorithm rather than to do ad hoc coding (Wadler 1987a).

4.1 Deriving Circuit Combinators

The lower level is a digital circuit specified functionally with the Hydra hardware description language, a domain specific language embedded in Haskell. Rather than describing the circuit directly, a *circuit generator* is used. This is a higher order function that constructs a large circuit from building blocks. The overall structure is a tree circuit, with building block circuits at the leaves (called cells) and nodes. A clock cycle in the circuit performs an upsweep and downsweep vertically through the tree.

```
sweep
  :: (s → d → (s, u))            -- cf = cell function
  → (d → u → u → (u, d, d))      -- nf = node function
  → d                            -- a = root input
  → CircuitState s               -- state of tree circuit
  → (CircuitState s, u)          -- (new state, root output)
```

A sweep causes each cell to apply a logic function to its state to produce an output to send up; later this function also calculates the new state using the current state and the incoming down message. These two logic functions are combined in the single function *cf*.

```
sweep cf nf a (Leaf c) = (Leaf c', y)
  where (c', y) = cf c a
```

Each node uses its *nf* function to calculate the value a' to send up, and the values p' and q' to send down to the left and right subtrees. Again, all these calculations are combined in one function *nf*. An alternative way to define a general tree circuit is to separate the cell and node functions into separate up and down functions, and then to define separate *upsweep* and *dnsweep* functions.

```
sweep cf nf a (Node x y) = (Node x' y', a')
  where (a', p', q') = nf a p q
        (x', p) = sweep cf nf p' x
        (y', q) = sweep cf nf q' y
```

The operations in Table 2 are implemented using a family of combinators that define global parallel computations. These include map and several varieties of fold and scan. The combinators are implemented by instantiating *sweep* with cell and node circuits, in order to generate the full machine. The *tscanl* combinator is similar to the *scanl* function defined in the Haskell prelude.

```
tscanl
  :: (s → a)          -- get singleton from cell
  → (s → a → s)       -- update cell using singleton
  → (a → a → a)       -- h: function to be folded
  → a                 -- initial accumulator
  → SystemState s e a
```

This function defines a communication pattern that begins with the cells, transmits information up the tree, and then transmits further information back down the tree. If h is associative, then *tscanl* performs a scan-from-left over the cells of the tree.

The implementation of *tscanl* is short but subtle, and some incorrect versions have been published. It can be derived using equational reasoning; this gives a clear insight as well as a correctness proof (O'Donnell 1994). The other combinators can be derived as well.

$$\begin{array}{l} . \, .tscanl \; f \; g \; h \; a = sweepm \; cf \; nf \; a \\ \quad \textbf{where} \; cf \; s \; a = (g \; s \; a, f \; s) \\ \qquad\qquad nf \; a \; p \; q = (h \; p \; q, a, h \; a \; p). \end{array}$$

4.2 Deriving Effect on State of Array Operations

The implementation of the array operations using the parallel combinators is even more subtle than the combinators themselves. Each operation must perform a combination of associative searches and conditional changes to fields within each cell. For most operations there are a number of different cases for each cell, depending on the relationship between that cell's fields and the arguments to the operation.

Again, equational reasoning is the key to calculating what each cell needs to do. The calculation is carried out at the level of sets rather than the machine representation; this abstracts away from some low level details and also allows for choices about the lowest level representation to be deferred.

We focus on the inclusion sets of the cells, not directly on the arrays. The aim is to show that after an operation, every element in the machine has the right inclusion set. An interesting case is $a' \leftarrow updateS \; a \; elt \; nfy$ where the new array (with code c) must be added to the inclusion set of a cell x. In this case, $low \; x \leqslant c \leqslant high \; x$. This case is called *expand* because the inclusion set of the cell x' after the operation has been expanded to include the new array b.

The full proof is given in (O'Donnell 2015), but the following calculation gives the flavour of the proofs underlying the system.

Lemma 1 (Case Expand). *If* $low \; x \leqslant c \leqslant high \; x$ *then* $iset \; x' \; st' = iset \; x \; st \cup \{b\}$ *where* b *is the result of the update.*

Proof.

$$\begin{array}{ll}
\equiv & \langle \; \textit{definition of iset} \; \rangle \\
& \{p \mid p \in Nat \wedge q = encode \; p \; st' \wedge low \; x' \leqslant q \leqslant high \; x'\} \\
\equiv & \langle \; \textit{substitute } x \textit{ for } x' \; \rangle \\
& \langle \; low \; x' = low \; x \; \textit{and} \; high \; x' = high \; x + 1 \; \rangle \\
& \{p \mid p \in Nat \wedge q = encode \; p \; st' \wedge low \; x \leqslant q \leqslant high \; x + 1\} \\
\equiv & \langle \; \textit{split into three sets based on location of } c \; \rangle \\
& \{p \mid p \in Nat \wedge q = encode \; p \; st' \wedge low \; x \leqslant q \leqslant c\} \\
& \cup \{p \mid p \in Nat \wedge q = encode \; p \; st' \wedge q \equiv c + 1\} \\
& \cup \{p \mid p \in Nat \wedge q = encode \; p \; st' \wedge c + 1 < q \leqslant high \; x + 1\} \\
\equiv & \langle \; \textit{(1) Where } q \leqslant c, \; q = encode \; p \; st' \equiv encode \; p \; st. \; \rangle \\
& \langle \; \textit{(2) Since } encode \; b \; st' = c + 1 \; \rangle \\
& \langle \; \textit{(3) Where } q > c, \; q = encode \; p \; st' \equiv encode \; p \; st + 1. \; \rangle \\
& \{p \mid p \in Nat \wedge q = encode \; p \; st \wedge low \; x \leqslant q \leqslant c\} \\
& \cup \{b\} \\
& \cup \{p \mid p \in Nat \wedge q = encode \; p \; st \wedge c + 1 < q + 1 \leqslant high \; x + 1\}
\end{array}$$

$\equiv$ $\langle$ *Subtract 1 from both sides of inequalities in (3)* $\rangle$
$\langle$ *Reorder the sets as* $\cup$ *is commutative* $\rangle$

$$\{p \mid p \in Nat \wedge q = encode\ p\ st \wedge low\ x \leqslant q \leqslant c\}$$
$$\cup \{p \mid p \in Nat \wedge q = encode\ p\ st \wedge c < q \leqslant high\ x\}$$
$$\cup \{b\}$$

$\equiv$ $\langle$ *Combine the set comprehensions* $\rangle$

$$\{p \mid p \in Nat \wedge q = encode\ p\ st \wedge low\ x \leqslant q \leqslant high\ x\}$$
$$\cup \{b\}$$

$\equiv$ $\langle$ *Definition of iset* $\rangle$

$$iset\ x\ st \cup \{b\}.$$

5 A New Array Operation: Performance on a GPU

Thinking about both the implementation and the usage of a data structure can suggest ways to enhance the API. An early example is Wadler's new array operation, which performs monolithic update with folds used to combine values at the same index (Wadler 1986). This section describes yet another new array operation(s) which appeared as the result of developing a parallel simulator for ESFA. The results in this section are preliminary, and have not been published.

The ESFM is a digital circuit that implements a smart memory. In some situations a *simulation* of ESFM can actually be faster than a direct implementation using state of the art tree algorithms. Simulation is a useful technique to study the behaviour of a circuit, but circuit simulators are slow because they make a calculation for each component on each clock cycle. Therefore we developed a parallel simulator for ESFA using a multicore GPGPU (a graphics processing unit enhanced for general purpose computing) (Owens 2007). The simulator was written in C+CUDA, and experiments were performed using an NVidia GeForce GTX 590 GPGPU with CUDA capability 2.0, 512 CUDA cores, and a clock speed of 1.22 GHz.

We would not expect the multicore simulator to be fast enough for real usage of ESFA, but it could be a helpful research tool. Instead of simulating the circuit at a very low level (registers and logic functions), the simulator implements the ESFM at the level of the parallel combinators. It was written to take advantage of the capabilities of a GPU, as well as overcoming the considerable difficulties with this architecture.

One property of GPGPUs is the ability for each thread to copy a value from shared memory to global memory, or vice versa, more or less in parallel. This suggests that we can introduce two new operations for the ESFA API: bulk update and bulk lookup. For a bulk lookup, we first identify the cells that belong to an array, and then copy the corresponding elements in parallel to the global memory. For a bulk update, we first allocate the required number of new cells in parallel (if the machine contains enough free cells), set the values of the *low* and *high* fields in parallel, and then copy the elements in parallel from global memory into these cells. This is decidedly *not* the same as copying a slice from an array stored in contiguous memory locations.

A bulk update takes an array a and a vector of k values $x_0, x_1, \ldots, x_{k-1}$. It creates a new array a' which is the same as a but with the mappings $0 \mapsto x_0, \ldots, k-1 \mapsto x_{k-1}$. The indices must be consecutive because they are generated by a parallel scan; it is also possible to define a more general bulk update where the indices are provided explicitly, but the implementation is slower.

In addition to the bulk operations we also introduced associative searching, which is a natural application of circuit parallelism. Indeed, the ESFM is a distant descendant of associative memories from nearly 50 years ago.

The entire API for ESFA has been implemented on the GPU, along with the bulk operations. For comparison, we have also implemented the ESFA API using the Map library in C++, a state of the art implementation of red black trees.

The C++ Map library implements essentially the same API as ESFA. It supports sparse array operations that are part of the ESFM API, and so makes an excellent basis for functional arrays. The main difference between C++ Map and ESFM is that Map performs side effects on the trees, so copying is sometimes necessary. If most updates are single threaded, then Map performs very well, but if there is a large amount of sharing then ESFM is faster and uses less space. The Map copy constructor can form the basis of an implementation of functional arrays, with a space penalty when arrays are shared.

We developed an experimental testbed for measuring performance. This uses two separate implementations of the API: one using the ESFM implemented in C+CUDA running on the GPU, and one using C++ Map running on a single CPU. We used a variety of test data: some small cases worked out by hand, and some large test data that was generated by machine.

Several points should be kept in mind while considering the performance measurements. The ESFM tests ran on a GPU with 512 relatively small processor cores, while the Map tests ran on a sequential CPU. However, the nominal performance of the GPU is not 512 times that of the CPU, due to a variety of overheads. (It is not really meaningful to say that the GPU is x times more powerful than the CPU, but a factor of 20 to 50 might be a reasonable guess.)

We found that initial tests of functional arrays in C++ were very fast, much faster than ESFM. In particular, one test run containing 50,000 (relatively) random operations created by Hydra, which we used to test our GPU implementation of ESFM, ran with 2 µs per operation under C++, as opposed to 18 µs for ESFM.

This appears to show that the copying required to implement functional arrays in C++ is not an undue burden when there is not too much sharing. However, the story becomes more interesting when we consider what happens when long arrays are created without deletes to control the use of space, associative search and bulk updates, and the use of no-copying semantics to make the C++ implementation as efficient as possible.

5.1 Bulk Updates

A bulk update is potentially more efficient than an equivalent sequence of incremental updates, even on a sequential machine, because the implementation of *update* knows something about the shape of the resulting data structure. On the GPU, bulk update is even better because many of the memory transfers can be done in parallel. We experimented with bulk updates of various "chunk sizes".

Ideally, we would expect that the C++ implementation would take time proportional to the size of the update chunk, while the GPU would be not too much worse than constant time, until the chunks become large enough to overwhelm the processor cores. Table 3 shows that this is largely true. However, because the C++ implementation was not doing copying, but was instead building the arrays from an empty array, this test was optimal for that implementation.

Table 3. Bulk update and search

μs per bulk update (rounded)		
Chunk Size	C++ Map	ESFM
1	1	58
2	2	58
4	3	58
8	6	58
16	12	59
32	23	59
64	46	59
128	93	61
256	190	65
512	387	72
1024	786	87
2048	1594	117
4096	3255	177
8192	6775	300
16384	13924	544

μs per search (rounded)		
Array Size	C++ Map	ESFM
1	0	66
2	0	66
4	0	66
8	0	66
16	1	66
32	1	66
64	2	66
128	4	66
256	9	66
512	17	66
1024	36	66
2048	74	66
4096	150	67
8192	276	66
16384	538	67

5.2 Searches

A search operation specifies an array and a value, and it produces a list of all indices in the array that are mapped to the value. If several indices map to the value there is a "collision", and resolving collisions is the hardest aspect of associative search. In general, searches are very fast on the ESFM. To investigate the worst case, we experimented with searches where all the elements of a chunk have the same value. The arrays used in these experiments were generated by bulk updates.

Our hypothesis was that the time for an associative search over an array where all indices map to the same value would be proportional to the array size on the C++ implementation, while it would remain constant on the GPU. This proved to be the case, as can be seen in Table 3.

Both the bulk update and search operations on the GPU require two parallel scans, one of which operates over 1024 cells per block. Several techniques were used to obtain a reasonably efficient implementation; these include loop unrolling, avoidance of bank conflicts, and the Xiao and Feng inter-block synchronisation algorithm (Xiao and Feng 2010).

5.3 Long Arrays Created Using Updates

The C++ implementation is fast when there is relatively limited sharing and plenty of bulk updates; this allows the C++ program to do little copying.

The situation is quite different when there is a lot of sharing, which requires expensive copying for C++ but which is easy for ESFM. Table 4 shows the average time per update for the C++ Map implementation and the ESFM running on the GPU. Each line shows the array size n and the time per update (averaged over the n updates to create the array) for each implementation. The arrays contain extensive sharing, and as a result the C++ implementation slows down in line with the array size; it fails to terminate with 16K elements. In contrast, the ESFM implementation takes constant time per update.

It is worth reiterating that the GPU program is simulating a circuit-parallel system. The GPU has limited parallelism (512 cores, which is a small number for circuit simulation) and the program has considerable overhead, especially in synchronisation. Despite this, the GPU version of ESFA can outperform a state of the art red black tree implementation when there is a large amount of sharing between arrays.

5.4 Long Arrays with Deletes for Intermediate Arrays

Since keeping intermediate arrays was eventually a disaster for the C++ implementation, we experimented with what happened when deletes were inserted at critical moments, ensuring that the most recent arrays still remained. This corresponds to a situation where a program builds arrays with a lot of sharing, but does not actually need the sharing and periodically cleans up by deleting the arrays that will not be needed again.

Manual cleanup using delete operations sped up the C++ version considerably, suggesting that explicit space management by the user program is really quite important for the C++ version. Of course, this technique cannot be used if the program needs access to the old arrays.

For the ESFM, each delete operation takes a constant time (13 µs on the machine we used). The time for a deletion does not depend on the shapes or sizes of the arrays, or the past history of operations. Furthermore, the usage of delete has no effect on the speed of the other operations; the only purpose of delete is to reclaim memory so that a future update will succeed.

Table 4. Update with sharing (μs per update)

Array Size	C++ Map	ESFM
1	1	19
2	1	20
4	2	20
8	3	20
16	5	20
32	10	20
64	18	20
128	34	20
256	63	20
512	114	20
1024	218	20
2048	428	20
4096	853	20
8192	1711	20
16384	Hangup	20

6 Fixing Some Space Leaks with a Hardware Garbage Collector

One of the greatest practical advances in functional programming has proven to be automatic memory management, which has been adopted widely. When the programmer must explicitly allocate and free memory, serious errors are hard to avoid. If memory is released that shouldn't be, the result is a dangling reference; conversely, if memory isn't released which is actually inaccessible, the result is a space leak.

Garbage collectors greatly reduce such errors, but they are not always perfect. Wadler identified a situation where a space leak results from a perfectly reasonable programming style, and showed how to fix the space leak by modifying the garbage collector (Wadler 1987b).

An analogous situation arises with ESF arrays, although the details are different. The low level API for ESFA provides an operation to delete an array a. The array handle is reclaimed, and all array elements that are inaccessible are reclaimed *in one instruction.* Any array elements that can be accessed via a different handle (some array other than a) are of course retained. The *delete* operation seems almost like a garbage collector that takes $O(1)$ time — apart from one problem.

The problem stems from shadowing: if an index is bound to several different values in an array, only the most recent binding is visible and the older ones are shadowed (inaccessible). More precisely, suppose array a_0 is *empty* and a_{i+1}

is defined by updating a_i, for $0 \leqslant i < n$. Consider an arbitrary update in this sequence: say a_j was created with the new element $i \mapsto x$. Now if another array a_k in the sequence is created later (so $k > j$) with the same index (e.g. with a new element $i \mapsto y$) we say that the older element $i \mapsto x$ is *shadowed* in a_n. That is, *lookup a i* should return y, not x.

Shadowing is relative to a specific array. Thus the element $i \mapsto x$ is shadowed in a, but it is visible in a_j and possibly many other arrays. The associative techniques, based on comparing an array code with the *low* and *high* fields of a cell, determine the set of *candidates* for the elements of an array. If this set contains two elements $i \mapsto x$ and $i \mapsto y$, then the *rank* is used to choose the right one for a *lookup*.

Now suppose a_j is deleted, while a_n still exists. Should the element associated with a_j be reclaimed? If is inaccessible via a_j but it might be accessible via some other array that still exists. But what if the element happens to be shadowed in *all* such arrays? In that case, the element is inaccessible in the sense that there does not exist a *lookup* that would return it. But *delete* cannot reclaim the element without knowing its shadowing status.

Work on this problem is currently in progress, with the ultimate goal of stating precisely the correctness conditions for a storage manager and proving that the system satisfies them.

7 Conclusion

ESF arrays are based on implementing an API using novel hardware as well as algorithms. Thus it brings together several diverse topics, including algorithms, data structures, parallelism, and digital circuits.

This work builds on a number of foundational ideas in computer science, and a remarkable number of them were introduced or advanced by Philip Wadler. These include new functional array operations, giving low level support for data structures in the garbage collector, using monads to allow effects to combine with purity, and arguing eloquently for the use of equational reasoning and program transformation.

Research is often insular and focused on narrow topics. A broader perspective can lead to solutions that would be unachievable within a specialised area.

References

Graefe, G.: Modern B-tree techniques. Found. Trends Databases **3**(4), 203–402 (2010)

O'Donnell, J.T., Hall, C., Monro, S.: Active data structures on GPGPUs. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 875–884. Springer, Heidelberg (2014)

O'Donnell, J.: A correctness proof of parallel scan. Parallel Process. Lett. **4**(3), 329–338 (1994)

O'Donnell, J.T.: Extensible sparse functional arrays with circuit parallelism. Sci. Comput. Prog. **111**(P1), 23–50 (2015)

O'Neill, M.E.: A data structure for more efficient runtime support of truly functional arrays. Master's thesis, Simon Fraser University (1994)

O'Neill, M.E.: Version stamps for functional arrays and determinacy checking: two applications of ordered lists for advanced programming languages. Ph.D. thesis, Simon Fraser University (2000)

Owens, J.D.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)

Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: ACM Symposium on Principles of Programming Languages. ACM, January 1993

Wadler, P.: A critique of Abelson and Sussman or why calculating is better than scheming. SIGPLAN Not. **22**(3), 83–94 (1987a)

Wadler, P.: Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP 1984, pp. 45–52. ACM, New York (1984)

Wadler, P.: A new array operation. In: Fasel, J.H., Keller, R.M. (eds.) Graph Reduction 1986. LNCS, vol. 279, pp. 328–335. Springer, Heidelberg (1987)

Wadler, P.: Fixing some space leaks with a garbage collector. Softw. Pract. Experience **17**(9), 595–608 (1987b)

Wadler, P.: Transforming programs to eliminate trees. Theoret. Comput. Sci. **73**, 231–248 (1990)

Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**, 461–493 (1992). Cambridge University Press

Xiao, S., Feng, W.: Inter-block GPU communication via fast barrier synchronization. In: IPDPS, pp. 1–12 (2010)

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1(✉)], Tiark Rompf[2], and Sandro Stucki[1]

[1] EPFL, Lausanne, Switzerland
{nada.amin,samuel.grutter,martin.odersky,sandro.stucki}@epfl.ch
[2] Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_{<:}$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_{<:}$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Keywords: Calculus · Dependent types · Scala

1 Introduction

While hiking together in the French alps in 2013, Martin Odersky tried to explain to Phil Wadler why languages like Scala had foundations that were not directly related via the Curry-Howard isomorphism to logic. This did not go over well. As you would expect, Phil strongly disapproved. He argued that anything that was useful should have a grounding in logic. In this paper, we try to approach this goal.

We will develop a foundation for Scala from first principles. Scala is a functional language that expresses central aspects of modules as first-class terms and types. It identifies modules with *objects* and signatures with *traits*. For instance, here is a trait Keys that defines an abstract type Key and a way to retrieve a key from a string.

```
trait Keys {
  type Key
  def key(data: String): Key
}
```

A concrete implementation of Keys could be

```
object HashKeys extends Keys {
  type Key = Int
  def key(s: String) = s.hashCode
}
```

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 249–272, 2016.
DOI: 10.1007/978-3-319-30936-1_14

Here is a function which applies a given key generator to every element of a list of strings.

```
def mapKeys(k: Keys, ss: List[String]): List[k.Key] =
    ss.map(k.key)
```

The function returns a list of elements of type k.Key. This is a *path-dependent* type, which depends on the variable name k. In general a path in Scala consists of a variable name potentially followed by field selections, but in this paper we consider only simple paths consisting of a single variable reference. Path-dependent types give a limited form of type/term dependency, where types can depend on variables, but not on general terms. This form of dependency already gives considerable power, but at the same time is easier to handle than full term dependency because the equality relation on variables is just syntactic equality.

There are three essential elements of a system for path-dependent types. First, there needs to be a way to define a type as an element of a term, as in the definitions type Key and type Key = Int. We will consider initially only *type tags*, values that carry just one type and nothing else. Second, there needs to be a way to recover the tagged type from such a term, as in p.Key. Third, there needs to be a way to define and apply functions whose types depend on their parameters.

These three elements are formalized in System $D_{<:}$, a simple calculus for path-dependent types that has been discovered recently in an effort to reconstruct and mechanize foundational type system proposals for Scala bottom-up from simpler systems (Rompf and Amin 2015). One core aspect of our approach to modeling path-dependent types is that instead of general substitutions we only have variable/variable renamings. This ensures that paths always map to paths and keeps the treatment simple. On the other hand, with substitutions not available, we need another way to go from a type designator like k.Key to the underlying type it represents. The mechanism used here is subtyping. Types of type tags define lower and upper bound types. An abstract type such as type Key has the minimal type $\bot$ and the maximal type $\top$ as bounds. A type alias such as type Key = T has T as both its lower and upper bound. A type designator is then a subtype of its upper bound and a supertype of its lower bound.

The resulting calculus is already quite expressive. It emerges as a generalization of System $F_{<:}$ (Cardelli et al. 1994), mapping type lambdas to term lambdas and type parameters to type tags. We proceed to develop System $D_{<:}$ into a richer calculus supporting objects as first class modules. This is done in three incremental steps. The first step adds records and intersections, the second adds type labels, and the third adds recursion. The final calculus, *DOT*, is a foundation for path-dependent object types. Many programming language concepts, including parameterized types and polymorphic functions, modules and functors, classes and algebraic data types can be modeled on this foundation via simple encodings.

A word on etymology: The term "System D" is associated in English and French with "thinking on your feet" or a "quick hack". In the French origin of the word, the letter 'D' stands for "se débrouiller", which means "to manage" or "to get things done". The meaning of the verb "débrouiller" alone is much nicer. It means: "Create order for things that are in confusion". What better motto for the foundations of a programming language?

Even if the name "System D" suggests quick thinking, in reality the development was anything but quick. Work on DOT started in 2007, following earlier work on higher-level formalizations of Scala features (Odersky et al. 2003; Cremet et al. 2006). Preliminary versions of a calculus with path-dependent types were published in FOOL 2012 (Amin et al. 2012) and OOPSLA 2014 (Amin et al. 2014). Soundness results for versions of System $D_{<:}$ and DOT similar to the ones presented here, but based on big-step operational semantics, were recently established with mechanized proofs (Rompf and Amin 2015).

The rest of this paper is structured as follows. Section 2 describes System $D_{<:}$. Section 3 shows how it can encode $F_{<:}$. Section 4 extends $D_{<:}$ to DOT. Sections 5 and 6 study the expressiveness of DOT and show how it relates to Scala. Section 7 outlines the implementation of the DOT constructs in a full Scala compiler. Section 8 discusses related work and Sect. 9 concludes.

2 System $D_{<:}$

Figure 1 summarizes our formulation of System $D_{<:}$. Its term language is essentially the lambda calculus, with one additional form of value: A type tag $\{A = T\}$ is a value that associates a label A with a type T. For the moment we need only a single type label, so A can be regarded as ranging over an alphabet with just one name. This will be generalized later.

Our description differs from Rompf and Amin (2015) in two aspects. First, terms are restricted to ANF form. That is, every intermediate value is abstracted out in a let binding. Second, evaluation is expressed by a small step reduction relation, as opposed to a big-step evaluator. Reduction uses only variable/variable renamings instead of full substitution. Instead of being copied by a substitution step, values stay in their let bindings. This is similar to the techniques used in the call-by-need lambda calculus (Ariola et al. 1995).

We use Barendregt's Variable Convention throughout. For example, in the third evaluation rule, which un-nests let-bindings, we assume that we can appropriately α-rename the variable y which changes scope so that it is not captured in the final term u.

The type assignment rules in Fig. 1 define a straightforward dependent typing discipline. A lambda abstraction has a dependent function type $\forall(x:S)T$. This is like a dependent product $\Pi(x:S)T$ in LF (Harper et al. 1993), but with the restriction that the variable x can be instantiated only with other variables, not general terms. Type tags have types of the form $\{A : S..U\}$, they represent types labeled A which are lower-bounded by S and upper-bounded by U. A type tag referring to one specific type is expressed by having the lower and upper bound coincide, as in $\{A : T..T\}$. The type of a variable x referring to a type tag can be recovered with a type projection $x.A$.

Syntax

x, y, z	**Variable**
$v ::=$	**Value**
$\{A = T\}$	type tag
$\lambda(x\!:\!T)t$	lambda
$s, t, u ::=$	**Term**
x	variable
v	value
$x\ y$	application
$\textbf{let}\ x = t\ \textbf{in}\ u$	let

$S, T, U ::=$	**Type**
$\top$	top type
$\bot$	bottom type
$\{A : S..T\}$	type declaration
$x.A$	type projection
$\forall(x\!:\!S)T$	dependent function

Evaluation $\boxed{t \longrightarrow t}$

$$\begin{array}{rcll}
\textbf{let}\ x = v\ \textbf{in}\ e[x\ y] & \longrightarrow & \textbf{let}\ x = v\ \textbf{in}\ e[[z := y]t] & \textbf{if}\ v = \lambda(z\!:\!T)t \\
\textbf{let}\ x = y\ \textbf{in}\ t & \longrightarrow & [x := y]t & \\
\textbf{let}\ x = \textbf{let}\ y = s\ \textbf{in}\ t\ \textbf{in}\ u & \longrightarrow & \textbf{let}\ y = s\ \textbf{in}\ \textbf{let}\ x = t\ \textbf{in}\ u & \\
e[t] & \longrightarrow & e[u] & \textbf{if}\ t \longrightarrow u \\
 & & \multicolumn{2}{l}{\textbf{where}\ e ::= [\,] \mid \textbf{let}\ x = [\,]\ \textbf{in}\ t \mid \textbf{let}\ x = v\ \textbf{in}\ e}
\end{array}$$

Type Assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma,\ x : T,\ \Gamma' \vdash x : T \quad (\textsc{Var}) \qquad\qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad (\textsc{Sub})$$

$$\frac{\Gamma,\ x : T \vdash t : U \quad x \notin fv(T)}{\Gamma \vdash \lambda(x\!:\!T)t : \forall(x\!:\!T)U} \quad (\textsc{All-I}) \qquad\qquad \frac{\Gamma \vdash x : \forall(z\!:\!S)T \quad \Gamma \vdash y : S}{\Gamma \vdash x\ y : [z := y]T} \quad (\textsc{All-E})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma,\ x : T \vdash u : U \quad x \notin fv(U)}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : U} \quad (\textsc{Let}) \qquad\qquad \Gamma \vdash \{A = T\} : \{A : T..T\} \quad (\textsc{Typ-I})$$

Subtyping $\boxed{\Gamma \vdash T <: T}$

$$\Gamma \vdash T <: \top \quad (\textsc{Top}) \qquad\qquad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (\textsc{Trans})$$

$$\Gamma \vdash T <: T \quad (\textsc{Refl})$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (\textsc{<:-Sel}) \qquad\qquad \frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\textsc{Sel-<:})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma,\ x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x\!:\!S_1)T_1 <: \forall(x\!:\!S_2)T_2} \quad (\textsc{All-<:-All}) \qquad\qquad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \quad (\textsc{Typ-<:-Typ})$$

$$\Gamma \vdash \bot <: T \quad (\textsc{Bot})$$

Fig. 1. System $D_{<:}$

The subtyping rules in Fig. 1 define a preorder $S <: T$ between types with rules (Refl) and (Trans). They specify $\top$ and $\bot$ as greatest and least types (Top), (Bot), and make a type projection $x.A$ a supertype of its lower bound (<:-Sel) and a subtype of its upper bound (Sel-<:). Furthermore, the standard co/contravariant subtyping relationships are introduced between pairs of function types (All-<:-All) and tagged types (Typ-<:-Typ).

System $\mathrm{D}_{<:}$ can encode System $\mathrm{F}_{<:}$ as we will see in Sect. 3. However, unlike System $\mathrm{F}_{<:}$, System $\mathrm{D}_{<:}$ does not have type variables. Instead, type definitions, such as $\{A = T\}$, are first-class values of type $\{A : T..T\}$. Combined with dependent functions, these path-dependent types can express the idioms of type variables, such as polymorphism.

For example, take the polymorphic identity function in System $F_{<:}$:

$$\vdash \Lambda(\alpha <: \top).\lambda(x : \alpha).x \quad : \quad \forall(\alpha <: \top).\alpha \rightarrow \alpha$$

and in System $D_{<:}$:

$$\vdash \lambda(a : \{A : \bot..\top\}).\lambda(x : a.A).x \quad : \quad \forall(a : \{A : \bot..\top\})\forall(x : a.A)a.A$$

Like in System $F_{<:}$, we can apply the polymorphic identity function to some type, say T, to get the identity function on T:

$$\vdash \textbf{let}\, f = \ldots \,\textbf{in let}\, a = \{A = T\} \,\textbf{in}\, f\ a \quad : \quad \forall(x : T)T$$

The role of subtyping is essential: (1) the argument a of type $\{A : T..T\}$ can be used for the parameter a of type $\{A : \bot..\top\}$, (2) the dependent result type $\forall(x : a.A)a.A$ can be converted to $\forall(x : T)T$ because $T <: a.A <: T$.

2.1 Example: Dependent Sums

Dependent sums can be encoded using dependent functions, through an encoding similar to that of existential types in System F.

$$\begin{aligned}
\Sigma(x : S)T &\equiv \forall(z : \{A : \bot..\top\})\forall(f : \forall(x : S)\forall(y : T)z.A)z.A \\
\textbf{pack}\,[x, y]\,\textbf{as}\,\Sigma(x : S)T &\equiv \lambda(z : \{A : \bot..\top\})\lambda(f : \forall(x : S)\forall(y : T)z.A)f\ x\ y \\
\textbf{unpack}\ \ x : S, y : T = t \,\textbf{in}\, u &\equiv \textbf{let}\, z_1 = t \,\textbf{in let}\, z_2 = \{A = U\} \,\textbf{in} \\
&\quad\ \textbf{let}\, z_3 = (\lambda(x : S)\lambda(y : T)u) \,\textbf{in} \\
&\quad\ \textbf{let}\, z_4 = z_1\ z_2 \,\textbf{in}\, z_4\ z_3 \\
z.1 &\equiv \textbf{unpack}\ \ x : S, y : T = z \,\textbf{in}\, x \\
z.2 &\equiv \textbf{unpack}\ \ x : S, y : T = z \,\textbf{in}\, y
\end{aligned}$$

where U is the type of u. The associated, admissible subtyping and typing rules are easy to derive and can be found in Appendix A.1. Note that

1. unpacking via **unpack** $x, y = t \,\textbf{in}\, u$ is only allowed if x and y do not appear free in the type U of u,

2. similarly, the second projection operator $-.2$ may only be used if x does not appear free in T. In such cases, we have $\Sigma(x : S)T = S \times T$, i.e. z is in fact an ordinary pair.

These restrictions may come as a surprise: while they are similar to the hygiene conditions imposed on existential types in System F/F$_{<:}$, they do not apply to dependent sums in (fully) dependently typed languages. In such languages, the bound names x, y can be prevented from leaking into the overall type of an **unpack** statement by substituting the projections $t.1$ and $t.2$ for occurrences of x and y in U. The same is true for the return type of the second projection $z.2$, which would be $[x := z.1]T$. Unfortunately, such substitutions are forbidden in D$_{<:}$ because types may only depend on variables, as opposed to arbitrary terms (like $z.1$ or $t.2$). For the same reason, the typing rule (Let) for **let** expressions features a similar hygiene condition. Finally, note that the above encoding allows for the unrestricted projection of "existential witnesses" via $-.1$, whereas no such operation exists on existential types in System F/F$_{<:}$.

3 Embedding F$_{<:}$ in D$_{<:}$

System D$_{<:}$ initially emerged as a generalization of F$_{<:}$, mapping type lambdas to term lambdas and type parameters to type tags, and removing certain restrictions in a big-step evaluator for F$_{<:}$ (Rompf and Amin 2015). We make this correspondence explicit below.

Pick an injective mapping from type variables X to term variables x_X. In the following, any variable names not written with an X subscript are assumed to be outside the range of that mapping. Let the translation * from F$_{<:}$ types and terms to D$_{<:}$ types and terms be defined as follows.[1]

$$\begin{aligned}
X^* &= x_X.A \\
\top^* &= \top \\
(T \to U)^* &= \forall(x{:}T^*)U^* \\
(\forall(X <: S)T)^* &= \forall(x_X{:}\{A : \bot..S^*\})T^*
\end{aligned}$$

$$\begin{aligned}
x^* &= x \\
(\lambda(x : T)t)^* &= \lambda(x : T^*)t^* \\
(\Lambda(X <: S)t)^* &= \lambda(x_X : \{A : \bot..S^*\})t^* \\
(t\ u)^* &= \ \mathbf{let}\, x = t^* \,\mathbf{in}\, \mathbf{let}\, y = u^* \,\mathbf{in}\, x\ y && x, y \text{ fresh} \\
(t[U])^* &= \ \mathbf{let}\, x = t^* \,\mathbf{in}\, \mathbf{let}\, y_Y = \{A = U^*\} \,\mathbf{in}\, x\ y_Y && x, Y \text{ fresh}
\end{aligned}$$

[1] The definition of t^* assumes a countable supply of fresh names x, $X \notin fv(t)$.

Note that there are $D_{<:}$ terms that are not in the image of *. An example is $\lambda(x : \{A{:}\top..\top\})x$. Typing contexts are translated point-wise as follows:

$$
\begin{aligned}
(X <: T)^* &= x_X : \{A : \bot..T^*\}\\
(x : T)^* &= x : T^*
\end{aligned}
$$

Theorem 1. *If $\Gamma \vdash_F S <: T$ then $\Gamma^* \vdash_D S^* <: T^*$.*

Proof. The proof is by straight-forward induction on $F_{<:}$ subtyping derivations. The only non-trivial case is subtyping of type variables, which follows from (Var) and (Sel-<:).

$$
\frac{\Gamma^*,\ x_X : \{A : \bot..T^*\},\ \Gamma'^* \vdash x_X : \{A : \bot..T^*\}}{\Gamma^*,\ x_X : \{A : \bot..T^*\},\ \Gamma'^* \vdash x_X.A <: T^*}\ (\textsc{Sel}- <:)
$$

Theorem 2. *If $\Gamma \vdash_F t : T$ then $\Gamma^* \vdash_D t^* : T^*$.*

Proof (Sketch). The proof is by induction on (System F) typing derivations. The case for subsumption follows immediately from preservation of subtyping. The only remaining interesting cases are type and term application, which are given in detail in Appendix A.2.

4 DOT

Figure 2 presents three extensions needed to turn $D_{<:}$ into a basis for a full programming language. The first extension adds records, the second adds type labels, and the third adds recursion. For reasons of space, all three extensions are combined in one figure.

4.1 Records

We model records by single field values that can be combined through intersections. A single-field record is of the form $\{a = t\}$ where a is a term label and t is a term. Records d_1, d_2 can be combined using the intersection $d_1 \wedge d_2$. The selection $x.a$ returns the term associated with label a in the record referred to by x. The evaluation rule for field selection is:

$$
\mathbf{let}\, x = v\, \mathbf{in}\, e[x.a] \;\longrightarrow\; \mathbf{let}\, x = v\, \mathbf{in}\, e[t] \quad \mathbf{if}\ v = \ldots\{a = t\}\ldots
$$

It's worth noting that records are "call-by-name", that is, they associate labels with terms, not values. This choice was made because it sidesteps the issue how record fields should be initialized. The choice does not limit expressiveness, as a fully evaluated record can always be obtained by using let bindings to pre-evaluate field values before they are combined in a record.

New forms of types are single field types $\{a{:}T\}$ and intersection types $T \wedge U$. Subtyping rules for fields and intersection types are as expected. The typing

Syntax ...

a, b, c	**Term member**	$d ::=$	**Definition**
A, B, C	**Type member**	$\quad \{A = T\}$	type definition
$v ::=$	**Value**	$\quad \{a = t\}$	field definition
$\quad \nu(x{:}T)d$	object	$\quad d \wedge d'$	aggregate definition
$\quad \lambda(x{:}T)t$	lambda	$S, T, U ::= \ldots$	**Type**
$s, t, u ::= \ldots$	**Term**	$\quad \{a : T\}$	field declaration
$\quad x.a$	selection	$\quad S \wedge T$	intersection
		$\quad \mu(x{:}T)$	recursive type

Evaluation ... $\boxed{t \longrightarrow t'}$

$$\textbf{let } x = v \textbf{ in } e[x.a] \;\longrightarrow\; \textbf{let } x = v \textbf{ in } e[t] \quad \textbf{if } v = \nu(x{:}T) \ldots \{a = t\} \ldots$$

Type Assignment (terms) ... $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma,\ x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)} \ (\{\}\text{-I}) \qquad \frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \ (\text{Fld-E})$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)} \ (\text{Rec-I}) \qquad \frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T} \ (\text{Rec-E})$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \ (\text{And-I})$$

Type Assignment (definitions) $\boxed{\Gamma \vdash d : T}$

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \ (\text{Typ-I}) \qquad \frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \quad dom(d_1) \cap dom(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \ (\text{AndDef-I})$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}} \ (\text{Fld-I})$$

Subtyping ... $\boxed{\Gamma \vdash T <: T}$

$$\Gamma \vdash T \wedge U <: T \ (\text{And}_1\text{-}<:) \qquad \Gamma \vdash T \wedge U <: U \ (\text{And}_2\text{-}<:)$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \ (<:\text{-And})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \ (\text{Fld-}<:\text{-Fld})$$

Fig. 2. DOT extensions to $D_{<:}$

rules (Fld-I) and (Fld-E) introduce and eliminate field types in the standard way. The typing rule (AndDef-I) types record combinations with intersection types. To ensure that combinations are consistent, it requires that the labels of the combined records are disjoint.

There is also the general introduction rule (And-I) well known from lambda calculus with intersections (Coppo et al. 1979). In the system with non-recursive records, that rule is redundant, because it can be obtained from subsumption and (<:-And). As demonstrated in Sect. 5, the rule does add expressiveness once recursion is added.

4.2 Type Labels

Records with multiple type fields are supported simply by extending the alphabet of type labels from a single value to arbitrary names. In the following we let letters A, B, C range over type labels.

4.3 Recursion

The final extension introduces objects and recursive types. An object $\nu(x:T)d$ represents a record with definitions d that can refer to each other via the variable x. Its type is the recursive type $\mu(x:T)$. Following the path-dependent approach, a recursive type recurses over a term variable x, since type variables are absent from DOT. Note that type tags are no longer terms in their own right: they must now be wrapped in a ν binder like any other record definition.

The evaluation rule for records is generalized to the one in Fig. 2. There, when selecting $x.a$, the selected reference x and the self-reference in the object $\nu(x:T)d$ are required to coincide (this can always be achieved by α-renaming).

The rule ({}-I) assigns a recursive type to an object by typing its definition. Note that the definitions are typed without subsumption (which is only defined on terms, and in this final extension, definitions d are syntactically not terms t).

The introduction and elimination rules (Rec-I) and (Rec-E) for recursive types are as simple as they can possibly be. There is no subtyping rule between recursive types (Amadio and Cardelli 1993). Instead, recursive types of variables are unwrapped with (Rec-E) and re-introduced with (Rec-I).

The choice of leaving out subtyping between recursive types represents a small loss in expressiveness but a significant gain in simplifying the meta-theory.

4.4 Meta-Theory

This section presents the main type soundness theorems of DOT and outlines their proofs, which have been machine-checked.[2] The central results are the usual small-step preservation and progress theorems.

Soundness results for slightly different versions of $D_{<:}$, DOT, and a number of variations were established previously with mechanized proofs by Rompf and Amin. These soundness results are phrased with respect to big-step evaluators. The type systems are slightly more expressive than the ones presented here in that they are not restricted to terms in ANF, and in including a subtyping rule on recursive types. This particular rule poses lots of challenges, which are described in detail in the technical report (Rompf and Amin 2015).

[2] The full development on paper and in Coq is at http://wadlerfest.namin.net.

Definition 1. *An* answer *n is defined by the production*

$$n ::= x \mid v \mid \textbf{\textit{let}}\, x = v\, \textbf{\textit{in}}\, n$$

Theorem 3 (Progress). *If $\vdash t : T$ then t is an answer or there is a term u such that $t \longrightarrow u$.*

Theorem 4 (Preservation). *If $\vdash t : T$ and $t \longrightarrow u$ then $\vdash u : T$.*

Note that the preservation property is weaker than normally stated in that it demands an empty context. We will strengthen the theorem in the proofs to get a useful induction property.

The central difficulty for coming up with proofs of these theorems was outlined in a previous FOOL workshop paper (Amin et al. 2012): it is possible to arrive at type definitions with unsatisfiable bounds. For instance a type label A might have lower bound $\top$ and upper bound $\bot$. By subtyping rules (Sel-<:), (<:-Sel) and transitivity of subtyping one obtains $\top <: \bot$, which makes every type a subtype of every other type. So a single "bad" definition causes the whole subtyping relation to collapse to one point, just as an inconsistent theory can prove every proposition. With full type intersection and full recursion, it is impractical to rule out such bad bounds *a priori*. Instead, a soundness proof has to make use of the fact that environments corresponding to an actual execution trace correspond to types of concrete values. In a concrete object value any type definition is of the form $A = T$, so lower and upper bounds are the same, and bad bounds are excluded. Similar reasoning applied in the big-step proofs (Rompf and Amin 2015), where the semantics has a natural distinction between static terms and run-time values. Here, we need to establish this distinction first.

We first define a *precise typing relation $\Gamma \vdash_! t : T$* as follows:

Definition 2. *$\Gamma \vdash_! t$: T if $\Gamma \vdash t$: T and the following two conditions hold.*

1. *If t is a value, the typing derivation of t ends in* (All-I) *or* ({}-I)*.*
2. *If t is a variable, the typing derivation of t consists only of* (Var), (Rec-E) *and* (Sub) *steps and every* (Sub) *step makes use of only the subtyping rules (*And$_1$-<:*), (*And$_2$-<:*) and* (Trans)*.*

Definition 3. *A* store *s is a sequence of bindings $x = v$, with ϵ representing the empty store.*

Definition 4. *The combination $s \mid t$ combines a store s and a term t. It is defined as follows:*

$$\begin{aligned} x = v, s \mid t &\quad\equiv\quad \textbf{\textit{let}}\, x = v\, \textbf{\textit{in}}\, (s \mid t) \\ \epsilon \mid t &\quad\equiv\quad t \end{aligned}$$

Definition 5. *An environment $\Gamma = \overline{x_i : T_i}$ corresponds to a store $s = \overline{x_i = v_i}$, written $\Gamma \sim s$, if $\Gamma \vdash_! v_i : T_i$.*

A precise typing relation over an environment that corresponds to a store can only derive type declarations where lower and upper bounds coincide. We make use of this in the following definition.

Definition 6. *A typing or subtyping derivation is* tight *in environment Γ if it only contains the following tight variants of (*SEL-<:*), (*<:-SEL*) when $\Gamma' = \Gamma$:*

$$\frac{\Gamma' \vdash_! x : \{A : T..T\}}{\Gamma' \, T <: x.A} \ (\text{<:-SEL-TIGHT}) \qquad \frac{\Gamma' \vdash_! x : \{A : T..T\}}{\Gamma' \, x.A <: T} \ (\text{SEL-<:-TIGHT})$$

*For environments that extend Γ, full (*SEL-<:*) and (*<:-SEL*) are permitted. We write $\Gamma \vdash_{\#} t : T$ or $\Gamma \vdash_{\#} S <: U$ if $\Gamma \vdash t : T$ or $\Gamma \vdash S <: U$ with a derivation that is tight in Γ.*

A core part of the proof shows that the full (SEL-<:) and (<:-SEL) rules are admissible wrt $\vdash_{\#}$ if the underlying environment corresponds to a store.

Lemma 1. *If $\Gamma \sim s$ and $s(x) = \nu(x : T)d$ and $\Gamma \vdash_{\#} x : \{A : S..U\}$ then $\Gamma \vdash_{\#} x.A <: U$ and $\Gamma \vdash_{\#} S <: x.A$.*

The next step of the proof is to characterize the possible types of a variable bound in a store s in an environment that corresponds to s.

Definition 7. *The* possible types $\mathbf{Ts}(\Gamma, x, v)$ *of a variable x bound in an environment Γ and corresponding to a value v is the smallest set $\mathcal{S}$ such that:*

1. *If $v = \nu(x : T)d$ then $T \in \mathcal{S}$.*
2. *If $v = \nu(x : T)d$ and $\{a = t\} \in d$ and $\Gamma \vdash t : T'$ then $\{a : T'\} \in \mathcal{S}$.*
3. *If $v = \nu(x : T)d$ and $\{A = T'\} \in d$ and $\Gamma \vdash S <: T'$, $\Gamma \vdash T' <: U$ then $\{A : S..U\} \in \mathcal{S}$.*
4. *If $v = \lambda(x : S)t$ and $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash S' <: S$ and $\Gamma, x : S' \vdash T <: T'$ then $\forall(x : S')T' \in \mathcal{S}$.*
5. *If $S_1 \in \mathcal{S}$ and $S_2 \in \mathcal{S}$ then $S_1 \wedge S_2 \in \mathcal{S}$.*
6. *If $S \in \mathcal{S}$ and $\Gamma \vdash_! y : \{A : S..S\}$ then $y.A \in \mathcal{S}$.*
7. *If $T \in \mathcal{S}$ then $\mu(x : T) \in \mathcal{S}$.*
8. *$\top \in \mathcal{S}$.*

The possible types of a variable are closed under subtyping:

Lemma 2. *If $\Gamma \sim s$, $s(x) = v$, $T \in \mathbf{Ts}(\Gamma, x, v)$ and $\Gamma \vdash T <: U$, then $U \in \mathbf{Ts}(\Gamma, x, v)$.*

The possible types of a variable include each of its typings:

Lemma 3. *If $\Gamma \sim s$ and $\Gamma \vdash x : T$ then $T \in \mathbf{Ts}(\Gamma, x, s(x))$.*

These possible types lemmas are proved first for tight subtyping and typing, by induction on the subtyping or typing derivation, and extended to standard subtyping and typing with Lemma 1, since (SEL-<:) and (<:-SEL) are admissible for $\vdash_{\#}$ over Γ.

With the above lemmas it is easy to establish a standard canonical forms lemma.

Lemma 4 (Canonical Forms). If $\Gamma \sim s$, then

1. If $\Gamma \vdash x : \forall(x : T)U$ then $s(x) = \lambda(x : T')t$ for some T' and t such that $\Gamma \vdash T <: T'$ and $\Gamma, x : T \vdash t : U$.
2. If $\Gamma \vdash x : \{a{:}T\}$ then $s(x) = \nu(x : S)d$ for some S, d, t such that $\Gamma \vdash d : S$, $\{a = t\} \in d$, $\Gamma \vdash t : T$.

As usual, we also need a substitution lemma, the proof of which is by a standard mutual induction on typing and subtyping derivations.

Lemma 5 (Substitution). *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash y : [x := y]S$ then $\Gamma \vdash [x := y]t : [x := y]T$.*

With (Canonical Forms) and (Substitution) one can then establish the following combination of progress and preservation theorems by an induction on typing derivations.

Proposition 1. *Assume $\Gamma \vdash t : T$ and $\Gamma \sim s$. Then either*

- *t is an answer, or*
- *There exist a store s' and a term t' such that $s \mid t \longrightarrow s' \mid t'$ and for any such s', t' there exists an environment Γ' such that $\Gamma, \Gamma' \vdash t' : T$ and $\Gamma, \Gamma' \sim s'$.*

Theorems 3 and 4 follow from Proposition 1, since the empty environment corresponds to the empty store.

5 Encodings

In this section, we explore the expressiveness of DOT by studying program fragments that are typable in it. We start with an encoding of booleans, which is in spirit similar to the standard Church encoding, except that it also creates Boolean as a new nominal type. We use the abbreviation

```
IFT ≡ { if: ∀(x: {A: ⊥..⊤})∀(t: x.A)∀(f: x.A): x.A }
```

A first step defines the type boolImpl.Boolean as an alias of its implementation, a record with a single member if.

```
let boolImpl =
  ν (b: { Boolean: IFT..IFT } ∧ { true: IFT } ∧ { false: IFT
      })
    { Boolean = IFT } ∧
    { true = λ(x: {A: ⊥..⊤})λ(t: x.A)λ(f: x.A)t } ∧
    { false = λ(x: {A: ⊥..⊤})λ(t: x.A)λ(f: x.A)f }
in ...
```

In a second step, the implementation of Boolean gets wrapped in a function that produces an abstract type. This establishes bool.Boolean as a nominal type, which is different from its implementation. The type is upper bounded by *IFT*, which means that the conditional if is still accessible as a member.

```
let bool =
  let boolWrapper =
    λ(x: μ(b: {Boolean: ⊥..IFT} ∧ {true: b.Boolean} ∧ { false:
        b.Boolean })) x
  in boolWrapper boolImpl
in ...
```

This example shows that that direct mappings into DOT can be tedious at times. The following shorthands help.

5.1 Abbreviations

– We group multiple intersected definitions or declarations in one pair of braces, replacing $\wedge$ with ; or a newline. E.g.

```
{ A = T; a = t } ≡ { A = T } ∧ { a = t }
{ A: S..T; a: T } ≡ { A: S..T } ∧ { a: T }
```

– We allow terms in applications and selections, using the expansions

```
t u ≡ let x = t in x u
x u ≡ let y = u in x y
t.a ≡ let x = t in x.a
```

– We expand type ascriptions to applications:

```
t: T ≡ (λ(x: T)x)t
```

– We abbreviate ν(x: T)d to ν(x)d if the type of definitions d is given explicitly.
– We abbreviate type bounds by expanding A $<:$ T to A: $\bot$..T, A $>:$ S to A: S..$\top$, A $=$ T to A: T..T, and A to A: $\bot$..$\top$.

5.2 Example: A Covariant Lists Package

As a more involved example we show how can define a parameterized abstract data type as a class hierarchy. A simplified version of the standard covariant List type can be defined in Scala as follows:

```
package scala.collection.immutable
trait List[+A] {
  def isEmpty: Boolean; def head: A; def tail: List[A]
}
object List {
  def nil: List[Nothing] = new List[Nothing] {
    def isEmpty = true; def head = head; def tail = tail /*
        infinite loops */
  }
  def cons[A](hd: A, tl: List[A]) = new List[A] {
    def isEmpty = false; def head = hd; def tail = tl
    }
  }
```

This defines List as a covariant type with head and tail members. The nil object defines an empty list of element type Nothing, which is Scala's bottom type. Its members are both implemented as infinite loops (lacking exceptions that's the only way to produce a bottom type). The cons function produces a non-empty list.

In the DOT encoding of this example, the element type A becomes an abstract type member of List, which is now itself a type member of the object denoting the package. From the outside, List is only upper-bounded, so that it becomes nominal: the only way to construct a List is through the package field members nil and cons. Also, note that the covariance of the element type A is reflected because the tail of the List only requires the upper bound of the element type to hold.

```
let scala_collection_immutable = ν(sci) {
  List = μ(self: {A; isEmpty: bool.Booleam; head: self.A; tail: sci.List∧{A
        <: self.A}})
  nil: sci.List∧{A = ⊥} =
    let result = ν(self) {
      A = ⊥; isEmpty = bool.true; head = self.head; tail = self.tail }
    in result
  cons: ∀(x: {A})∀(hd: x.A)∀(tl: sci.List∧{A <: x.A})sci.List∧{A <: x.A} =
    λ(x: {A})λ(hd: x.A)λ(tl: sci.List∧{A <: x.A})
      let result = ν(self) {
        A = x.A; isEmpty = bool.false; head = hd; tail = tl }
      in result
}: { μ(sci: {
  List <: μ(self: {A; head: self.A; tail: sci.List∧{A <: self.A}})
  nil: sci.List∧{A = ⊥}
  cons: ∀(x: {A})∀(hd: x.A)∀(tl: sci.List∧{A <: x.A})sci.List∧{A <: x.A}
})}
in ...
```

As an example of a typing derivation in this code fragment consider the right hand side of the cons method. To show that result corresponds to the given result type sci.List∧{A <: x.A}, the typechecker proceeeds as follows. First step:

by typing of rhs
`result: ν(self: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}})`
by (Rec-E)
`result: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}}`
by (Sub), *since* self.A = x.A
`result: { A = x.A, hd: self.A, tl: sci.List∧{A = self.A}}`
by (Sub), *since* A: x.A..x.A <: A
`result: { A, hd: self.A, tl: sci.List∧{ A = self.A}}`
by (Rec-I)
`result: ν(self: { A, hd: self.A, tl: sci.List∧{ A = self.A}})`
by (Sub) *via* (<:-Sel) *on* sci.List
`result: sci.List`

Second step:

```
result: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}}
      by (SUB)
result: {A <: x.A}
```

Combining both steps with (AND-I) gives

```
result: sci.List∧{A <: x.A}
```

The explicit naming of the result was necessary so that we could "unwrap" the recursive type on it, apply subsumption and then "rewrap" using (REC-I) and (AND-I). In Scala, this sequence of steps can be an automatic conversion on the value; no separate let-binding is needed.

How close is this encoding to actual Scala? Here's legal Scala code which closely mimicks the previous list encoding in DOT.

```
object scala_collection_immutable { sci =>
  trait List { self =>
    type A
    def isEmpty: Boolean
    def head: self.A
    def tail: List{type A <: self.A}
  }

  def nil: sci.List{type A = Nothing} = new List{ self =>
    type A = Nothing
    def isEmpty = true
    def head: A = self.head
    def tail: List{type A = Nothing} = self.tail
  }

  def cons(x: {type A})(hd: x.A)(tl: sci.List{type A <: x.A})
     : sci.List{type A <: x.A} = new List{ self =>
    type A = x.A
    def isEmpty = false
    def head = hd
    def tail = tl
  }
}
```

The main difference between the DOT and Scala versions concerns recursive types. Scala does not support recusive types directly, but recursion is employed indirectly in the definition of traits and objects. Consequently, List in now a trait instead of a type definition, and scala_collection_immutable is an object instead of a let-bound variable. Otherwise, there are only minor syntactic differences.

6 Dependent Types in Scala

The previous section showed how generic types can be encoded as path-dependent types in DOT. So it established that DOT is sufficiently expressive to encode standard abstraction patterns we expect in programming languages today. In this section we show by means of an example[3] that path-dependent types are themselves a useful programming concept.

The example models command line options using heterogeneous maps. Say, you want to represent the settings given in the bash command line

```
ls -l --sort time --width=120
```

There are two settings: sort is associated with the string "time" and width is associated with the integer 120. We'd like to store these settings in a map in a typesafe way. This map is necessarily heterogeneous: if the key is "sort" it should accept and return a string as value, whereas for the "width" key it should accept and return integer values.

Here is the interface of a HMap trait for heterogeneous maps in Scala:

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}
```

Type dependencies are expressed with the Key type which contains a single type field Value to indicate the type of the value associated with that key. HMap contains two methods, get and add with result types that depend on the passed key argument.

To represent command line settings as HMap keys, we define a class Setting that extends Key and that defines a str field:

```
class Setting(val str: String) extends Key
```

Settings for sort and width define the setting name and the type of the values associated with the setting.

```
val sort = new Setting("sort") { type Value = String }
val width = new Setting("width") { type Value = Int }
```

The settings in the command line above can now be represented as:

```
val params = HMap.empty
  .add(width)(120)
  .add(sort)("time")
```

The system keeps track of the type associated with a setting, so the following two accesses of the `params` map both typecheck:

[3] The example is based on a talk by Jon Pretty (2015).

```
val x: Option[Int] = params.get(width)
val y: Option[String] = params.get(sort)
```

Here is a minimally complete implementation of **HMap** that makes this work:

```
trait HMap { self =>
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value) = new HMap {
    def get(k: Key) =
      if (k == key) Some(value.asInstanceOf[k.Value])
      else self.get(k)
  }
}

object HMap {
  def empty = new HMap { def get(k: Key) = None }
}
```

Note the **asInstanceOf** cast in the definition of **get**. This is necessary because the type checker cannot conclude that **k == key** implies **k.Value == key.Value**, i.e. that equality is extensional. Equality **(==)** is user-defined in Scala, so extensionality needs to be ensured in the definitions of the actual key types and values.

If we compare this to Haskell's **HMap** (van der Ploeg 2013) we notice several differences. First, Haskell, lacking dependent method types, expresses a dependent method type with an additional polymorphic parameter. Second, the type of keys in Haskell's implementation is fixed, whereas in Scala's approach it can be augmented with new information through subclassing, as was seen in the case of **Setting**. Third, in Haskell key creation is a monadic operation, so all code manipulating **HMap** has to be placed in a monad. By contrast, the Scala code is Monad-free (and also side-effect-free).

7 Implementation

The DOT calculus is at the core of the new *dotty* compiler for Scala (Odersky et al. 2013), which has been under development since 2013. *dotty* is currently used as an experimental platform to develop future versions of Scala. One of the first changes, influenced by DOT, is the introduction of true intersection types.

Unlike *nsc*, the current reference compiler for Scala, *dotty* does not maintain parameterized types and type applications. Instead, parameterized types in Scala sources are immediately mapped to types with abstract type members, and type applications are mapped to type refinements, very similar to what is shown in Sect. 5.

An important benefit of this reductionist approach is that it gives for the first time a satisfactory explanation of *wildcard types*, which arise when modeling type parameter variance. If we combine generics and subtyping, we inevitably face the problem that we want to express a generic type where the type argument is an unknown type that can range over a set of possible types. The prototypical case is where the argument ranges over all subtypes or supertypes of some type bound, as in List[_ <: Fruit] (or List<? extends Fruit> in Java).

Such partially undetermined types come up when we want to express variance. We would like to have that List[Apple] is a subtype of List[Fruit] since Apple is a subtype of Fruit. An equivalent way to state this is to say that the type List[Fruit] includes lists where the elements are of an arbitrary subtype of Fruit. The wildcard type List[_ <: Fruit] expresses that notion directly. *Definition-site variance* can be regarded as a user-friendly notation that expands into *use-site variance* expressions using wildcards.

The problem is how to model a wildcard type such as List[_ <: Fruit] in a formal type system. The original proposal of wildcard types by Igarashi and Viroli (Igarashi and Viroli 2002) interpreted them as existential types. However, the implementation of the feature in Java uses subtyping rules that are not explicable using just existential types. A tentative formalization exists with WildFJ (Torgersen et al. 2004), but the issues look quite complicated. Tate, Leung and Learner have explored some possible explanations (Tate et al. 2011); however their treatment raises about as many questions as it answers.

Say, C is a class with type parameter A, which can be declared nonvariant, covariant, or contravariant. The scheme used in dotty is to model C as a class with a single type member

```
class C { type A }
```

A type application C[T] is then expressed as C & { type A = T } if A is nonvariant, as C & { type A <: T } if C is covariant and as C & { type A >: T } if C is contravariant.

Wildcard type applications are mapped as they are written. For instance, C[_ <: T] would be expressed as C & { type A <: T }. So the previously thorny issue how to model wildcards has an almost trivially simple solution in this approach.

8 Related Work

The DOT calculus represents the latest development in a long-standing effort to formalize the core features of the Scala programming language. Previous formalizations include the νObj calculus (Odersky et al. 2003), Featherweight Scala (Cremet et al. 2006) and Scalina (Moors et al. 2008). The νObj calculus features a rich type language, including distinct notions of singleton types, type selections, record types, class types and compound types. However, this richness makes νObj rather unwieldy in practice and somewhat unsuitable as a core calculus. Furthermore, subtyping in νObj lacks unique upper or lower bounds, and mixin composition is not commutative. Type checking in νObj was shown to be undecidable, prompting the development of Featherweight Scala (Cremet et al. 2006) as an attempt at a calculus with decidable type checking. Scalina (Moors et al. 2008) was proposed as a formal underpinning for introducing higher-kinded types in Scala. Among these three systems, only the νObj has been proven sound, and its proof has not been machine-verified.

A common theme among earlier systems is their complexity, which makes them hard to reason about and extend with new features. The goal of DOT is to provide a more streamlined calculus that could serve as a basis for mechanized proofs of various extensions. To this end, DOT focuses on the essence of path-dependent types and reduces complexity as far as having only variables as paths. Despite and due to such restrictions, DOT is more general than previous models as it does not assume a class-based language or any other particular choice of implementation reuse. A preliminary version of DOT was formalized using a small-step operational semantics with a store (Amin et al. 2012) without giving a soundness proof. The first mechanized soundness proof was established for a more restricted variant (μDOT) using a closure-based big-step semantics (Amin et al. 2014). Soundness results with mechanized proofs for versions of System $D_{<:}$ and DOT similar to the ones presented here (but still based on big-step semantics) were only recently established (Rompf and Amin 2015).

Our work shares many goals with recent work on ML modules. In this context, path-dependent types go back at least to SML (Macqueen 1986), with foundational work on transparent bindings by Harper and Lillibridge (1994) and Leroy (1994). Unlike DOT, their systems are stratified, i.e. definitions can only depend on earlier definitions in the same type. This ensures well-formedness of definitions by construction, but rules out recursive definitions. Also, type bounds are not considered. MixML (Dreyer and Rossberg 2008) drops the stratification requirement between module and term language and enables modules as first class values as well as mixin-style recursive definitions. More recent work models key aspects of modules as extensions of System F (Montagu and Rémy 2009; Rossberg et al. 2014). The latest development, 1 ML (Rossberg 2015), unifies the ML module and core languages through an elaboration to System F_ω. Our work differs from these in its integration of subtyping and recursive definitions, as well as an overriding focus on keeping the formalism minimal.

In object-oriented programming languages, Ernst first proposed path-dependent types for family polymorphism (2001) and virtual classes (2003) in gbeta. Calculi for virtual classes include *vc* (Ernst et al. 2006) and Tribe (Clarke et al. 2007). Virtual classes abstract not only over types, but also over classes and inheritance – requiring additional restrictions or type machinery to control interactions between statically unknown class hierarchies. In contrast, the core DOT calculus does not model inheritance by design.

Abstract type tags $\{A\colon S..T\}$ provide a form of first-class subtyping constraints in DOT. As illustrated in Sect. 3, this subsumes the type of bounded polymorphism found in e.g. $F_{<:}$. However, unlike $F_{<:}$, DOT allows abstract types to be lower-bounded, and even to assume absurd bounds, such as $\{A\colon \top..\bot\}$. This makes the type system more expressive, but complicates the meta theory and renders reduction under abstraction unsafe in general (see Sect. 4). In DOT, these issues are solved by adopting a weak-reduction strategy and ensuring type safety only in run-time typing contexts (i.e. those that correspond to a store). Cretin, Scherer and Rémy propose an alternative approach in their F_{cc} and F_{th} systems (Cretin and Rémy 2014; Scherer and Rémy 2015). They consider a more general notion of *coercion constraints* which come in two flavors: consistent constraints, under which reduction is safe, and (potentially) inconsistent constraints, under which reduction is forbidden. Their approach thus regains a more flexible reduction strategy at the expense of a considerably more complex type system.

9 Conclusion and Future Work

We have developed DOT, a minimal formal foundation for Scala. The calculus departs from standard practice in that it places type names related by subtyping constraints and path-dependent types at the center and regards type parameterization as a secondary feature that can be expressed through encodings.

Developing a sound meta-theory for DOT has been difficult because the issues were at first poorly understood. In retrospect, the main difficulty came from the fact that the combination of upper and lower type bounds[4] allows programmers to define their own subtyping theory and that theory might well be inconsistent. An inconsistent subtyping theory arises from *bad bounds*, where the lower bound of a type is not a subtype of its upper bound. Through transitivity of subtyping, bad bounds can turn every type into a subtype of every other type, which makes the progress part of type soundness fail. Because of recursion and intersection types, it's impossible to stratify the system so that bad bounds are ruled out *a priori*.

We solve the issue by carefully distinguishing values that can arise at run-time from other terms. Run-time values carrying types are always constructed using a ν-term and its typing rule makes sure that every type member is defined as an alias of a concrete type. So run-time values cannot have bad bounds. The tricky aspect of the meta theory was how to exploit this intuition in the formal proofs.

The intuitions gained by DOT feed into Scala in several ways. For one, Scala is set to adopt constructs studied in DOT, such as full intersection types. Furthermore, the *dotty* Scala compiler has an internal type representation that generally resembles DOT and in particular encodes type parameterization as membership.

We also plan to study several extensions of DOT with the aim of bridging the gaps between the calculus and the programming language. Areas to be studied include formalizations of type parameters, variance, traits and classes, as well as inheritance. We expect that most of these extensions will be formalized through encodings into the base calculus. Their soundness will likely be established by proving type preservation of the encodings instead of adding to the meta-theory of DOT itself.

Acknowledgments. Adriaan Moors, Donna Malayeri and Geoffrey Washburn have contributed to previous versions of DOT. We thank the anonymous reviewers of Wadlerfest for helpful comments on this paper. For insightful discussions we thank Amal Ahmed, Derek Dreyer, Erik Ernst, Matthias Felleisen, Paolo Giarrusso, Scott Kilpatrick, Grzegorz Kossakowski, Alexander Kuklev, Viktor Kuncak, Jon Pretty,

[4] Having both kind of bounds is essential in DOT to model type aliases; if we would replace bounds by aliases we would run into the same problems.

Didier Rémy, Lukas Rytz, Miles Sabin, Ilya Sergey, Jeremy Siek, Josh Suereth and Phil Wadler. Contributors to the Dotty project include Dmitry Petrashko, Guillaume Martres, Vladimir Nikolaev, Ondřej Lhoták, Vera Salvisberg and Jason Zaugg. This research was supported by the European Research Council (ERC) under grant 587327 DOPPLER and the Swiss National Science Foundation under grant "Foundations of Scala".

A Additional Rules and Detailed Proofs

A.1 Typing of Dependent Sums/$\boldsymbol{\Sigma}$s

There are one subtyping rule and four typing rules (all admissible in D$_{<:}$) associated with the encoding of dependent sums given in Sect. 2.1.

$$\frac{\Gamma \vdash S_1 <: S_2 \qquad \Gamma,\ x : S_1 \vdash T_1 <: T_2}{\Gamma \vdash \Sigma(x : S_1)T_1 <: \Sigma(x : S_2)T_2} \tag{Σ-<:-Σ}$$

$$\frac{\Gamma \vdash x : S \qquad \Gamma \vdash y : T \qquad x \notin fv(S)}{\Gamma \vdash \ \mathbf{pack}\,[x, y]\,\mathbf{as}\,\Sigma(x : S)T : \Sigma(x : S)T} \tag{Σ-I}$$

$$\frac{\Gamma \vdash t : \Sigma(x : S)T \qquad \Gamma,\ x : S,\ y : T \vdash u : U \qquad x, y \notin fv(U)}{\Gamma \vdash \ \mathbf{unpack}\ \ x : S, y : T = t\,\mathbf{in}\,u : U} \tag{Σ-E}$$

$$\frac{\Gamma \vdash z : \Sigma(x : S)T}{\Gamma \vdash z.1 : S} \tag{Σ-Proj$_1$}$$

$$\frac{\Gamma \vdash z : \Sigma(x : S)T \qquad x \notin fv(T)}{\Gamma \vdash z.2 : T} \tag{Σ-Proj$_2$}$$

The proofs of admissibility are left as an (easy) exercise to the reader.

A.2 Proof of Theorem 2

We want to show that the translation $-^*$ preserves typing, i.e. if $\Gamma \vdash_F t : T$ then $\Gamma^* \vdash_D t^* : T^*$.

We start by stating and proving two helper lemmas. Write $[x.A := U]T$ for the capture-avoiding substitution of $x.A$ for U in T.

Lemma 6. *Substitution of type variables for types commutes with translation of types, i.e.*

$$([X := U]T)^* = [x_X.A := U^*]T^*$$

Proof. The proof is by straight-forward induction on the structure of T.

Lemma 7. *The following subtyping rules are admissible in D$_{<}$:*

$$\frac{\Gamma\ x : \{A : U..U\}}{\Gamma\ [x.A := U]T <: T} \qquad\qquad \frac{\Gamma\ x : \{A : U..U\}}{\Gamma\ T <: [x.A := U]T}$$

Proof. The proof is by induction on the structure of T.

Proof (Theorem 2). Proof is by induction on ($F_{<:}$) typing derivations. The case for subsumption follows immediately from preservation of subtyping (Theorem 1). The only remaining non-trivial cases are type and term application.

Term Application Case. We need to show that

$$\frac{\Gamma^*\ s^* : \forall(z : S^*)T^* \qquad \Gamma^*\ t^* : S^*}{\Gamma^*\ \mathbf{let}\, x = s^*\, \mathbf{in}\ \mathbf{let}\, y = t^*\, \mathbf{in}\, x\ y : T^*}$$

is admissible. First note that z is not in $fv(T^*)$, hence $[z := y]T^* = T^*$ for all y. In particular, using (Var) and (All-E), we have

$$\frac{\Gamma'\ x : \forall(z : S^*)T^* \qquad \Gamma'\ y : S^*}{\Gamma'\ x\ y : T^*}\ (\text{All-E})$$

where $\Gamma' = \Gamma^*, x : \forall(z : S^*)T^*, y : S^*$. Applying the induction hypothesis, context weakening and (Let), we obtain

$$\frac{\Gamma^*, x : \forall(z : S^*)T^*\ t^* : S^* \qquad \Gamma'\ x\ z : T^*}{\Gamma^*, x : \forall(z : S^*)T^*\ \mathbf{let}\, y = t^*\, \mathbf{in}\, x\ y : T^*}\ (\text{Let})$$

The result follows by applying the induction hypothesis once more:

$$\frac{\Gamma^*\ s^* : \forall(z : S)^*T^* \qquad \Gamma^*, x : \forall(z : S)^*T^*\ \mathbf{let}\, y = t^*\, \mathbf{in}\, x\ y : T^*}{\Gamma^*\ \mathbf{let}\, x = s^*\, \mathbf{in}\ \mathbf{let}\, y = t^*\, \mathbf{in}\, x\ y : T^*}\ (\text{Let})$$

Type Application Case. By preservation of subtyping, we have $\Gamma^* \vdash U^* <: S^*$, hence it suffices to show that

$$\frac{\Gamma^*\ t^* : \forall(x_X : \{A : \bot..S^*\})T^* \qquad \Gamma^*\ U^* <: S^*}{\Gamma^*\ \mathbf{let}\, x = t^*\, \mathbf{in}\ \mathbf{let}\, y_Y = \{A = U^*\}\, \mathbf{in}\, x\ y_Y : ([X := U]T)^*}$$

is admissible. By context weakening, (Typ-<:-Typ), (Bot) and (Sub), we have

$$\frac{\Gamma'\ y_Y : \{A : U^*..U^*\} \qquad \dfrac{\Gamma'\ \bot <: U^* \qquad \Gamma'\ U^* <: S^*}{\Gamma'\ y_Y : \{A : \bot..S^*\}}\ (\text{Typ-<:-Typ})}{\Gamma'\ y_Y : \{A : \bot..S^*\}}\ (\text{Sub})$$

where $\Gamma' = \Gamma^*, x : \forall(x_X : \{A : \bot..S^*\})T^*, y_Y : \{A : U^*..U^*\}$. By (Var) and (All-E), we have

$$\frac{\Gamma'\; x : \forall(x_X : \{A : \bot..S^*\})T^* \qquad \Gamma'\; y_Y : \{A : \bot..S^*\}}{\Gamma'\; x\; y_Y : [x_X := y_Y]T^*}\;(\text{All-E})$$

Next, we observe that, by Lemma 6

$$\begin{aligned}([X := U]T)^* &= [x_X.A := U^*]T^* \\ &= [y_Y.A := U^*][x_X := y_Y]T^*\end{aligned}$$

By Lemma 7, (Var) and (Sub), we thus have

$$\frac{\Gamma'\; xy_Y : [x :=_]\; Xy_YT^* \qquad \dfrac{\Gamma'\; y_Y : \{A : U^*..U^*\}}{\Gamma'\; [x_X := y_Y]T^* <: ([X := U]T)^*}\;(\text{Lemma 7})}{\Gamma'\; x\; y_Y : ([X := U]T)^*}\;(\text{Sub})$$

The result follows from applying (Let) twice. Note that, being fresh, neither x nor y_Y appear free in $([X := U]T)^*$, hence the hygiene condition of (Let) holds.

References

Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. **15**(4), 575–631 (1993)

Amin, N., Moors, A., Odersky, M.: Dependent object types. In: FOOL (2012)

Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: OOPSLA (2014)

Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: POPL (1995)

Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system F with subtyping. Inf. Comput. **109**(1/2), 4–56 (1994)

Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: a simple virtual class calculus. In: AOSD (2007)

Coppo, M., Dezani-Ciancaglini, M., Sallé, P.: Functional characterization of some semantic equalities inside lambda-calculus. In: Automata, Languages and Programming, 6th Colloquium (1979)

Cremet, V., Garillot, F., Lenglet, S., Odersky, M.: A core calculus for scala type checking. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 1–23. Springer, Heidelberg (2006)

Cretin, J., Rémy, D.: System F with coercion constraints. In: CSL-LICS (2014)

Dreyer, D., Rossberg, A.: Mixin' up the ML module system. In: ICFP (2008)

Ernst, E.: Family polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, p. 303. Springer, Heidelberg (2001)

Ernst, E.: Higher-order hierarchies. In: ECOOP (2003)

Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL (2006)

Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993)

Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: POPL (1994)

Adsul, B., Viroli, M.: On variance-based subtyping for parametric types. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, p. 441. Springer, Heidelberg (2002)

Leroy, X.: Manifest types, modules and separate compilation. In: POPL (1994)

Macqueen, D.: Using dependent types to express modular structure. In: POPL (1986)

Montagu, B., Rémy, D.: Modeling abstract types in modules with open existential types. In: POPL (2009)

Moors, A., Piessens, F., Odersky, M.: Safe type-level abstraction in scala. In: FOOL (2008)

Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: ECOOP (2003)

Odersky, M., Petrashko, D., Martres, G., others.: The dotty project (2013). https://github.com/lampepfl/dotty

van der Ploeg, A.: The HMap package (2013). https://hackage.haskell.org/package/HMap

Pretty, J.: Minimizing the slippery surface of failure. Talk at Scala World (2015). https://www.youtube.com/watch?v=26UHdZUsKkE

Rompf, T., Amin, N.: From F to DOT: Type soundness proofs with definitional interpreters. Purdue University, Technical report (2015). http://arxiv.org/abs/1510.05216

Rossberg, A.: 1ML - core and modules united (f-ing first-class modules). In: ICFP (2015)

Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. J. Funct. Program. **24**(5), 529–607 (2014)

Scherer, G., Rémy, D.: Full reduction in the face of absurdity. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 685–709. Springer, Heidelberg (2015)

Tate, R., Leung, A., Lerner, S.: Taming wildcards in java's type system. In: PLDI (2011)

Torgersen, M., Ernst, E., Hansen, C.P.: WildFJ. In: FOOL (2004)

Linear $\lambda\mu$ is CP (more or less)

Jennifer Paykin(✉) and Steve Zdancewic

University of Pennsylvania, Philadelphia, PA 19104, USA
jpaykin@seas.upenn.edu, stevez@cis.upenn.edu

Abstract. In this paper we compare Wadler's CP calculus for classical linear processes to a linear version of Parigot's $\lambda\mu$ calculus for classical logic. We conclude that linear $\lambda\mu$ is "more or less" CP, in that it equationally corresponds to a polarized version of CP. The comparison is made by extending a technique from Melliès and Tabareau's tensor logic that correlates negation with polarization. The polarized CP, which is written CP$^{\pm}$ and pronounced "CP more or less," is an interesting bridge in the landscape of Curry-Howard interpretations of logic.

1 Introduction

In 2012 Philip Wadler introduced CP, a language of "classical processes" in the style of Caires and Pfenning's session-typed processes (2010). CP is a recent advance in the long and distinguished line of work that connects logic to computation via the Curry-Howard correspondence. In this instance, Wadler connects Girard's (classical) linear logic (1987) to a variant of Milner's π-calculus (1992) by using types to express communication protocols—sessions—that synchronize concurrent threads of execution. The connection to logic endows CP with strong correctness properties: type safety and deadlock freedom.

Over the years, research into the Curry-Howard isomorphism has built up a vast landscape of languages and logics. One particularly rich and relevant domain is calculi typed by classical (but not necessarily linear) logic. Among these are Griffin's $\lambda_{\mathcal{C}}$-calculus (1990), Parigot's $\lambda\mu$-calculus (1992), Curien and Heberlin's $\overline{\lambda}\mu\tilde{\mu}$-calculus/System L (2000), and Wadler's own dual calculus (2003). With CP in mind, we can even consider linear versions of these calculi, such as Linear L (Munch-Maccagnoni 2009; Spiwack 2014) and linear $\lambda\mu$, the natural linear variant of Parigot's $\lambda\mu$-calculus.

A natural question then arises:

How does CP *relate to linear interpretations of classical calculi?*

In this paper, we answer the question by showing that CP is *almost* (but not quite) the same thing as linear $\lambda\mu$. In particular, linear $\lambda\mu$ corresponds exactly with a *polarized* version of CP, which we introduce here and dub CP$^{\pm}$. Compared to Wadler's original CP, CP$^{\pm}$ (which can be read "CP more or less") separates positive (sending) from negative (receiving) types and inserts shift connectives between them. Operationally, polarization reduces the amount of

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 273–291, 2016.
DOI: 10.1007/978-3-319-30936-1_15

nondeterminism, statically making more choices about how processes synchronize.[1] CP$^{\pm}$ is thus "more or less" CP, where the scheduling is statically decided. We summarize our results in the following Wadleresque slogan:

Linear $\lambda\mu$ *is* CP *(more or less).*

The relationship between linear $\lambda\mu$ and CP hinges on *duality*. Linear $\lambda\mu$ is a classical logic of two-sided sequents, while CP (and consequently CP$^{\pm}$) is a classical logic of one-sided sequents. To relate them, we follow a plan laid out by Melliès and Tabareau (2010) in the context of *tensor logic*, which similarly comes in two flavors. The two-sided presentation of tensor logic is an intuitionistic logic of (non-involutive) negation; the one-sided presentation is a polarized logic of (non-invertible) shift operators. When linear $\lambda\mu$ takes the place of the two-sided presentation, CP$^{\pm}$ arises naturally from the dualization procedure.

The contributions of this paper are summarized as follows. Section 2 presents a core formulation of Wadler's CP, using a novel and elegant operational semantics. Section 3 introduces linear $\lambda\mu$, including both call-by-value and call-by-name operational semantics. We define CP$^{\pm}$ in Sect. 4, first describing the duality derived from tensor logic, and then inferring the syntax and semantics from the type structure. Section 5 gives the main result, establishing the equational correspondence between linear $\lambda\mu$ and CP$^{\pm}$. We briefly conclude with a discussion of the relationships between CP, CP$^{\pm}$, and other computational interpretations of classical logics.

2 CP and Session Types

In this section we describe a variation on Wadler's CP calculus (2012; 2014), based on the line of work by Caires and Pfenning (2010) that treats the types of linear logic as descriptions of session protocols and the proofs of linear logic as processes obeying these protocols. A (binary) session is just a communication channel between two processes. The types of linear logic describe the protocols by which these channels should behave.

$$X, Y ::= 1 \mid X \otimes Y \mid \bot \mid X \mathbin{⅋} Y \mid 0 \mid X \oplus Y \mid \top \mid X \mathbin{\&} Y$$

Consider a process P that offers a channel x obeying protocol X. The process at the other end of the channel, Q, must obey the *dual* protocol $X^{\perp}$. Duality is involutive—that is, we have $(X^{\perp})^{\perp} = X$. We write the composition of these two processes as $\nu x.\langle P \mid Q\rangle$, and the session type X describes their behavior as shown in Fig. 1.

Each connective has an associated direction—processes *send* messages over channels behaving like a tensor $\otimes$, but they *receive* messages over channels behaving like a par ⅋. However, the direction of each channel is not fixed. For example, a process may send a value on a channel, then immediately receive a flag on the same channel. The session-based type of such a channel would be $X_1 \otimes (X_2 \mathbin{\&} X_3)$.

[1] Pfenning and Griffith (2015) have also studied polarization in their work on intuitionistic session types, where it distinguishes synchronous and asynchronous communication.

$x : X$	P	$x : X^{\perp}$	Q
1	sends an empty message over x and immediately halts	$\bot$	receives an empty message on x and closes the channel
$X_1 \otimes X_2$	sends a channel obeying X_1 over x, then behaves like X_2	${X_1}^{\perp} \parr {X_2}^{\perp}$	receives a channel obeying ${X_1}^{\perp}$ over x, then behaves like ${X_2}^{\perp}$
0	(no channels obey 0)	$\top$	aborts
$X_1 \oplus X_1$	sends a flag indicating whether x will behave as X_1 or X_2	${X_1}^{\perp} \,\&\, {X_2}^{\perp}$	receives a flag indicating that x will behave as ${X_1}^{\perp}$ or ${X_2}^{\perp}$

Fig. 1. Session types and their intended semantics. Process P interacts with Q via the cut $\nu x.\langle P \mid Q\rangle$. By duality, P could equally well offer a channel behaving as $X^{\perp}$.

2.1 Syntax and Static Semantics

As in the π-calculus, a process P is made up of bodies and prefixes to manipulate channels.

$$\begin{array}{rll} P ::= & x \leftrightarrow y & \mid \nu x.\langle P_1 \mid P_2\rangle \\ & \mid x[].0 & \mid x().P \\ & \mid x[y].(P_1 \mid P_2) & \mid x(y).P \\ & & \mid x.\mathbf{case}() \\ & \mid x[\mathbf{inj}_i].P & \mid x.\mathbf{case}(P_1 \mid P_2) \end{array}$$

The link operator $x \leftrightarrow y$ connects the channels x and y together. The cut operator $\nu x.\langle P_1 \mid P_2\rangle$ synchronizes communication on a channel whose endpoints lie in P_1 and P_2. The other syntactic forms are *prefix actions*, which send or receive on channels according to one of the session protocols.

The typing judgment $P \vdash \Omega$ specifies that the channels offered by P obey certain protocols, specified by Ω. Here Ω is a collection of channel names x with their types X. The typing rules for CP are shown in Fig. 2.

2.2 Operational Semantics

Wadler (2012) nicely summarizes the Curry-Howard interpretation of session types as follows: propositions are session types, proofs are processes, and *cut elimination is communication.* The operational behavior of processes is the description of how communication occurs over channels. The end goal of this communication is a process that is free of both cuts and (non-atomic) links. With that goal in mind, each step of communication falls into one of three groups.

First, β-reduction rules describe the synchronous communication between two processes.

$$\begin{aligned} \nu x.\langle y \leftrightarrow x \mid P\rangle &\to_\beta P\{y/x\} \\ \nu x.\langle x[].0 \mid x().P\rangle &\to_\beta P \\ \nu x.\langle x[y].(P_1 \mid P_2) \mid x(y).P_3\rangle &\to_\beta \nu y.\langle P_1 \mid \nu x.\langle P_2 \mid P_3\rangle\rangle \\ \nu x.\langle x[\mathbf{inj}_i].P \mid x.\mathbf{case}(P_1 \mid P_2)\rangle &\to_\beta \nu x.\langle P \mid P_i\rangle \end{aligned}$$

$$\frac{}{y \leftrightarrow x \vdash y : A^\perp, x : A}\ \text{AX} \qquad \frac{P_1 \vdash \Omega_1, x : A \quad P_2 \vdash x : A^\perp, \Omega_2}{\nu x.\langle P_1 \mid P_2\rangle \vdash \Omega_1, \Omega_2}\ \text{CUT}$$

$$\frac{}{x[].0 \vdash x : 1}\ 1 \qquad \frac{P \vdash \Omega}{x().P \vdash x : \bot, \Omega}\ \bot$$

$$\frac{P_1 \vdash \Omega_1, y : A \quad P_2 \vdash \Omega_2, x : B}{x[y].(P_1 \mid P_2) \vdash \Omega_1, \Omega_2, x : A \otimes B}\ \otimes \qquad \frac{P \vdash y : A, x : B, \Omega}{x(y).P \vdash x : A \parr B, \Omega}\ \parr$$

$$\text{(no 0 rule)} \qquad \frac{}{x.\mathbf{case}() \vdash x : \top, \Omega}\ \top$$

$$\frac{P \vdash \Omega, x : A_i}{x[\mathbf{inj}_i].P \vdash \Omega, x : A_1 \oplus A_2}\ \oplus \qquad \frac{P_1 \vdash x : A, \Omega \quad P_2 \vdash x : B, \Omega}{x.\mathbf{case}(P_1 \mid P_2) \vdash x : A \,\&\, B, \Omega}\ \&$$

Fig. 2. The CP calculus

Structural equivalence rules allow us to swap the order of cuts and forwarding links in these rules to avoid unnecessary duplication.

$$\nu x.\langle P_1 \mid P_2\rangle \equiv_s \nu x.\langle P_2 \mid P_1\rangle \qquad x \leftrightarrow y \equiv_s y \leftrightarrow x \tag{1}$$

Second, link forwarding is defined by η-expansion rules.

$$(y \leftrightarrow x \vdash y : \bot, x : 1) \rightarrow_\eta y().x[].0$$
$$(y \leftrightarrow x \vdash y : X^\perp \parr Y^\perp, x : X \otimes Y) \rightarrow_\eta y(y').x[x'].(y' \leftrightarrow x' \mid y \leftrightarrow x)$$
$$(y \leftrightarrow x \vdash y : \top, x : 0) \rightarrow_\eta y.\mathbf{case}()$$
$$(y \leftrightarrow x \vdash y : A^\perp \,\&\, B^\perp, x : A \oplus B) \rightarrow_\eta y.\mathbf{case}(x[\mathbf{inj}_1].y \leftrightarrow x \mid x[\mathbf{inj}_2].y \leftrightarrow x)$$

The rest of the operational behavior of processes deals with *commuting conversions*, by which non-interfering actions can be executed in any order. Presentations of commuting conversions are often very intricate, as there are a quadratic number of interleavings of non-interfering actions. Here, we present a unified view of commuting conversions by classifying them into three groups.

The first group of commuting conversions permutes an action (corresponding to one of the session protocols) around a cut. For example, when x does not occur in P_2 and y does not occur in P_1, then

$$\nu x.\langle P_1 \mid y[z].(P_2 \mid P_3)\rangle \rightarrow_{CC} y[z].(P_2 \mid \nu x.\langle P_1 \mid P_3\rangle).$$

The presence of the additives, $\top$ and $\&$, adds even stranger looking conversions:

$$\nu x.\langle y.\mathbf{case}() \mid Q\rangle \rightarrow_{CC} y.\mathbf{case}()$$
$$\nu x.\langle y.\mathbf{case}(P_1 \mid P_2) \mid Q\rangle \rightarrow_{CC} y.\mathbf{case}(\nu x.\langle P_1 \mid Q\rangle \mid \nu x.\langle P_2 \mid Q\rangle)$$

This class of conversions is *directed*, to push the cut after the action so that it becomes closer to its own synchronizing channel. The combination of this class of conversions with the β-reduction rules results in a cut elimination procedure (Theorem 1).

The second group of commuting conversions says that two cuts on different channels can be performed in either order. Assuming that x does not occur in P_2 and y does not occur in P_1, we have

$$\nu x.\langle P_1 \mid \nu y.\langle P_2 \mid P_3\rangle\rangle \equiv_{CC} \nu y.\langle P_2 \mid \nu x.\langle P_1 \mid P_3\rangle\rangle.$$

This type of conversion is classified by Wadler as a structural equivalence, which we denote $\equiv_s$. However, while the two structural equivalences in Eq. (1) are necessary to define the normalization or communication procedure of processes, the *conversion* in this section is not.

The third class of conversions says that prefix actions on different channels can occur in any order. For example, if x (which is not equal to z) does not occur in P_1, then

$$x[\mathbf{inj}_1].z[y].(P_1 \mid P_2) \equiv_{CC} z[y].(P_1 \mid x[\mathbf{inj}_1].P_2).$$

The motivation behind this type of commuting conversion is the independence of non-interfering sessions. That is, when two prefixes do not refer to each other, they should not interfere. Again, this class of conversions is undirected; it is not necessary for the definition of a communcation/cut-elimination procedure. The relation $\equiv_{CC}$ instead defines *behavioral equivalence* of processes, as described by Pérez et al. (2014).

Commuting Conversions via Contexts. A context C is any process with holes (possibly zero or more than one) for other processes. The number of holes in C is called its arity. We write $C[P_1, .., P_i]$ for the operation that fills holes with processes.[2] Two contexts are disjoint, written $C_1 \perp C_2$, if the set of channels referred to by C_1 is disjoint from the set of channels referred to by C_2.

The composition of two contexts $C_2 \circ C_1$ replicates the inner context C_1 in all of the holes of the outer context C_2. This is illustrated by the following examples:

$$\begin{gathered}\nu x.\langle \Box \mid P\rangle \circ C = \nu x.\langle C \mid P\rangle \\ x.\mathbf{case}() \circ C = x.\mathbf{case}() \\ x.\mathbf{case}(\Box \mid \Box) \circ C = x.\mathbf{case}(C \mid C)\end{gathered}$$

The arity of $C_1 \circ C_2$ is the product of the arities of the two subcontexts.

For any context C with a single hole, we have a congruence rule to reduce processes under actions.

$$\frac{P \equiv_s P'}{C[P] \equiv_s C[P']} \tag{2}$$

[2] To give the definitions in this section precisely, it would be necessary to use named holes and substitutions to fill in the holes. For the purposes of this paper we leave these operations informal.

To define commuting conversions, we pick out certain classes of contexts that refer to the different components of the commuting conversion rules. Evaluation contexts, written C^e, consist of cuts with a single top-level hole, and action contexts, written C^a, consist of prefix actions also with top-level holes.

$$\begin{aligned} C^e &::= \nu x.\langle \Box \mid P_2 \rangle \mid \nu x.\langle P_1 \mid \Box \rangle \\ C^a &::= x().\Box \mid x[y].(\Box \mid P_2) \mid x[y].(P_1 \mid \Box) \mid x(y).\Box \\ &\quad \mid x.\mathbf{case}() \mid x[\mathbf{inj}_i].\Box \mid x.\mathbf{case}(\Box \mid \Box) \end{aligned}$$

Notice that $x.\mathbf{case}()$ is a context with arity zero, and $x.\mathbf{case}(\Box \mid \Box)$ is a context with arity two.

The three classes of commuting conversion rules can be summarized neatly as follows:

$$\frac{C^e \bot C^a}{C^e \circ C^a[P_1, .., P_i] \rightarrow_{CC} C^a \circ C^e[P_1, .., P_i]} \quad \text{ACTION-CUT}$$

$$\frac{C^e_1 \bot C^e_2}{C^e_1 \circ C^e_2[P] \equiv_{CC} C^e_2 \circ C^e_1[P]} \quad \text{CUT-CUT}$$

$$\frac{C^a_1 \bot C^a_2}{C^a_1 \circ C^a_2[P_1, .., P_i] \equiv_{CC} C^a_2 \circ C^a_1[P_1, .., P_i]} \quad \text{ACTION-ACTION}$$

2.3 Reduction and Equivalence

The step relation on processes is the least relation generated by $\rightarrow_\beta$, $\rightarrow_\eta$, $\rightarrow_{CC}$, and $\equiv_s$. Behavioral equivalence, written $\equiv$, is defined to be the least congruence containing the step relation and $\equiv_{CC}$.

Theorem 1 (Progress and Preservation).

1. *If $P \vdash \Omega$ then either P is cut- and link-free, or P can take a step.*
2. *If $P \vdash \Omega$ and P steps to P', then $P' \vdash \Omega$.*

The structural equivalence relation $\equiv_s$ makes reduction in CP extremely non-deterministic. We conjecture that the step relation is confluent up to commuting conversions $\equiv_{CC}$, but have not proved this fact. In addition, there is no meaningful notion of evaluation order; the properties of call-by-name and call-by-value are not even expressible. In Sect. 4 we will see that CP$^{\pm}$ has the ability to resolve this non-determinism by employing different evaluation strategies, in the style of Wadler's dual calculus (2003).

3 Linear $\lambda\mu$ and Linear Classical Logic

In this section we answer the question: what is a linearized classical language, as opposed to a language for classical linear logic? Just as the simply-typed

λ-calculus can be *linearized* to obtain a calculus for intuitionistic linear logic, we can similarly linearize Parigot's $\lambda\mu$-calculus for classical logic (1992). Such systems have been considered before, for example by Munch-Maccagnoni (2009) and Spiwack (2014).

Following Wadler (2005), we take the types of linear $\lambda\mu$ to consist of negation, units, products and sums—implication is defined in terms of negation and products. Products and sums are written in the style of linear logic.

$$A, B ::= \neg A \mid 1 \mid A \otimes B \mid 0 \mid A \oplus B.$$

3.1 Syntax and Static Semantics

The syntax of $\lambda\mu$ consists of two parts: terms and commands. Terms t are typed expressions, while commands c have no types, but instead define a relation between the two different sorts of variables. Term variables, written x, can be thought of in the usual way—as providing input to an expression. Continuation variables, written α, consume the output of an expression.

The quintessential command feeds a term t into a continuation variable α, and is written $[\alpha]t$. On the other hand, if a continuation variable α is used in a command c, the term $\mu\alpha.c$ extracts the value passed to α inside of c.

The syntax of terms and commands is summarized below. We use expressions e to refer to either syntactic form.

$$\begin{array}{rll}
t ::= & x & \mid \mu\alpha.c \\
\mid & \overline{\lambda} x.c & \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \\
\mid & () & \mid \textbf{let}\ () = t_1\ \textbf{in}\ t_2 \\
\mid & (t_1, t_2) & \mid \textbf{let}\ (x_1, x_2) = t_1\ \textbf{in}\ t_2 \\
 & & \mid \textbf{case}\ t\ \textbf{of}\ () \\
\mid & \textbf{inj}_i\ t & \mid \textbf{case}\ t\ \textbf{of}\ (\textbf{inj}_1\ x_1 \to t_1, \textbf{inj}_2\ x_2 \to t_2) \\
c ::= & [\alpha]t & \\
\mid & t_1\ t_2 & \mid \textbf{let}\ x = t\ \textbf{in}\ c \\
\mid & \textbf{let}\ () = t\ \textbf{in}\ c & \mid \textbf{let}\ (x_1, x_2) = t\ \textbf{in}\ c \\
\mid & \textbf{case}\ t\ \textbf{of}\ () & \mid \textbf{case}\ t\ \textbf{of}\ (\textbf{inj}_1\ x_1 \to c_1, \textbf{inj}_2\ x_2 \to c_2)
\end{array}$$

There are similarly two typing judgments: $\Gamma \vdash t : A \mid \Pi$ for terms, and $\Gamma \vdash c \mid \Pi$ for commands. Here Γ is a context of term variables x, and Π is a context of continuation variables α. For rules that are polymorphic in the judgment, we write $\Gamma \vdash e : \Theta \mid \Pi$, where Θ is a stoup of either zero or one types.

Figure 3 shows the typing rules. Many of the rules are unsurprising for a linear λ-calculus. Note that $\overline{\lambda}$ is used for negation abstraction instead of the usual λ, and every application is a command, not a term.

To understand how these language features interact, consider the encoding of linear implication $A \multimap B$ in classical logic as $\neg(A \otimes \neg B)$. We can encode the

$$\dfrac{}{x : A \vdash x : A \mid \cdot}\ \textsc{var} \qquad \dfrac{\mathit{\Gamma}_1 \vdash t : A \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2, x : A \vdash e : \mathit{\Theta} \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \mathbf{let}\, x = t\, \mathbf{in}\, e : \mathit{\Theta} \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \textsc{let}$$

$$\dfrac{\mathit{\Gamma} \vdash c \mid \mathit{\Pi}, \alpha : A}{\mathit{\Gamma} \vdash \mu\alpha.c : A \mid \mathit{\Pi}}\ \mu\text{-I} \qquad \dfrac{\mathit{\Gamma} \vdash t : A \mid \mathit{\Pi}}{\mathit{\Gamma} \vdash [\alpha]t \mid \mathit{\Pi}, \alpha : A}\ \mu\text{-E}$$

$$\dfrac{\mathit{\Gamma}, x : A \vdash c \mid \mathit{\Pi}}{\mathit{\Gamma} \vdash \overline{\lambda}x.c : \neg A \mid \mathit{\Pi}}\ \neg\text{-I} \qquad \dfrac{\mathit{\Gamma}_1 \vdash t_1 : \neg A \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2 \vdash t_2 : A \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash t_1\, t_2 \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \neg\text{-E}$$

$$\dfrac{}{\cdot \vdash () : 1 \mid \cdot}\ 1\text{-I} \qquad \dfrac{\mathit{\Gamma}_1 \vdash t : 1 \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2 \vdash e : \mathit{\Theta} \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \mathbf{let}\, () = t\, \mathbf{in}\, e : \mathit{\Theta} \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ 1\text{-E}$$

$$\dfrac{\mathit{\Gamma}_1 \vdash t_1 : A_1 \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2 \vdash t_2 : A_2 \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash (t_1, t_2) : A_1 \otimes A_2 \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \otimes\text{-I}$$

$$\dfrac{\mathit{\Gamma}_1 \vdash t : A \otimes B \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2, x : A, y : B \vdash e : \mathit{\Theta} \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \mathbf{let}\, (x, y) = t\, \mathbf{in}\, e : \mathit{\Theta} \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \otimes\text{-E}$$

$$\dfrac{\mathit{\Gamma} \vdash t : 0 \mid \mathit{\Pi}}{\mathit{\Gamma} \vdash \mathbf{case}\, t\, \mathbf{of}\, () : \mathit{\Theta} \mid \mathit{\Pi}}\ 0\text{-E} \qquad \dfrac{\mathit{\Gamma} \vdash t : A_i \mid \mathit{\Pi}}{\mathit{\Gamma} \vdash \mathbf{inj}_i\, t : A_1 \oplus A_2 \mid \mathit{\Pi}}\ \oplus\text{-I}$$

$$\dfrac{\mathit{\Gamma}_1 \vdash t : A_1 \oplus A_2 \mid \mathit{\Pi}_1 \quad \mathit{\Gamma}_2, x_1 : A_1 \vdash e_1 : \mathit{\Theta} \mid \mathit{\Pi}_2 \quad \mathit{\Gamma}_2, x_2 : A_2 \vdash e_2 : \mathit{\Theta} \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \mathbf{case}\, t\, \mathbf{of}\, (\mathbf{inj}_1\, x_1 \to e_1, \mathbf{inj}_2\, x_2 \to e_2) : \mathit{\Theta} \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \oplus\text{-E}$$

Fig. 3. The Linear $\lambda\mu$-calculus

application rule as

$$\dfrac{\mathit{\Gamma}_1 \vdash t_1 : A \multimap B \mid \mathit{\Pi}_1 \qquad \mathit{\Gamma}_2 \vdash t_2 : A \mid \mathit{\Pi}_2}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash t_1\, t_2 : B \mid \mathit{\Pi}_1, \mathit{\Pi}_2}\ \multimap\text{-E} \qquad \Rightarrow$$

$$\dfrac{\dfrac{\mathit{\Gamma}_1 \vdash t_1 : \neg(A \otimes \neg B) \mid \mathit{\Pi}_1 \qquad \dfrac{\mathit{\Gamma}_2 \vdash t_2 : A \mid \mathit{\Pi}_2 \qquad \dfrac{\dfrac{\dfrac{}{y : B \vdash y : B \mid \cdot}}{y : B \vdash [\beta]y \mid \beta : B}}{\cdot \vdash \overline{\lambda}y.[\beta]y : \neg B \mid \beta : B}}{\mathit{\Gamma}_2 \vdash (t_2, \overline{\lambda}y.[\beta]y) : A \otimes \neg B \mid \mathit{\Pi}_2, \beta : B}}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash t_1\, (t_2, \overline{\lambda}y.[\beta]y) \mid \mathit{\Pi}_1, \mathit{\Pi}_2, \beta : B}}{\mathit{\Gamma}_1, \mathit{\Gamma}_2 \vdash \mu\beta.t_1\, (t_2, \overline{\lambda}y.[\beta]y) : B \mid \mathit{\Pi}_1, \mathit{\Pi}_2}$$

We can also consider the encoding of Felleisen's $\mathcal{C}$ operator (1988), which is typed by double negation elimination.

$$\dfrac{\dfrac{\mathit{\Gamma} \vdash t : \neg\neg A \mid \mathit{\Delta} \qquad \dfrac{}{\cdot \vdash \overline{\lambda}x.[\alpha]x : \neg A \mid \alpha : A}}{\mathit{\Gamma} \vdash t\, (\overline{\lambda}x.[\alpha]x) \mid \mathit{\Delta}, \alpha : A}}{\mathit{\Gamma} \vdash \mu\alpha.t\, (\overline{\lambda}x.[\alpha]x) : A \mid \mathit{\Delta}}$$

3.2 Operational Semantics

In this section we define an operational semantics for linear $\lambda\mu$ in both call-by-value and call-by-name style. We first give the β and η rules that are shared by the two reduction strategies. The β rules are given in terms of let bindings, and the difference between CBV and CBN comes down to the treatment of let reduction and evaluation contexts, as shown in Fig. 4.

First, the β rules:

$$\begin{array}{lll}
(\mu) & - & \\
(\neg) & (\overline{\lambda} x.c)\, t & \rightarrow_\beta \mathbf{let}\, x = t\, \mathbf{in}\, c \\
(1) & \mathbf{let}\, () = ()\, \mathbf{in}\, e & \rightarrow_\beta e \\
(\otimes) & \mathbf{let}\, (x_1, x_2) = (t_1, t_2)\, \mathbf{in}\, e & \rightarrow_\beta \mathbf{let}\, x_1 = t_1\, \mathbf{in}\, \mathbf{let}\, x_2 = t_2\, \mathbf{in}\, e \\
(0) & - & \\
(\oplus) & \mathbf{case}\, \mathbf{inj}_i\, t\, \mathbf{of}\, (\mathbf{inj}_1\, x_1 \rightarrow e_1, \mathbf{inj}_2\, x_2 \rightarrow e_2) & \rightarrow_\beta \mathbf{let}\, x_i = t\, \mathbf{in}\, e_i
\end{array}$$

Next we have local η expansions:

$$\begin{array}{lll}
(\mu) & t : A & \rightarrow_\eta \mu\alpha.[\alpha]t \\
(\neg) & t : \neg A & \rightarrow_\eta \overline{\lambda} x.t\, x \\
(1) & t : 1 & \rightarrow_\eta \mathbf{let}\, () = t\, \mathbf{in}\, () \\
(\otimes) & t : A_1 \otimes A_2 & \rightarrow_\eta \mathbf{let}\, (x_1, x_2) = t\, \mathbf{in}\, (x_1, x_2) \\
(0) & t : 0 & \rightarrow_\eta \mathbf{case}\, t\, \mathbf{of}\, () \\
(\oplus) & t : A_1 \oplus A_2 & \rightarrow_\eta \mathbf{case}\, t\, \mathbf{of}\, (\mathbf{inj}_1\, x_1 \rightarrow \mathbf{inj}_1\, x_1, \mathbf{inj}_2\, x_2 \rightarrow \mathbf{inj}_2\, x_2)
\end{array}$$

CBV	CBN
$\mathbf{let}\, x = v\, \mathbf{in}\, e \rightarrow_\beta e\{v/x\}$	$\mathbf{let}\, x = t\, \mathbf{in}\, e \rightarrow_\beta e\{t/x\}$
$v ::= \overline{\lambda} x.c \mid () \mid (v_1, v_2) \mid \mathbf{inj}_i\, v$	$v ::= \overline{\lambda} x.c \mid () \mid (t_1, t_2) \mid \mathbf{inj}_i\, t$
$E ::= \Box \mid [\alpha]E$	$E ::= \Box \mid [\alpha]E$
$\mid E\, t \mid \mathbf{let}\, x = E\, \mathbf{in}\, e$	$\mid E\, t$
$\mid \mathbf{let}\, () = E\, \mathbf{in}\, e$	$\mid \mathbf{let}\, () = E\, \mathbf{in}\, e$
$\mid (E, t) \mid (v, E) \mid \mathbf{let}\, (x_1, x_2) = E\, \mathbf{in}\, e$	$\mid \mathbf{let}\, (x_1, x_2) = E\, \mathbf{in}\, e$
$\mid \mathbf{case}\, E\, \mathbf{of}\, (\mathbf{inj}_1\, x_1 \rightarrow e_1, \mathbf{inj}_2\, x_2 \rightarrow e_2)$	$\mid \mathbf{case}\, E\, \mathbf{of}\, (\mathbf{inj}_1\, x_1 \rightarrow e_1, \mathbf{inj}_2\, x_2 \rightarrow e_2)$
$\mid \mathbf{inj}_i\, E$	

Fig. 4. CBV and CBN for linear $\lambda\mu$: let reduction, values, and evaluation contexts

Terms reduce under arbitrary evaluation contexts, and under μ binders. Notice that evaluation contexts are closed under term variables x; we only consider reduction over closed terms. However, evaluation contexts are open under continuation variables α, and these contexts characterize the capturing of μ binders inside larger expressions.

$$\frac{t \rightarrow t'}{E[t] \rightarrow E[t']} \qquad \frac{c \rightarrow c'}{\mu\alpha.c \rightarrow \mu\alpha.c'} \qquad \frac{E[\mu\alpha.c] \text{ is a command}}{E[\mu\alpha.c] \rightarrow c\{E/\alpha\}}$$

As an example of the last rule, consider the application $(\mu\alpha.c)\,t$ which captures the current context $\Box\,t$ as the continuation α inside the command c. To achieve this, the continuation variable α should be replaced everywhere by the actual continuation $\Box\,t$. We write this substitution $c\{E/\alpha\}$, with the defining clause as

$$([\beta]t)\{E/\alpha\} = \begin{cases} E[t] & \text{if } \alpha = \beta \\ [\beta]t\{E/\alpha\} & \text{otherwise} \end{cases}$$

Note that in the first case, substitution does not continue inside of t because the use of continuation variables is linear.

Since this rule only applies to evaluation contexts that form a command, how do such contexts reduce when $E[\mu\alpha.c]$ is a term? First, the term undergoes η-expansion, and then μ-capturing applies to the new context $[\beta]E$:[3]

$$E[\mu\alpha.c] \to_\eta \mu\beta.[\beta](E[\mu\alpha.c]) \to \mu\beta.c\{[\beta]E/\alpha\}.$$

We define two classes of normal forms, t^N for terms and c^N for commands:

$$t^N ::= v \mid \mu\alpha.c^N \qquad\qquad c^N ::= [\beta]v$$

Theorem 2 (Progress and Preservation).

1. *If $\Gamma \vdash e : \Theta \mid \Pi$ then e is either one of t^N or c^N, or e can take a step.*
2. *If $\Gamma \vdash e : \Theta \mid \Pi$ and $e \to e'$, then $\Gamma \vdash e' : \Theta \mid \Pi$.*

4 Dualization and CP$^\pm$

On the surface, the programming style of CP seems alien to the programming style of $\lambda\mu$. To rephrase Bernardy et al. (2014), the $\lambda\mu$-calculus is still a λ-calculus; CP is not. To go even further, the type systems are not obviously equivalent; it is not clear how negation relates to the operators of CLL.

Tensor logic (Melliès and Tabareau 2010) provides insight into the relationship between a λ calculus and a one-sided judgment. Introduced in the setting of game theory, it also has applications in category theory (Melliès and Tabareau 2010) and as a type theory for CPS (Paykin et al. 2015). Perhaps the defining feature of tensor logic is that it can be studied equally well from two perspectives. The two-sided perspective results in a logic of non-involutive negation, while the one-sided perspective results in a logic of non-invertible polarization.

In this section we will replicate the two perspectives of tensor logic in the setting of classical linear logic. The question we must answer is: how does the two-sided $\lambda\mu$ calculus relate to the one-sided CP?

[3] The operational semantics we present here is quite a bit different from Selinger (2001) and Wadler (2005). Their $\mu-\beta$ rules are encompassed by our notion of μ-capturing. Their ξ rule corresponds to the procedure of μ-capturing inside a term described here.

4.1 Duality in $\lambda\mu$

De Morgan duality, written $(-)^{\perp}$, is the key to permuting types around the turnstile in a derivation. The canonical rules for duality are

$$\frac{\Gamma \vdash A, \Pi}{\Gamma, A^{\perp} \vdash \Pi} \qquad \frac{\Gamma, A \vdash \Pi}{\Gamma \vdash A^{\perp}, \Pi}$$

Duality is strictly involutive $((A^{\perp})^{\perp} = A)$, which means that the above rules are also invertible. This flexibility allows us to shift perspective by freely moving hypotheses to different parts of the judgment. Therefore, any judgment $\Gamma \vdash \Pi$ in a two-sided logic can be viewed in a one-sided logic as $\cdot \vdash \Gamma^{\perp}, \Pi$.

In classical logics, De Morgan duality is a meta-operation on types. We saw this operation in CLL, where $(X \otimes Y)^{\perp} = X^{\perp} \parr Y^{\perp}$, for example. But what is duality in the $\lambda\mu$ calculus, where $\parr$ is not a type operator? We could try to encode duality as negation $\neg A$, except that negation is not *strictly* involutive.

And yet, for any type A of linear $\lambda\mu$, we may imagine its *syntactic* dual $A^{\perp}$, not as a meta-operation, but as an explicit constructor. If we respect the fact that $A^{\perp\perp}$ is syntactically equal to A, we find that there are two disjoint classes of propositions: negative propositions A^{-}, which are syntactic duals, and positive propositions A^{+}, which correspond to the original types.[4]

We assign names to the duals of various connectives to simplify their presentation. In the style of CLL, we write $A^{\perp} \parr B^{\perp}$ for $(A \otimes B)^{\perp}$ where A and B are positive types. Thus each connective gets attributed both a positive and a negative copy, as summarized in Fig. 5. Seen another way, the positive and negative types are axiomatized as follows:

$$A^{+} ::= 1 \mid {A_1}^{+} \otimes {A_2}^{+} \mid 0 \mid {A_1}^{+} \oplus {A_2}^{+}$$
$$A^{-} ::= \perp \mid {A_1}^{-} \parr {A_2}^{-} \mid \top \mid {A_1}^{-} \;\&\; {A_2}^{-}$$

Operator	Positive Copy	Negative Copy
1	1	$\perp$
$\otimes$	$\otimes$	$\parr$
0	0	$\top$
$\oplus$	$\oplus$	$\&$
$\neg$	$\downarrow$	$\uparrow$

Fig. 5. Positive and negative copies of linear operators

[4] Drawing the connection to category theory, every object $A \in \mathcal{C}$ has a dual object $A^{\mathrm{op}} \in \mathcal{C}^{\mathrm{op}}$. Thus A and its dual live in distinct categories.

Negation. To incorporate negation into this analysis, first consider that it is self-dual: $(\neg A)^{\perp} = \neg A^{\perp}$. Close examination tells us that these are really two distinct copies of negation, one positive and one negative.

However, this presentation of negation is incompatible with the one-sided sequent calculus, because negation is *contravariant*: it moves a proposition from one side of the judgment to the other. In the presence of syntactic duality, we could imagine negation being partitioned into two parts: the contravariant duality operator $(-)^{\perp}$, along with a covariant shift operator written $\downarrow$, which turns a negative proposition into a positive proposition. The left and right negation rules from the two-sided formulation might instead be written as follows, where $\neg A^{+} = \downarrow(A^{+})^{\perp}$:

$$\dfrac{\dfrac{\Gamma \vdash A, \Pi}{\Gamma, A^{\perp} \vdash \Pi}}{\Gamma, \downarrow A^{\perp} \vdash \Pi} \qquad \dfrac{\dfrac{\Gamma, A \vdash \Pi}{\Gamma \vdash A^{\perp}, \Pi}}{\Gamma \vdash \downarrow A^{\perp}, \Pi}$$

The shift operator $\downarrow$, as well as its dual copy $\uparrow$, explicitly change the polarity of their argument. These fit nicely into Fig. 5, and they augment the syntax of types as follows:

$$\begin{aligned} A^{+} &::= \cdots \mid \downarrow A^{-} \\ A^{-} &::= \cdots \mid \uparrow A^{+} \end{aligned}$$

Thus the act of dualizing a logic with negation naturally generates polarized types, in the sense of Girard (1991). This relationship between negation and polarization is not new. It has been explored by Laurent and Regnier (2003) relating polarization procedures of classical logic to CPS translations using negation. Similarly, Laurent (2003) gave a semantics of the (non-linear) $\lambda\mu$-calculus in terms of polarized proof nets. Finally, Zeilberger (2008) takes negation to be a primitive operator in his polarized logic and observes that $\neg A^{+}$ is isomorphic to $\downarrow(A^{+})^{\perp}$.

4.2 Polarizing CP

It remains to see how polarization can be incorporated into the type system of CP. This question has already been addressed, at least in the case of intuitionistic session types, by Pfenning and Griffith (2015). In their setting, channels are directed, where positive channels are outgoing, and negative channels are incoming. Shift operations explicitly change the direction of the channel.

This intuition carries over naturally to the classical case. We define a term language CP$^{\pm}$, pronounced "CP more or less," to be a polarized version of CP. The syntax of processes is identical to that of CP, with the addition of two actions for shifting the direction of a channel. The syntax of processes is as follows:

$$P := \cdots \mid \downarrow x.P \mid \uparrow x.P$$

$$\frac{}{y \leftrightarrow x \vdash y : (A^+)^\perp; x : A^+}\ \text{AX} \qquad \frac{P_1 \vdash \Delta_1; \Pi_1, x : A \quad P_2 \vdash x : A^\perp, \Delta_2; \Pi_2}{\nu x.\langle P_1 \mid P_2 \rangle \vdash \Delta_1, \Delta_2; \Pi_1, \Pi_2}\ \text{CUT}$$

$$\frac{P \vdash \Delta, x : A^-; \Pi}{\downarrow x.P \vdash \Delta; x : \downarrow A^-, \Pi}\ \downarrow \qquad \frac{P \vdash \Delta; x : A^+, \Pi}{\uparrow x.P \vdash \Delta, x : \uparrow A^+; \Pi}\ \uparrow$$

$$\frac{}{x[].0 \vdash \cdot; x : 1}\ 1 \qquad \frac{P \vdash \Delta; \Pi}{x().P \vdash x : \bot, \Delta; \Pi}\ \bot$$

$$\frac{P_1 \vdash \Delta_1; \Pi_1, y : A^+ \quad P_2 \vdash \Delta_2; \Pi_2, x : B^+}{x[y].(P_1 \mid P_2) \vdash \Delta_1, \Delta_2; \Pi_1, \Pi_2, x : A^+ \otimes B^+}\ \otimes \qquad \frac{P \vdash y : A^-, x : B^-, \Delta; \Pi}{x(y).P \vdash x : A^- \parr B^-, \Delta; \Pi}\ \parr$$

(no 0 rule)

$$\frac{}{x.\mathbf{case}() \vdash x : \top, \Delta; \Pi}\ \top$$

$$\frac{P \vdash \Delta; \Pi, x : A_i{}^+}{x[\mathbf{inj}_i].P \vdash \Delta; \Pi, x : A_1{}^+ \oplus A_2{}^+}\ \oplus \qquad \frac{P_1 \vdash x : A^-, \Delta; \Pi \quad P_2 \vdash x : B^-, \Delta; \Pi}{x.\mathbf{case}(P_1 \mid P_2) \vdash x : A^- \,\&\, B^-, \Delta; \Pi}\ \&$$

Fig. 6. CP$^\pm$

The process $\downarrow x.P$ offers an output channel $x : \downarrow A^-$, which sends a special "shift" message over x, before reversing the direction of the channel and continuing with an input channel $x : A^-$. Similarly, $\uparrow x.Q$ waits to receive a shift message on x, before treating x as output in Q.

Judgments of CP$^\pm$ have the form $P \vdash \Delta; \Pi$, where Δ consists of negative (incoming) channels, and P_i consists of positive (outgoing) channels. The typing rules are given in Fig. 6.

The operational semantics of CP$^\pm$ augments the semantics of CP with rules for the shift operators. The β and η rules are as follows:

$$\nu x.\langle \downarrow x.P_1 \mid \uparrow x.P_2 \rangle \to_\beta \nu x.\langle P_2 \mid P_1 \rangle$$
$$(y \leftrightarrow x \vdash y : \uparrow A^+; x : \downarrow (A^+)^\perp) \to_\eta \downarrow x. \uparrow y. y \leftrightarrow x$$

The structural equivalences $\equiv_s$ in Eq. (1) are no longer applicable in CP$^\pm$. The reason is that only one of $\nu x.\langle P_1 \mid P_2 \rangle$ and $\nu x.\langle P_2 \mid P_1 \rangle$ is ever well-typed in CP$^\pm$, and similarly for $x \leftrightarrow y$ and $y \leftrightarrow x$.

Action contexts in CP$^\pm$ are extended with the shift operators.

$$C^a ::= \cdots \mid \downarrow x.\square \mid \uparrow x.\square.$$

4.3 Evaluation Order

Taking the commuting conversion rules verbatim from CP results in a similarly nondeterministic reduction strategy for CP$^\pm$. However, CP$^\pm$ has the ability to express other reduction strategies. In particular, we can specify either a call-by-name and call-by-value evaluation strategy in a way similar to Wadler's dual calculus (2003).

To illustrate what we mean by CBV and CBN in this context, consider the following encoding of implication as the polarized type $\Downarrow(A^+ \multimap B^-)$. Application can be encoded as follows:

$$\dfrac{P \vdash x : \Downarrow(A^+ \multimap B^-) \qquad \dfrac{\dfrac{Q \vdash y : A^+ \qquad \dfrac{}{z \leftrightarrow x \vdash z : B^-; x : (B^-)^{\perp}}}{x[y].(Q \mid z \leftrightarrow x) \vdash z : B^-; x : A^+ \otimes (B^-)^{\perp}}}{\uparrow x.x[y].(Q \mid z \leftrightarrow x) \vdash z : B^-, x : \Uparrow(A^+ \otimes (B^-)^{\perp}); \cdot}}{\nu x.\langle P \mid \uparrow x.x[y].(Q \mid z \leftrightarrow x)\rangle \vdash z : B^-; \cdot}$$

The process P is the equivalent of a λ-abstraction if it has the form $\downarrow x.x(y).P'$ for $P' \vdash x : B^-, y : (A^+)^{\perp}$. In this case, the application reduces to $\nu y.\langle Q \mid P'\{z/x\}\rangle$.

It is here we can talk about evaluation order. A call-by-value evaluation order reduces the argument Q before continuing in the evaluation of the result $z : B^-$. We can achieve this by prioritizing commuting conversions on the right of a cut over conversions on the left. This can be done by refining the definition of evaluation contexts.

$$C^e := \nu x.\langle P \mid \Box\rangle \mid \nu x.\langle \Box \mid P^x\rangle$$

where P^x is any process starting with an action on x.

Similarly, a call-by-name evaluation order tries to compute $z : B^-$ before evaluating the argument Q, by prioritizing conversions on the left:

$$C^e := \nu x.\langle \Box \mid P\rangle \mid \nu x.\langle P^x \mid \Box\rangle.$$

5 Equational Correspondence

The dualization described in the previous section shows how typing judgments in the linear $\lambda\mu$-calculus relate to typing judgments in a CP-like language. But the structure of the typing judgments says nothing about the equivalence of the operational semantics of the two languages. The notion of *equational correspondence* (Sabry and Felleisen 1993) makes this relationship formal.

Definition 3. *Let L_1 and L_2 be term languages with equality, written $t_1 =_1 t_1'$ and $t_2 =_2 t_2'$. Suppose that there exist two maps F_1 and F_2 as shown below. These maps form an* equational correspondence *if the following conditions hold:*

$$\begin{tikzcd} L_1 \arrow[r, bend left, "F_1"] & L_2 \arrow[l, bend left, "F_2"] \end{tikzcd}$$

1. *If $t_1 =_1 t_1'$ then $F_1(t_1) =_2 F_1(t_1')$.*
2. *If $t_2 =_2 t_2'$ then $F_2(t_2) =_1 F_2(t_2')$.*
3. *For all terms $t_1 \in L_1$, $t_1 =_1 F_2(F_1(t_1))$.*
4. *For all terms $t_2 \in L_2$, $t_2 =_2 F_1(F_2(t_2))$.*

We thus come to our main theorem:

Theorem 4. *Linear $\lambda\mu$ equationally corresponds with* CP$^{\pm}$.

$$\frac{\Gamma \vdash t : A \mid \Pi}{t^{(x)} \vdash \Gamma^{\perp}; \Pi, x : A} \qquad \frac{\Gamma \vdash c \mid \Pi}{c^{*} \vdash \Gamma^{\perp}; \Pi} \qquad\qquad \frac{P \vdash \Delta; \Pi}{\Delta^{\perp} \vdash (P)_{*} \mid \Pi}$$

$$\begin{aligned}
y^{(x)} &:= y \leftrightarrow x \\
(\mathbf{let}\, y = t \,\mathbf{in}\, e)^{(x)} &:= \nu y.\langle t^{(y)} \mid e^{(x)} \rangle \\
(\mu\alpha.c)^{(x)} &:= (c^{*})\{x/\alpha\} \\
([\alpha]t)^{*} &:= (t^{(x)})\{\alpha/x\} \\
(\overline{\lambda}x.c)^{(x)} &:= \downarrow x.c^{*} \\
(t_1\, t_2)^{*} &:= \nu x.\langle {t_1}^{(x)} \mid \uparrow x.{t_2}^{(x)} \rangle \\
()^{(x)} &:= x[].0 \\
(\mathbf{let}\, () = t \,\mathbf{in}\, e)^{(x)} &:= \nu y.\langle t^{(y)} \mid y().e^{(x)} \rangle \\
(t_1, t_2)^{(x)} &:= x[x'].({t_1}^{(x')} \mid {t_2}^{(x)}) \\
(\mathbf{let}\, (y', y) = t \,\mathbf{in}\, e)^{(x)} &:= \nu y.\langle t^{(y)} \mid y(y').e^{(x)} \rangle \\
(\mathbf{case}\, t \,\mathbf{of}\, ())^{(x)} &:= \nu y.\langle t^{(y)} \mid y.\mathbf{case}() \rangle \\
(\mathbf{inj}_i\, t)^{(x)} &:= x[\mathbf{inj}_i].t^{(x)} \\
(\mathbf{case}\, t \,\mathbf{of}\, (\mathbf{inj}_1\, y \to e_1, \mathbf{inj}_2\, y \to e_2))^{(x)} &:= \\
\nu y.\langle t^{(y)} \mid y.\mathbf{case}({e_1}^{(x)} \mid {e_2}^{(x)}) \rangle &
\end{aligned}$$

$$\begin{aligned}
(y \leftrightarrow x)_{*} &:= [x]y \\
(\nu y.\langle P_1 \mid P_2 \rangle)_{*} &:= \mathbf{let}\, y = (P_1)_{*} \,\mathbf{in}\, (P_2)_{*} \\
(\downarrow x.P)_{*} &:= [x]\overline{\lambda}x.(P)_{*} \\
(\uparrow y.P)_{*} &:= y\,((P)_{(y)}) \\
(x[].0)_{*} &:= [x]() \\
(y().P)_{*} &:= \mathbf{let}\, () = y \,\mathbf{in}\, (P)_{*} \\
(x[y].(P_1 \mid P_2))_{*} &:= [x]((P_1)_{*}, (P_2)_{*}) \\
(y(y').P)_{*} &:= \mathbf{let}\, (y', y) = y \,\mathbf{in}\, (P)_{*} \\
(y.\mathbf{case}())_{*} &:= \mathbf{case}\, y \,\mathbf{of}\, () \\
(x[\mathbf{inj}_i].P)_{*} &:= [x]\mathbf{inj}_i\, (P)_{*} \\
(y.\mathbf{case}(P_1 \mid P_2))_{*} &:= \\
\mathbf{case}\, y \,\mathbf{of}\, (\mathbf{inj}_1\, y \to (P_1)_{*}, \mathbf{inj}_2\, y \to (P_2)_{*}) &
\end{aligned}$$

Fig. 7. Equational correspondence between linear $\lambda\mu$ and CP$^{\pm}$

Proof (Sketch). We start by defining translations, shown in Fig. 7, between the two languages. First, for $\Gamma \vdash e : \Theta \mid \Pi$, we define $e^{(x)}$ to be a process so that $e^{(x)} \vdash \Gamma^{\perp}; \Pi, x : \Theta$. When e is a command c we write c^{*}.

Next, for $P \vdash \Delta; \Pi$, we define $(P)_{*}$ to be a command so that $\Delta^{\perp} \vdash (P)_{*} \mid \Pi$. The classes of term and continuation variables are collapsed in CP$^{\pm}$, so we move between them freely in the translations.

The remainder of the proof must show that the four conditions in the definition of equational correspondence hold. But condition (4), which says that all $\lambda\mu$ expressions e satisfy $e = (e^{(\alpha)})_{*}$, is not quite right for terms, because every $(P)_{*}$ is a command. Instead we amend the condition to say that for every term t, we have $t = \mu\alpha.(t^{(\alpha)})_{*}$. $\square$

6 Discussion

6.1 Comparing Classical Type Systems

Unlike the simply-typed λ-calculus for intuitionistic logic, there is no canonical type system for classical logic. Instead, there are a number of candidate type systems corresponding to different well-known formulations of classical logic.

- Classical logic can be formulated as intuitionistic logic plus the property of double negation elimination. The $\lambda_{\mathcal{C}}$-calculus (Griffin 1990) is a typed λ-calculus with the addition of the control operator $\mathcal{C}$, of type $((A \to \bot) \to \bot) \to A$.

- Classical logic can alternatively be formulated as intuitionistic logic with multiple conclusions, which corresponds to the structure of the $\lambda\mu$-calculus (Parigot 1992).
- The logic may be in natural deduction or sequent calculus form. Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus (2000), which is also called System L (Spiwack 2014), and Wadler's dual calculus (2003) are variations on equivalent sequent calculus formulations of $\lambda\mu$.

These systems are all related by equational correspondences—de Groote (1994) proves the correspondence between $\lambda\mu$ and $\lambda_{\mathcal{C}}$, and Wadler (2005) proves the correspondence between $\lambda\mu$ and the dual calculus. Our work continues in this tradition by indicating that the linear versions of these systems (for example, Spiwack's Linear L 2014) are also equivalent to CP$^{\pm}$.

6.2 A Syntax for Tensor Logic

Melliès and Tabareau introduce tensor logic in the setting of game-theoretic semantics for linear logic. While they study the denotational semantics of the logic extensively and provide translations from CLL, Melliès and Tabareau do not give proof terms for either of the two formulations—two-sided or one-sided.

However, proof terms are not hard to imagine in the style of $\lambda\mu$ and CP$^{\pm}$. The two-sided perspective types a (strict) sublanguage of linear $\lambda\mu$ with negation but without the μ operator. The result is a simply-typed λ-calculus where the return type of every function is $\perp$, as described by Lafont et al. (1993). The one-sided perspective of tensor logic can similarly type a (strict) sublanguage of CP$^{\pm}$, where each process offers at most one output channel. This formulation is equivalent to Spiwack's Polarized L (2014), and corresponds to a *focused* polarized logic, as opposed to CP$^{\pm}$, which is unfocused.

6.3 Session Types and Linear Logic

Linear logic has long been explored as a type system for concurrent programming, especially process calculi in the style of the π-calculus (Bellin and Scott 1994; Beffara 2006; Kobayashi et al. 1999). Caires and Pfenning (2010) present an elegant interpretation of an (intuitionistic) linearly-typed fragment of the π-calculus by interpreting linear propositions as session types (Honda 1993; Honda et al. 1998) in a calculus they call π-DILL. The connections between session types and linear logic have been studied in the past (Mazurak and Zdancewic 2010; Gay and Vasconcelos 2010) but had not been formally compared. Wadler (2014) makes the connection formal by exhibiting a translation between CP, a classical variant of π-DILL, and GV, a variation of the session-typed calculus of Gay and Vasconcelos (2010). Lindley and Morris (2014, 2015) expand this translation to a bisimulation between CP and GV.

6.4 Variations on CP

The process calculus described in Sect. 2 varies from Wadler's original formulation of CP in a few significant ways. The first is that Wadler's processes only communicate at the top level, never under prefix actions. This means that normal forms consist of any processes that do not begin with a cut. Following Pérez et al. (2014), we allow communication to take place at any point in a process, even if another channel is waiting to synchronize. For example, in the process

$$x().\nu y.\langle y[].0 \mid y().P\rangle,$$

communication takes place over the channel y even though x sends a message first. Operationally, the difference is the exclusion of the structural rule in Eq. (2). Wadler's equational theory is thus finer-grained than ours, and does not reach the level of behavioral equivalence.

In the same vein, Wadler does not equate the processes defined by our third class of commuting conversions. Following Bellin and Scott (1994) and Pérez et al. (2014), behavioral equivalence means that the order of non-interfering prefix actions is unobservable, and should be equated.

The treatment of η-expansions for link forwarding is another departure from Wadler's presentation. Like Wadler, we allow link forwarding at arbitrary type for the sake of expressibility. However, just as in the λ-calculus, η-expansion reduces the allowable normal forms by equating behaviorally equivalent processes.

The operational semantics we define for CP in Sect. 2 extends naturally to CP$^{\pm}$. It is worth noting, however, that the equational correspondence proved in Sect. 5 relies on the second class of commuting conversions (CUT-CUT) but not the third (ACTION-ACTION). Therefore it would still be applicable using a semantics closer to Wadler's original presentation (2014).

Finally, Wadler's presentation of CP includes the linear exponentials ! and ?, as well as polymorphism $\forall$ and $\exists$. In this paper we work with the linear "core" of CP, excluding the exponentials for the sake of clarity. To incorporate polymorphism into our presentation, we would need to extend types with type variables. In that case, communication would eliminate all cuts, but not necessarily links at atomic type.

6.5 How Much is "More or Less"?

In the introduction we made the claim that linear $\lambda\mu$ was "more or less" CP, and we made the statement precise by exhibiting CP$^{\pm}$. However, we haven't addressed exactly how much CP$^{\pm}$ is "more or less" CP.

We can easily define an erasure operation $|-|$ on both types and processes that removes all shift operations. Trivially, if $P \vdash \Delta; \Pi$ then $|P| \vdash |\Delta|, |\Pi|$. As Zeilberger (2008) shows for a different polarized term syntax, the completeness of such an erasure can be summarized as follows: If $Q \vdash |\Delta|, |\Pi|$ in CP, then there exists a CP$^{\pm}$ process P such that $P \vdash \Delta; \Pi$ and $|P| = Q$. In addition, we would require that this "expansion" of Q respects equalities in a similar way to equational correspondences. However, even describing the necessary relationship in detail is beyond the scope of this work.

Meanwhile, CP$^\pm$ offers operational behaviors not expressible in CP. In particular, CP$^\pm$ has the ability to set a particular reduction strategy such as CBV, CBN, or something in between. Once an evaluation strategy is fixed, the types dictate the evaluation of any particular process. The programmer thus has full control over scheduling in CP$^\pm$, while in CP the programmer has no control.

Acknowledgments. Thanks to Neel Krishnaswami, Sam Lindley, and the paper's reviewers for their valuable insights. We would also like to thank Phil Wadler for inspiring a fascination with the Curry-Howard correspondence. This material is based in part upon work supported by the NSF Graduate Research Fellowship under Grant No. DGE-1321851 and by NSF Grant No. CCF-1421193.

References

Beffara, E.: A concurrent model for linear logic. Electron. Notes Theor. Comput. Sci. **155**, 147–168 (2006)

Bellin, G., Scott, P.: On the π-calculus and linear logic. Theor. Comput. Sci. **135**(1), 11–65 (1994)

Bernardy, J.P., Rosn, D., Smallbone, N.: Compiling linear logic using continuations (2014)

Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)

Curien, P.L., Herbelin, H.: The duality of computation. ACM SIGPLAN Not. **35**(9), 233–243 (2000)

Felleisen, M.: The theory and practice of first-class prompts. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 180–190. ACM, New York (1988)

Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**, 19–50 (2010)

Girard, J.Y.: Linear logic. Theor. Comput. Sci. **50**(1), 1–101 (1987)

Girard, J.Y.: A new constructive logic: classic logic. Math. Struct. Comput. Sci. **1**, 255–296 (1991)

Griffin, T.G.: A formulae-as-type notion of control. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, pp. 47–58. ACM, New York (1990)

de Groote, P.: On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In: Pfenning, F. (ed.) Logic Programming and Automated Reasoning. Lecture Notes in Computer Science, vol. 822, pp. 31–43. Springer, Heidelberg (1994)

Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)

Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the π-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999)

Lafont, Y., Reus, B., Streicher, T.: Continuation semantics or expressing implication by negation. Technical report 9321, Ludwig-Maximilians-Universitt (1993)

Laurent, O.: Polarized proof-nets and $\lambda\mu$-calculus. Theor. Comput. Sci. **290**(1), 161–188 (2003)

Laurent, O., Regnier, L.: About translations of classical logic into polarized linear logic. In: Logic in Computer Science, pp. 11–20. IEEE (2003)

Lindley, S., Morris, J.G.: Sessions as propositions. Electron. Proc. Theor. Comput. Sci. **155**, 9–16 (2014)

Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 560–584. Springer, Heidelberg (2015)

Mazurak, K., Zdancewic, S.: Lolliproc: To concurrency from classical linear logic via Curry-Howard and control. SIGPLAN Not. **45**(9), 39–50 (2010)

Melliès, P.A., Tabareau, N.: Resource modalities in tensor logic. Ann. Pure Appl. Logic **161**(5), 632–653 (2010). The Third Workshop on Games for Logic and Programming Languages (GaLoP), 5–6 April 2008

Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**, 119–141 (1992)

Munch-Maccagnoni, G.: Focalisation and classical realisability. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 409–423. Springer, Heidelberg (2009)

Parigot, M.: $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) Logic Programming and Automated Reasoning. Lecture Notes in Computer Science, vol. 624, pp. 190–201. Springer, Heidelberg (1992)

Paykin, J., Zdancewic, S., Krishnaswami, N.R.: Linear temporal type theory for event-based reactive programming (2015)

Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Inf. Comput. **239**, 254–302 (2014)

Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) FOSSACS 2015. LNCS, vol. 9034, pp. 3–22. Springer, Heidelberg (2015)

Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LISP Symbolic Comput. **6**(3–4), 289–360 (1993)

Selinger, P.: Control categories and duality: On the categorical semantics of the lambda-mu calculus. Math. Struct. Comput. Sci. **11**, 207–260 (2001). http://journals.cambridge.org/article_S096012950000311X

Spiwack, A.: A dissection of L (2014)

Wadler, P.: Call-by-value is dual to call-by-name. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, pp. 189–201. ACM, New York (2003)

Wadler, P.: Call-by-value is dual to call-by-name reloaded. In: Giesl, J. (ed.) Term Rewriting and Applications. LNCS, vol. 3467, pp. 185–203. Springer, Heidelberg (2005)

Wadler, P.: Propositions as sessions. SIGPLAN Not. **47**(9), 273–286 (2012)

Wadler, P.: Propositions as sessions. J. Funct. Program. **24**, 384–418 (2014)

Zeilberger, N.: On the unity of duality. Ann. Pure Appl. Logic **153**(13), 66–96 (2008). Special Issue: Classical Logic and Computation (2006)

A Reflection on Types

Simon Peyton Jones[1(✉)], Stephanie Weirich[2], Richard A. Eisenberg[2], and Dimitrios Vytiniotis[1]

[1] Microsoft Research, Cambridge, UK
simonpj@microsoft.com
[2] Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA
sweirich@cis.upenn.edu

Abstract. The ability to perform type tests at runtime blurs the line between statically-typed and dynamically-checked languages. Recent developments in Haskell's type system allow even programs that use reflection to themselves be statically typed, using a type-indexed runtime representation of types called *TypeRep*. As a result we can build dynamic types as an ordinary, statically-typed library, on top of *TypeRep* in an open-world context.

1 Preface

If there is one topic that has been a consistent theme of Phil Wadler's research career, it would have to be *types*. Types are the heart of the Curry-Howard isomorphism, occupying the intersection of *logic* and *practical programming*. Phil has always been fascinated by this remarkable dual role, and many of his papers explore that idea in more detail.

One of his most seminal ideas was that of *type classes*, which (with his student Steve Blott) he proposed, fully-formed, to the Haskell committee in February 1988 [WB89]. At that time we were wrestling with the apparent compromises necessary to support equality, numerics, serialisation, and similar functions that have type-specific, rather than type-parametric, behaviour. Type classes completely solved that collection of issues, and we enthusiastically adopted them for Haskell [HHPJW07]. What we did not know at the time is that, far from being a niche solution, type classes would turn out to be a seed bed from which would spring all manner of remarkable fruit: before long we had multi-parameter type classes; functional dependencies; type classes over type constructors (notably the *Monad* and *Functor* classes, more recently joined by a menagerie of *Foldable*, *Traversable*, *Applicative* and many more); implicit parameters, derivable classes, and more besides.

One particular class that we did not anticipate, although it made an early appearance in 1990, was *Typeable*. The *Typeable* class gives Haskell a handle on *reflection*: the ability to examine types at runtime and take action based on those tests. Many languages support reflection in some form, but Haskell is moving steadily towards an unusually statically-typed form of reflection, contradictory though that sounds, since reflection is all about dynamic type tests. That topic is the subject of this paper, a reflection on types in homage to Phil.

2 Introduction

Static types are the world's most successful formal method. They allow programmers to specify properties of functions that are proved on every compilation. They provide a design language that programmers can use to express much of the architecture of their programs before they write a single line of algorithmic code. Moreover this design language is not divorced from the code but part of it, so it cannot be out of date. Types dramatically ease refactoring and maintenance of old code bases.

Type systems should let you say what you mean. Weak type systems get in the way, which in turn give types a bad name. For example, no one wants to write a function to reverse a list of integers, and then duplicate the code to reverse a list of characters: we need polymorphism! This pattern occurs again and again, and is the motivating force behind languages that support sophisticated type systems, of which Haskell is a leading example.

And yet, there comes a point in every language at which the static type system simply cannot say what you want. As Leroy and Mauny put it *"there are programming situations that seem to require dynamic typing in an essential way"* [LM91]. How can we introduce dynamic typing into a statically typed language without throwing the baby out with the bathwater? In this paper we describe how to do so in Haskell, making the following contributions:

- We motivate the need for dynamic typing (Sect. 3), and why it needs to work in an *open* world of types (Sect. 4). Supporting an open world is a real challenge, which we tackle head on in this paper. Many other approaches are implicitly closed-world, as Sect. 9 discusses.
- Dynamic typing requires a runtime test of type equality, so some structure that represents a type—a *type representation*—must exist at runtime, along with a way to get the type representation for a value. We describe a *type-indexed* form of type representation, *TypeRep a* (Sects. 5.1 and 5.2), and explain how to use it for a type-safe dynamic type test (Sect. 5.3).
- We show that simply *comparing* type representations is not enough; in some applications we must also safely *decompose* them (Sect. 5.4).
- Rather unexpectedly, it turns out that supporting decomposition for type representations requires GADT-style *kind equalities*, a feature that has only just been added to GHC 8.0 (Sect. 5.5). Type-safe reflection requires a very sophisticated type system indeed!

Our key result is a way to build open-world dynamic typing as an ordinary statically-typed library (i.e. not as part of the trusted code base), using a very small (trusted) reflection API for *TypeRep*. We also briefly discuss our implementation (Sect. 6), metatheory (Sect. 7), and other applications (Sect. 8), before concluding with a review of related work (Sect. 9). This paper is literate Haskell and our examples compile under GHC 8.0.

3 Dynamic Types in a Statically Typed Language

Haskell's type system is so expressive that it is remarkably hard to find a compelling application for dynamic typing. But here is one. Suppose you want to write a Haskell library to implement the following familiar state-monad API[1,2]:

```
data ST s a        -- Abstract type for state monad
data STRef s a     -- Abstract type for references (to value of type a)
runST       :: (∀ s. ST s a) → a
newSTRef    :: a → ST s (STRef s a)
readSTRef   :: STRef s a → ST s a
writeSTRef  :: STRef s a → a → ST s ()
```

Papers about state monads usually assume that the implementation is built in, but what if it were not? This is not a theoretical question: actively-used Haskell libraries, such as *vault*[3] face exactly this challenge. To implement *ST* we need some kind of "store" that maps a key (a *STRef*) to its value. This *Store* should have the following API (ignore the *Typeable* constraints for now):

```
extendStore :: Typeable a ⇒ STRef s a → a → Store → Store
lookupStore :: Typeable a ⇒ STRef s a → Store → Maybe a
```

It makes sense to implement the *Store* by a finite map, keyed by *Int* or some other unique key, which itself is kept inside the *STRef*. For that purpose, we can use the standard Haskell type *Map k v*, mapping keys *k* to values *v*. But what type should *v* be? As the type of *extendStore* declares, we must be able to insert *any* type of value into the *Store*. This is where type *Dynamic* is useful:

```
type Key = Int
data STRef s a = STR Key
type Store = Map Key Dynamic
```

Dynamic suffices if we have the following operations available, used to create and take apart *Dynamic* values:[4]

```
toDynamic   :: Typeable a ⇒ a → Dynamic
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a
```

[1] There is another connection with Phil's work here: an API like this was first proposed in *"Imperative functional programming"* [PJW93], a collaboration between one of the present authors and Phil, directly inspired by Phil's ground-breaking paper *"Comprehending monads"* [Wad90].

[2] For our present purposes you can safely ignore the "*s*" type parameter; the paper *"State in Haskell"* explains what is going on [LPJ95].

[3] https://hackage.haskell.org/package/vault.

[4] These types also motivate the *Typeable* constraint above. We discuss that constraint further in Sect. 5.2, but without looking that far ahead, Phil's insight about "theorems for free" tells us that the type *fromDynamic* :: *Dynamic* → *Maybe a* is a non-starter [Wad89]. Any function with that type must return *Nothing*, *Just* $\bot$, or diverge.

We can now implement the functions on *Store*, thus:

```
extendStore (STR k) v s = insert k (toDynamic v) s
lookupStore (STR k)   s = case lookup k s of
                              Just d   → fromDynamic d
                              Nothing → Nothing
```

In *lookupStore* the dynamic type check made by *fromDynamic* will always succeed. (That is, when looking up a *STRef s a*, we should always find a value of type *a*.) The runtime tests compensate where the static type system is inadequate.

In summary, there are a few occasions when even a type system as sophisticated as Haskell's is not powerful enough to give the static guarantees we seek. A *Dynamic* type, equipped with *toDynamic* and *fromDynamic*, can plug the gap.

4 The Challenge of an Open World

Where does type *Dynamic* come from? One classic approach is to make *Dynamic* a tagged union of all the types we care about, like this:

```
data Dynamic = DInt Int
             | DBool Bool
             | DChar Char
             | DPair Dynamic Dynamic
               ...

toDynInt :: Int → Dynamic
toDynInt = DInt

fromDynInt :: Dynamic → Maybe Int
fromDynInt (DInt n) = Just n
fromDynInt _        = Nothing

toDynPair :: Dynamic → Dynamic → Dynamic
toDynPair = DPair

dynFst :: Dynamic → Maybe Dynamic
dynFst (DPair x1 x2) = Just x1
dynFst _             = Nothing
```

For each type constructor (*Int*, *Bool*, pairs, lists, etc.) we define a data constructor (e.g. *DPair*) in the *Dynamic* type, plus a constructor function (e.g. *toDynPair*) and one or more destructors (e.g. *dynFst*, *dynSnd*).

This approach has a fundamental shortcoming: it is not extensible. Concretely, what is the "..." above? The data type declaration for *Dynamic* can enumerate only a fixed set of type constructors (integers, booleans, pairs, etc.)[5]. We call this the *closed-world assumption*. Sometimes a closed world is acceptable. For example, if we were writing an evaluator for a small language we would

[5] Although the set of type *constructors* is fixed, you can use them to build an infinite number of *types*; e.g. (*Int*, *Bool*), (*Int*, (*Bool*, *Int*)), etc.

need *Dynamic* to have only enough data constructors to represent the types of the object language.

But in general the world is simply not closed; it is unreasonable to extend *Dynamic*, whenever the user defines a new data type! In the *ST* example, it is fundamental that the monad be able to store values of user-declared types.

So in this paper we focus exclusively on the challenge of open-world extensibility. Before moving on, it is worth noting that a surprisingly large fraction of the academic literature on dynamics in a statically typed language makes a closed-world assumption (see Sect. 9). Moreover, even if we accept a closed world, the approach sketched above has other difficulties, discussed in Sect. 9.2.

5 TypeRep: Runtime Reflection in an Open World

We can implement an open-world *Dynamic* as an ordinary, type-safe Haskell library, on top of a new abstraction, that of *type-indexed type representations* or *TypeRep*. In fact, Haskell has supported (un-type-indexed) type representations and an open-world *Dynamic* for years, but in a rather unsatisfactory way (the "old design"). However, recent developments in Haskell's type system—notably GADTs [XCC03,PJVWW06], kind polymorphism [YWC+12], and kind equalities [WHE13]—have opened up new opportunities (the "new design"). A major purpose of this paper is to motivate and describe this new design. For readers familiar with the old design, we compare it with the new one in Sect. 9.1.

5.1 Introducing TypeRep

The key to our approach is our type-indexed type representation *TypeRep*. But what is a type-indexed type representation? It is best understood by example:

- The representation of *Int* is the value of type *TypeRep Int*.
- The representation of *Bool* is the value of type *TypeRep Bool*.
- And so on.

That is, the index in a type-indexed type representation is itself the represented type. *TypeRep* is abstract, and thus we don't write the *TypeRep* value in the examples above. Note, however, that we have said *the* value, not *a* value—there is precisely one value of type *TypeRep Int*[6]. *TypeRep* thus defines a family of singleton types [EW12]; indeed, *TypeRep* is the singleton family associated with the kind $\star$ of types.

As we build out the API for *TypeRep*, we will consider how to build an efficient, type-safe, and open-world implementation of *Dynamic*. Converting to and from *Dynamic* should not touch the value itself; instead we represent a dynamic value as a pair of a value and a runtime-inspectable representation of its type. Thus:

[6] Recall that $\perp$ is not a value.

```
data Dynamic where
  Dyn :: TypeRep a → a → Dynamic
```

Here we are using GADT-style syntax to declare the constructor *Dyn*, whose payload includes a runtime representation of a type *a* and a value of type *a*. The type *a* is existentially bound; that is, it does not appear in the result type of the data constructor.

Now we have two challenges: where do we get the *TypeRep* from when creating a *Dynamic* in *toDynamic* (Sect. 5.2); and what do we do with it when unpacking it in *fromDynamic* (Sect. 5.3)?

5.2 The Typeable Class

Because each type has its own *TypeRep*, the obvious approach is to use a type class, thus:

```
class Typeable a where
  typeRep :: TypeRep a
```

This class has only one operation, a nullary function (or simple value) that is the type representation for the type. Now we can write *toDynamic*:

```
toDynamic :: Typeable a ⇒ a → Dynamic
toDynamic x = Dyn typeRep x
```

The type of the data constructor *Dyn* ensures that the call of *typeRep* produces a type representation for the type *a*. Easy!

But where do *instances* of *Typeable* come from? The magic of type classes gives us a simple way to solve the open-world challenge, by using a single piece of built-in compiler support: every data type declaration gives rise to a *Typeable* instance for that type (Sect. 6). Furthermore, because *Typeable* and its instances are built in, we can be sure that these representations uniquely define types; for example, the user cannot write bogus instances of *Typeable* that use the same *TypeRep* for two different types.

5.3 Type-Aware Equality for TypeReps

The second challenge is to unpack dynamics. We need a function with this signature:

```
fromDynamic :: Typeable a ⇒ Dynamic → Maybe a
```

The function *fromDynamic* takes a *Dynamic* and tests whether it wraps a value of the desired type; if so, it returns the value wrapped in *Just*; if not, it returns *Nothing*. The *Typeable* constraint allows *fromDynamic* to know what the "desired type" is.

But how is *fromDynamic* implemented? Patently it must compare type representations, so we might try this:

```
fromDynamic :: ∀ d. Typeable d ⇒ Dynamic → Maybe d
fromDynamic (Dyn (ra :: TypeRep a) (x :: a))
   | rd == ra  = Just x   -- Eeek! Type error!
   | otherwise = Nothing
  where
    rd = typeRep :: TypeRep d
```

The type signatures for *ra* and *x* could be omitted, but we have put them in to remind ourselves that the types of *ra* and *x* are connected through the existentially-bound type *a*. The value *rd* is (the runtime representation of) the "desired type", disambiguated by a type signature. We compare *rd* with *ra*, (the representation of) *x*'s type, and return *Just x* if they match. The operational behaviour is just what we want, but the type checker will reject it. It has no reason to believe that *x* actually has type *d*: the type-checker surely does not understand that we have just compared *TypeRep*s linking up *x*'s type with *d*.

Fortunately, we have a fine tool to use whenever a runtime operation needs to inform us about types: generalised algebraic data types, or GADTs. We need an equality on *TypeRep* that returns a GADT; pattern-matching on the return value gives the type-checker just the information it needs. Here are the definitions[7]:

```
eqT :: TypeRep a → TypeRep b → Maybe (a :≈: b)
    -- Primitive; implemented by compiler
data a :≈: b where
  Refl :: a :≈: a
```

Here *eqT* returns *Nothing* if the two *TypeRep*s are different, and (*Just Refl*) if they are the same. The data constructor *Refl* is the sole, nullary data constructor of the GADT (*a* :≈: *b*). Pattern matching on *Refl* tells the type checker that the two types are the same. In short, when the argument types are equal, *eqT* returns a proof of this equality, in a form that the type checker can use.

To be concrete, here is the definition of *fromDynamic*:

```
fromDynamic :: ∀ d. Typeable d ⇒ Dynamic → Maybe d
fromDynamic (Dyn (ra :: TypeRep a) (x :: a))
   = case eqT ra (typeRep :: TypeRep d) of
       Nothing   → Nothing
       Just Refl → Just x
```

We use *eqT* to compare the two *TypeRep*s, and pattern-match on *Refl*, so that in the second case alternative we know that *a* and *d* are equal, so we can return *Just x* where a value of type *Maybe d* is needed.

Since *Maybe* is a monad, we can use **do** notation for this code, and instead write it like this:

[7] Here we are using GHC's ability to define infix type constructors.

```
fromDynamic (Dyn ra x)
  = do Refl ← eqT ra (typeRep :: TypeRep d)
       return x
```

When we make multiple matches this style is more convenient, so we use it from now on.

More generally, *eqT* allows to implement *type-safe* cast, a useful operation in its own right [Wei04,LPJ03,LPJ05].

```
cast :: ∀ a b. (Typeable a, Typeable b) ⇒ a → Maybe b
cast x = do Refl ← eqT (typeRep :: TypeRep a)
                       (typeRep :: TypeRep b)
            return x
```

5.4 Decomposing Type Representations

So far, the only operation we have provided over *TypeRep* is *eqT*, which compares two type representations for equality. But that is not enough to implement *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn pab x)
 =   -- Check that pab represents a pair type
     -- Take (fst x) and wrap it in Dyn
```

How can we decompose the type representation *pab*, to check that it indeed represents a pair, and extract its first component? Since types in Haskell are built via a sequence of type applications (much like how an expression applying a function to multiple arguments is built with several nested term applications), the natural dual is to provide a way to decompose type applications:

```
splitApp :: TypeRep a → Maybe (AppResult a)
    -- Primitive; implemented by compiler
data AppResult t where
  App :: TypeRep a → TypeRep b → AppResult (a b)
```

The primitive operation *splitApp* allows us to observe the structure of types. If *splitApp* is applied to a type constructor, such as *Int*, it returns *Nothing*; otherwise, for a type application, it decomposes one layer of the application, and returns (*Just* (*App ra rb*)) where *ra* and *rb* are representations of the sub-components. Like *eqT*, it returns a GADT, *AppResult*, to expose the type equalities it has discovered to the type checker.

Now we can implement *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn rpab x)
  = do App rpa rb ← splitApp rpab
```

```
        App rp  ra ← splitApp rpa
        Refl       ← eqT rp (typeRep :: TypeRep (,))
        return (Dyn ra (fst x))
```

We check that the type of *x*, whose *TypeRep*, *rpab*, is of form (,) *a b*, by decomposing it twice with *splitApp*. Then we must check that *rp*, the *TypeRep* of the function part of this application, is indeed the pair type constructor (,); we can do that using *eqT*. These three GADT pattern matches combine to tell the type checker that the type of *x*, which began life in the (*Dyn rpab x*) pattern match as an existentially-quantified type variable, is indeed a pair type (*a*, *b*). So we can safely apply *fst* to *x*, to get a result whose type representation *ra* we have in hand.

In the same way we can use *splitApp* to implement *dynApply*, which applies a function *Dynamic* to an argument *Dynamic*, provided the types line up:

```
dynApply :: Dynamic → Dynamic → Maybe Dynamic
dynApply (Dyn rf f) (Dyn rx x) = do
  App ra  rt2 ← splitApp rf
  App rtc rt1 ← splitApp ra
  Refl        ← eqT rtc (typeRep :: TypeRep (→))
  Refl        ← eqT rt1 rx
  return (Dyn rt2 (f x))
```

In both cases, the code is simple enough, but the type checker has to work remarkably hard behind the scenes to prove that it is sound. Let us take a closer look.

5.5 Kind Polymorphism and Kind Equalities

There is something suspicious about our use of *typeRep* :: *TypeRep* (,). So far we have discused type representations for only types of kind $\star$. But (,) has kind ($\star \to \star \to \star$); does it too have a *TypeRep*? Of course it must, to allow *TypeRep Int*, *TypeRep Maybe*, and *TypeRep* (,). So the type constructor *TypeRep* must be polymorphic in the kind of its type argument, or *poly-kinded*, and so must be its accompanying class *Typeable*, thus:

```
data TypeRep (a :: k)   -- primitive, indexed by type and kind
class Typeable (a :: k) where
  typeRep :: TypeRep a
```

Fortunately, GHC has offered kind polymorphism for some years [YWC+12]. Similarly, the result GADT *AppResult* must be kind-polymorphic. Here is its definition with kind signatures added[8]:

[8] The kind signatures are optional. With *PolyKinds* enabled, GHC infers them, but we often add them for clarity.

```
data AppResult (t :: k) where
  App :: ∀ k1 k (a :: k1 → k) (b :: k1).
           TypeRep a → TypeRep b → AppResult (a b)
```

In *AppResult*, note that k_1, the kind of b, is *existentially* bound in this data structure, meaning that it does not appear in the kind of the result type ($a\ b$). We know the result kind of the type application but there is no way to know the kinds of the subcomponents.

With kind polymorphism in mind, let's add some type annotations to see what existential variables are introduced by the two calls to *splitApp* in *dynFst*:

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn (rpab :: TypeRep pab) (x :: pab))
   = do App (rpa :: TypeRep pa) (rb :: TypeRep b) ← splitApp rpab
              -- introduces kind k2, and types pa :: k2 → ⋆, b :: k2
        App (rp  :: TypeRep p)  (ra :: TypeRep a) ← splitApp rpa
              -- introduces kind k1, and types p :: k1 → k2 → ⋆, a :: k1
        Refl ← eqT rp (typeRep :: TypeRep (,))
              -- introduces p ∼ (,) and (k1 → k2 → ⋆) ∼ (⋆ → ⋆ → ⋆)
        return (Dyn ra (fst x))
```

Focus on the arguments to the call to *eqT* in the third line. We know that:

- $rp\ \ \ \ \ \ \ :: TypeRep\ p$ and $p\ \ :: k_1 \to k_2 \to \star$
- $typeRep :: TypeRep\ (,)$ and $(,) :: \star\ \to \star\ \to \star$

So *eqT* must compare the *TypeReps* for two types of different kinds; if the run-time test succeeds, we know not only that $p \sim (,)$, but also that $k_1 \sim \star$ and $k_2 \sim \star$. That is, the pattern match on *Refl* GADT constructor brings local *kind equalities* into scope, as well as *type equalities*.

We can make this more explicit by writing out kind-annotated definitions for (:≈:) and *eqT*, thus:

```
eqT :: ∀ k1 k2 (a :: k1) (b :: k2). TypeRep a → TypeRep b → Maybe (a :≈: b)
data (a :: k1) :≈: (b :: k2) where
  Refl :: ∀ k (a :: k). a :≈: a
```

If the two types are the same, then *eqT* returns a proof that the types are equal *and* simultaneously a proof that the kinds are equal: a heterogeneous (often referred to as "John Major") equality [McB02].

In the case of *dynFst*, if *eqT* succeeds, the type checker can conclude $(k_1 \to\ \ k_2 \to \star) \sim (\star \to \star \to \star)$ and $p \sim (,)$. The GHC constraint solver uses these equalities to conclude that the type of x is (a, b), validating the projection *fst x*.

The addition of first-class kind equalities, to accompany first-class type equalities, is the most recent innovation in GHC 8.0. Indeed, they motivate a systemic change, namely collapsing types and kinds into a single layer, so that we have $\star :: \star$. This change is described and motivated in a recent paper [WHE13]. Type-safe decomposition of type representations is a compelling motivation for kind equalities.

5.6 Visible vs. Invisible Type Representations

Here are two functions with practically identical functionality:

```
cast  :: (Typeable a, Typeable b) ⇒ a → Maybe b
castR :: TypeRep a → TypeRep b → a → Maybe b
```

A *Typeable* class constraint is represented at runtime by a value argument, a *Typeable* "dictionary" in the jargon of type classes [WB89]. A dictionary is just a record of the methods of the class. Since *Typeable a* has only one method, a *Typeable a* dictionary is represented simply by a *TypeRep a* value. So, in implementation terms the function *cast* actually takes two *TypeRep* arguments exactly like *castR*. It's just that *castR* takes visible *TypeRep* arguments, while *cast* takes invisible (compiler-generated) *Typeable* arguments. So which is "better"?

The answer is primarily stylistic. Sometimes, in library code that manipulates many different *TypeRep* values, it is much clearer to name them explicitly, as we have done in the earlier examples in this section. But in other places (usually client code) it is vastly more convenient to take advantage of type classes to construct relevant *Typeable* evidence.

The two are, of course, equally expressive, since the implementation is the same in either case. Going from an implicit type representation (*Typeable*) to an explicit one (*TypeRep*) is easy, if inscrutable: just use the method *typeRep*. For example, here is how to define *cast* using *castR*:

```
cast :: (Typeable a, Typeable b) ⇒ a → Maybe b
cast = castR typeRep typeRep
```

The two calls to *typeRep* are at different types, but that is not very visible in the code. That is why it is often clearer to pass *TypeRep* values explicitly. But for the *caller* of *cast* is it much easier to pass invisible arguments; for example, in the call:

```
(cast x) :: Maybe Bool
```

the compiler will construct a *TypeRep* for *x*'s type and one for *Bool*, both wrapped as *Typeable* dictionaries, and pass them to *cast*.

However, going from explicit to implicit is not as easy. Suppose we have a *TypeRep a* and we wish to call a function with a *Typeable a* constraint. We essentially need to invent an **instance** *Typeable a* on the spot. Haskell provides no facility for local instances, chiefly because doing so would imperil class coherence.[9] In the context of type representations, though, incoherence is impossible:

[9] Though, some Haskellers have hacked around this restriction with abandon. See Kiselyov and Shan [KS04] and Edward Kmett's *reflection* package (at http://hackage.haskell.org/package/reflection).

there really is only one *TypeRep a* in existence, and so one *Typeable a* instance is surely the same as any other. Our API thus provides the following additional function *withTypeable*, which we can use to close the loop by writing *castR* in terms of *cast*:

```
withTypeable :: TypeRep a → (Typeable a ⇒ r) → r
castR :: TypeRep a → TypeRep b → a → Maybe b
castR ta tb = withTypeable ta (withTypeable tb cast)
```

We cannot implement *withTypeable* in Haskell source. But we *can* implement it in GHC's statically-typed intermediate language, System FC [SCPJD07]. The definition is simple, roughly like this:

```
withTypeable tr k = k tr   -- Not quite right
```

Its second argument *k* expects a *Typeable* dictionary as its value argument. But since a *Typeable* dictionary is represented by a *TypeRep*, we can simply pass *tr* to *k*. When written in System FC there is a type-safe coercion to move from *TypeRep a* to *Typeable a*, but that coercion is erased at runtime. Since the definition can be statically type checked, *withTypeable* does not form part of the trusted code base.

5.7 Comparing TypeReps

It is sometimes necessary to use type representations in the key of a map. For example, Shake [Mit12] uses a map keyed on type representations to look up class instances (dictionaries) at runtime; these instances define class operations for the types of data stored in a collection of *Dynamic*s. Storing the class operations once per type, instead of with each *Dynamic* package, is much more efficient.[10]

More specifically, we would like to implement the following interface:

```
data TyMap
empty  :: TyMap
insert :: Typeable a ⇒ a → TyMap → TyMap
lookup :: Typeable a ⇒ TyMap → Maybe a
```

But how should we implement these type-indexed maps? One option is to use the standard Haskell library *Data.Map*. We can define the typed-map as a map between the type representation and a dynamic value.

```
data TypeRepX where
  TypeRepX :: TypeRep a → TypeRepX
type TyMap = Map TypeRepX Dynamic
```

[10] See also http://stackoverflow.com/q/32576018/791604 for another use case for a map keyed on type representations.

Notice that we must wrap the *TypeRep* key in an existential *TypeRepX*, otherwise all the keys would be for the same type, which would rather defeat the purpose! The *insert* and *lookup* functions can then use *toDynamic* and *fromDynamic* to ensure that the right type of value is stored with each key.

```
insert   :: ∀ a. Typeable a ⇒ a → TyMap → TyMap
insert x = Map.insert (TypeRepX (typeRep :: TypeRep a)) (toDynamic x)
lookup :: ∀ a. Typeable a ⇒ TyMap → Maybe a
lookup = fromDynamic <=< Map.lookup (TypeRepX (typeRep :: TypeRep a))
```

Because *Data.Map* uses balanced binary trees, *TypeRepX* must be an instance of *Ord*:

```
instance Ord TypeRepX where
  compare (TypeRepX tr1) (TypeRepX tr2) = compareTypeRep tr1 tr2
compareTypeRep :: TypeRep a → TypeRep b → Ordering   -- primitive
```

The *TypeRep* API includes a comparison function *compareTypeRep* that compares two *TypeRep*s, indexed by possibly-different types *a* and *b*. Notice that we cannot make an instance for *Ord* (*TypeRep a*): if we compare two values both of type *TypeRep t*, following the signature of *compare*, they should always be equal!

Alternatively, we could use a more strongly typed data structure that internally keeps track of the dependency between the key and the element type. A simple example might be the following binary search tree:

```
data TyMap = Empty | Node Dynamic TyMap TyMap
```

Of course, much more general structures are also possible.[11]

Looking up values in this tree requires comparing the ordering of type representations. We could implement this comparison using the ordering for *TypeRepX*, but that implementation is clumsy. Once we have found the value, we must do an extra cast to show that it has the correct type.

```
lookup :: TypeRep a → TyMap → Maybe a
lookup tr1 (Node (Dyn tr2 v) left right) =
  case compareTypeRep tr1 tr2 of
    LT → lookup tr1 left
    EQ → castR tr2 tr1 v   -- know this cast will succeed
    GT → lookup tr1 right
lookup tr1 Empty = Nothing
```

However, we can improve this implementation using the following *more informative* comparison function, thereby avoiding this redundant check. In particular, when the two type representations are equal, this function will return an equality proof, just like *eqT*.

[11] The *dependent-map* library is an example of such a data structure. See https://hackage.haskell.org/package/dependent-map-0.1.1.3/docs/Data-Dependent-Map.html.

```
cmpT :: TypeRep a → TypeRep b → OrderingT a b
    -- definition is primitive


data OrderingT a b where
  LTT :: OrderingT a b
  EQT :: OrderingT t t
  GTT :: OrderingT a b
```

It is, of course, trivial to define *compareTypeRep* in terms of *cmpT*.

5.8 Representing Polymorphic and Kind-Polymorphic Types

Our interface does not support representations of polymorphic types, such as *TypeRep* ($\forall$ *a*. *a* $\to$ *a*). Although plausible, supporting those in our setting brings in a whole new range of design decisions that are as of yet unexplored (e.g. higher-order representations vs. de-Bruijn?). Furthermore, it requires the language to support impredicative polymorphism (the ability to instantiate quantified variables with polymorphic types, for instance the *a* variable in *TypeRep a* or *Typeable a*), which GHC currently does not. Finally, representations of polymorphic types have implications on semantics and possibly parametricity, an issue that we discuss in the next section.

Similarly, constructors with polymorphic kinds would require impredicative *kind* polymorphism. A representation of type *TypeRep* (*Proxy* :: $\forall$ *kp*. *kp* $\to \star$) would require the kind parameter *k* of *TypeRep* (*a* :: *k*) to be instantiated to the polymorphic kind $\forall$ *kp*. *kp* $\to \star$. Type inference for impredicative kind polymorphism is no easier than for impredicative type polymorphism and we have thus excluded this possibility.

5.9 Summary

It is time to draw breath. We used the *ST* example to motivate a *Dynamic* type (Sect. 3); then we used *Dynamic* to motivate type-indexed type representations (Sect. 5.1); and in the rest of Sect. 5 we have discussed the various operations we need over those representations. Our final API for *TypeRep* is summarised in Fig. 1.

Why do we make *TypeRep* primitive rather than *Dynamic*? When we design a primitive, built-in feature for a language, we seek the smallest, most sharply-focused feature that serves the need. We need built-in support for something like *TypeRep* and the *Typeable* class to implement *Dynamic*. If the *Dynamic* library becomes just an ordinary library, with no uses of *unsafeCoerce*, that usefully shrinks the trusted code base. Moreover, *TypeRep* is independently useful to support other abstractions (not just *Dynamic*), as we describe in Sect. 8.

```
  -- Related definitions ──────────────────────────────────────
  -- Informative propositional equality
data (a :: k1) :≈: (b :: k2) where
  Refl :: ∀ k (a :: k). a :≈: a
  -- An informative ordering type, asserting type equality in the EQ case
data OrderingT a b where
  LTT :: OrderingT a b
  EQT :: OrderingT t t
  GTT :: OrderingT a b
  -- Data.Typeable ────────────────────────────────────────────
data TypeRep (a :: k)
    -- primitive, indexed by type and kind
instance Show (TypeRep a)
class Typeable (a :: k) where
  typeRep :: TypeRep a
    -- Typeable instances automatically generated for all type constructors
  -- class access
withTypeable :: TypeRep a → (Typeable a ⇒ r) → r
  -- existential version
data TypeRepX where
  TypeRepX :: TypeRep a → TypeRepX
instance Eq   TypeRepX
instance Ord  TypeRepX
instance Show TypeRepX
  -- comparison
eqT  :: TypeRep a → TypeRep b → Maybe (a :≈: b)
cmpT :: TypeRep a → TypeRep b → OrderingT a b
  -- construction
mkTyApp :: TypeRep a → TypeRep b → TypeRep (a b)
  -- pattern matching
splitApp :: TypeRep a → Maybe (AppResult a)
data AppResult (t :: k) where
  App :: TypeRep a → TypeRep b → AppResult (a b)
  -- information about the "head" type constructor
tyConPackage :: TypeRep a → String
tyConModule  :: TypeRep a → String
tyConName    :: TypeRep a → String
```

Fig. 1. New Typeable interface

6 Implementation

How do we implement type representations? We use a GADT like this:

```
data TypeRep (a :: k) where
  TrApp  :: TypeRep a → TypeRep b → TypeRep (a b)
```

```
    TrTyCon :: TyCon → TypeRep k → TypeRep (a :: k)
data TyCon = TyCon { tc_module :: Module, tc_name :: String }
data Module = Module { mod_pkg :: String, mod_name :: String }
```

The *TyCon* type is a runtime representation of the "identity" of a type constructor. For every datatype declaration, GHC silently generates a binding for a suitable *TyCon*. For example, for *Maybe* GHC will generate:

```
$tcMaybe :: TyCon
$tcMaybe = TyCon { tc_module = Module { mod_pkg   = "base"
                                      , mod_name = "Data.Maybe" }
                 , tc_name   = "Maybe" }
```

The name $\$tcMaybe$ is not directly available to the programmer. Instead (this is the second piece of built-in support), GHC's type-constraint solver has special behaviour for *Typeable* constraints, as follows.

To solve *Typeable* $(t_1\ t_2)$, GHC simply solves *Typeable* t_1 and *Typeable* t_2, and combines the results with *TrApp*. To solve *Typeable* T where T is a type constructor, the solver uses *TrTyCon*. The first argument of *TrTyCon* is straightforward: it is the (runtime representation of the) type constructor itself, e.g. $\$tcMaybe$.

But *TrTyCon* also stores the representation of the kind of this very constructor, of type *TypeRep* k. Recording the kind representations is important, otherwise we would not be able to distinguish, say, $Proxy :: \star \to \star$ from $Proxy :: (\star \to \star) \to \star$, where *Proxy* has a polymorphic kind ($Proxy :: \forall\ k.\ k \to \star$). We do not support direct representations of kind-polymorphic constructors like *Proxy*, for reasons outlined in Sect. 5.8; rather *TrTyCon* encodes the *instantiation* of a kind-polymorphic constructor (such as *Proxy*). There is a limitation here: the kind of the instantiated type constructor must be monomorphic (again, for reasons outlined in Sect. 5.8), so (a) the type constructor must have a rank-1 prenex-polymorphic kind, and (b) it can be instantiated only with monomorphic kinds (i.e. predicatively).

Notice that *TrTyCon* is fundamentally insecure: you could use it to build a *TypeRep* t for any t whatsoever. That is why we do not expose the representation of *TypeRep* to the programmer. Instead the part of GHC's *Typeable* solver that builds *TrTyCon* applications is part of GHC's trusted code base.

Another reason for keeping *TypeRep* abstract is that it allows us to vary details of the representation. For example, the SYB pattern (Sect. 8.2) makes many equality tests for *TypeRep*. If we stored a fingerprint, or even a hash-value, in every node, we could make comparisons work in constant time. Another aspect that we might want to vary is how much meta-data we make available in a *TyCon*.

7 Properties of a Language with Reflection

The addition of features such as *Dynamic* and *TypeRep* has implications to the semantics and metatheory of a programming language.

Strong Normalization. The addition of type *Dynamic* (whether implemented with *TypeRep* or not) creates the possibility for loops without explicitly using recursion (nor recursive data types):

```
delta :: Dynamic → Dynamic
delta dn = case fromDynamic dn of
   Just f    → f dn
   Nothing → dn
loop = delta (toDynamic delta)
```

Effectively *Dynamic* behaves like a negative recursive datatype, a feature that breaks strong normalization.

A similar example can be encoded with the more primitive *TypeRep*, adapting an example from previous work [VW10] (Sect. 5.3):

```
data Rid = MkT (∀ a. TypeRep a → a → a)
rt :: TypeRep Rid
rt = typeRep
delta :: ∀ a. TypeRep a → a → a
delta ra x = case (eqT ra rt) of
   Just Refl → case x of MkT y → y rt x
   Nothing   → x
loop = delta rt (MkT delta)
```

These examples demonstrate that primitives like *TypeRep* or *Dynamic* cannot be incorporated in languages where logical consistency is required, such as Coq or Agda, without further restrictions (e.g. predicative polymorphism, or weaker elimination forms). Fortunately Haskell is not one of those.

Parametricity. Now that we have added *TypeRep*, *Typeable* and automatically-generated type representations as built-in features, the alert reader might wonder whether we have perhaps accidentally weakened parametricity, and thereby lost Phil's free theorems.

Fortunately, the type system makes it explicit when run-time reflection is used, and must do so *even though every type is Typeable*. Phil's free theorems would go out of the window if we allowed *cast* to have the type *cast* :: *a* → *Maybe b*! That is the principled reason for requiring explicit *Typeable* constraints. There is an operational reason too: the *Typeable* constraint is implemented as a runtime argument; if there was no explicit *Typeable* constraint we would be forced to pass a *Typeable* dictionary to every polymorphic function, just in case it needed it, which would be disgusting.

Thanks to those explicit type representation arguments, techniques that have appeared in previous work [VW10] for a *closed world* of representations can be used to deduce "free theorems" from the types of all polymorphic functions. When those functions include *TypeRep* arguments (or *Typeable* constraints) then such theorems can *still* be derived but are often not very informative.

We conjecture that these results will carry over to (a) an open world of type representations, and (b) representations of types that hide polymorphic types in their defining equations (such as *Rid* above; see Sect. 5.3 of [VW10]), but both directions are open problems – perhaps new puzzles for Phil to solve!

8 Other Applications of **TypeRep** and dynamic

One of the advantages of building *Dynamic* on top of *TypeRep* is that the latter is independently useful to build other abstractions. We briefly describe some of these applications in this section.

8.1 Variants of Dynamic

In ML, the extensible *exception* type is built-in, but not so in Haskell: it is programmed as a library using *Typeable*, using a design described by *"An extensible dynamically-typed hierarchy of exceptions"* [MPJMR01]. Here are the key definitions:

$throw\# :: SomeException \to a$

data $SomeException$ **where**
  $SomeException :: Exception\ e \Rightarrow e \to SomeException$

class $(Typeable\ e, Show\ e) \Rightarrow Exception\ e$ **where** $\{...\}$

The primitive exception-raising operation is *throw#*. Its argument is the fixed type *SomeException*, whose definition is an existential rather like *Dynamic*; the difference is that as well as having a *Typeable* superclass (which makes it like *Dynamic*), *Exception* also has *Show* superclass, and some methods of its own.

The fundamental data structure of the Shake build system [Mit12] is a directed acyclic graph (DAG) in which each node contains a value, a recipe for recomputing that value, and the dependencies of the node. If the values of any of the dependent nodes changes, the node's own value should be recomputed. The value in a node may be of any type, including types defined by the client of the Shake library, so again we have a fundamentally open-world problem. Shake solves this by ubiquitous use of dynamically typed values, both as keys and as values of its finite mappings [Mit12, Sect. 4.1]. For example, the Shake type *Any* is just like *Dynamic*, except that it includes *Binary*, *Eq*, *Hashable*, *Show*, and *NFData* constraints.

8.2 Supporting Generic Programming

Here is the opening example from *"Scrap Your Boilerplate"* [LPJ03]:

$increase :: Float \to Company \to Company$
$increase\ k = everywhere\ (mkT\ (incS\ k))$

A *Company* is a tree-shaped data structure describing a company; the function *increase* is supposed to find every employee in the data structure and increase his or her salary by *k*, using *incS* :: *Float* → *Salary* → *Salary*. The function *everywhere* applies its argument function to every node in the data structure; we will not consider it further here. Our focus is the function *mkT*, which depends critically on *Typeable*:

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
mkT f x = case (cast f) of
  Just g   → g x
  Nothing  → x
```

That is, *mkT* takes a type-specific function (such as *incS*) and lifts it to work on values of any type, as follows: if types match use *g*, otherwise use the identity function. (*cast* was described in Sect. 5.7.)

So the SYB approach to generic programming depends crucially on dynamic type tests. The popular Uniplate library for generic programming also makes essential use of comparison of *TypeRep*s, for a similar purpose as SYB [MR07]. Note, however, that *TypeRep* alone is not enough to support generic programming (see Sect. 9.5).

8.3 Distributed Programming and Persistence

Cloud Haskell is an Erlang-style library for programming a distributed system in Haskell [EBPJ11]. A key component of the implementation, described in the paper, is the ability to serialise a *code pointer* and send it from one node to another in the distributed system. This code pointer could be implemented in a variety of ways, such as: a machine address, a small integer, a long string, a URL.

But regardless of how it is implemented, the receiving node must deserialise the code pointer to some code. To guarantee that the code is then applied to appropriately typed values, the receiving node must perform a dynamic type test. That way, even if the code pointer was corrupted in transit, by accident or malice, the receiving node will be type-sound. A simple way to do this is to serialise a code pointer as a key into a *static pointer table* containing *Dynamic* values. When receiving code pointer with key *k*, the recipient can lookup up entry *k* in the table, find a *Dynamic*, and check that it has the expected type. The key can be an integer, a string, or whatever; regardless, if it is corrupted, the recipient might access the wrong entry in the table, but the type test will ensure soundness.

In other variants of this idea, one might want to serialise and deserialise values of type *Dynamic*, and hence of type *TypeRep*. It turns out that this raises some quite interesting issues that are beyond the scope of this paper[12].

[12] See the tree of wiki pages rooted at https://ghc.haskell.org/trac/ghc/wiki/DistributedHaskell for lots more information.

8.4 Meta Programming

It is perhaps unsurprising that meta programming for a statically typed target language often involves type reflection. For example, it is popular to define a type-indexed version of the syntax tree of expressions, so that a value of type *Expr t* is a syntax tree for an expression of type *t*. But then what is the type of a front end for the language, which takes a *String*, parses it (presumably to an un-typed syntax tree), and then typechecks it to reject type-incorrect programs. The front end cannot have this type

```
frontEnd :: String → Maybe (Expr a)   -- No!
```

because that is too polymorphic. The type *a* has to depend on the contents of the string! So we need something more like this:

```
data DynExp where
  DE :: TypeRep a → Expr a → DynExp
frontEnd :: String → DynExp
```

Here *frontEnd* returns an existential pair of a *Expr a*, and a *TypeRep* that describes the type of the expression. The earliest paper we have found that clearly embodies this idea is *"Tagless staged interpreters for typed languages"* [PTS02], but it has become more widespread since Haskell has supported this programming style [GM08, MCGN15].

9 Related Work

9.1 The Old Implementation of Typeable and Dynamic

GHC has supported *Dynamic* and a non-indexed version of *TypeRep* for some time. Here is the essence of the implementation in GHC 7.10:

```
data TypeRep   -- Abstract
class Typeable a where
  typeRep :: proxy a → TypeRep
data Dynamic where
  Dyn :: TypeRep → a → Dynamic
data Proxy a = Proxy
```

Typeable had built-in support, so that newly declared data types would automatically get *Typeable* instances. But *TypeRep* was not indexed, so there was no connection between the *TypeRep* stored in a *Dynamic* and the corresponding value. Indeed, accessing the *typeRep* required a *proxy* argument to specify the type that should be represented.

Because there is no connection between types and their representations, this implementation of *Dynamic* requires *unsafeCoerce*. For example, here is the old *fromDynamic*:

```
fromDynamic :: ∀ d. Typeable d ⇒ Dynamic → Maybe d
fromDynamic (Dyn trx x)
  | typeRep (Proxy :: Proxy d) == trx = Just (unsafeCoerce x)
  | otherwise                         = Nothing
```

Likewise, *unsafeCoerce* was used in the definition of *dynApply*:

```
dynApply :: Dynamic → Dynamic → Maybe Dynamic
dynApply (Dyn trf f) (Dyn trx x) =
   case splitTyConApp trf of
     (tc, [t1, t2]) | tc == funTc && t1 == trx →
        Just (Dyn t2 ((unsafeCoerce f) x))
     _ → Nothing
```

Here *splitTyConApp*, a special definition from *Data.Typeable* that decomposes type representations, and *funTc* is the function type constructor. There is nothing special about function types. Other forms of data also need their own unsafe elimination functions to be defined. For example, the implementations of *dynFst* and *dynSnd* are similar, and also require *unsafeCoerce*.

Furthermore, the library designer cannot hide all such uses of *unsafeCoerce* from the user. If they want to use their own parameterized type with *Dynamic*, then they too must use *unsafeCoerce*. In short, the old interface to this library is not expressive enough to work with type representations and dynamics safely.

9.2 Dynamics in a Closed World

We have focused entirely on an open-world setting (Sect. 4). If, however, your application can work with a closed world, with a predetermined set of type constructors, then simpler designs are available.

Universal Datatype Implementation of Dynamic. The most obvious way to implement closed-world dynamics is to use a *universal datatype* to represent dynamic values (see Sect. 4). But even in a closed-world setting this approach is unsatisfactory. Most prominently it suffers from a serious efficiency problem, because converting a value to or from *Dynamic* traverses the value itself. For example, to convert an (*Int*, *Bool*) pair to a *Dynamic*, we would have to deep-copy the value, and then do the reverse when we get it out. This is silly: in the *ST* example of Sect. 3 we only want to store the value in the *Store*, and read it back out later; we shouldn't need to *process* the value in any way whatsoever! Processing the value has a semantic problem too: a dynamic type test forces evaluation of a term, so it will fail on diverging terms; in our *ST* example, we could not store bottom in the *Store*.

GADT-Based Type Representations. In a closed-world setting, *TypeRep* can be implemented as an ordinary library without built-in support, thus:

```
data TypeRep (a :: ⋆) where
  TBool :: TypeRep Bool
  TFun  :: TypeRep a → TypeRep b → TypeRep (a → b)
  TProd :: TypeRep a → TypeRep b → TypeRep (a, b)
    ...
```

With this representation for *TypeRep*, the functions *eqT*, *dynApply*, *dynFst*, etc., are all easily written, using pattern-matching on the *TypeRep* GADT. The trouble with this approach is simply that it is not extensible: the set of representable types is limited those with data constructors in *TypeRep*.

History of Encoding Type Representations. In concurrent work, Cheney and Hinze [CH02] and Baars and Swierstra [BS02] showed how to implement type *Dynamic* by first encoding indexed type representations, similar to the GADT shown above. (GADTs were not a part of GHC at that time). These type representation encodings were based on earlier work by Yang [Yan98] and Weirich [Wei04]. In particular, Yang showed how to encode the *TypeRep* type above using higher-order polymorphism (available in the ML module system). And Weirich used type classes to encode a version of type Dynamic that supported *type-safe* cast (i.e. *toDynamic* and *fromDynamic*) but could not destruct types (i.e. no *dynApply*).

9.3 Dynamic Typing in Other Statically-Typed Languages

Several statically typed languages include a *Dynamic* type as a language primitive. Abadi et al. [ACPP91, ACPR95] laid the groundwork for such extensions, by incorporating a special type *Dynamic* to contain values of unknown type. Values of this type could be refined using a *typecase* operator that allowed pattern matching on the actual type of the value.

Clean. The Clean language includes a dynamic type as well as a class constraint similar to *Typeable*, called *TC*, for types that support "type codes" [Pil99]. Unlike Haskell, where *Dynamic* is defined in terms of *Typeable*, *both* of these structures are language primitives in Clean. Pil [Pil99] makes the case that type *Dynamic* is not enough on its own; languages should also include something like *Typeable*. Making *Dynamic* a language primitive is powerful. For example, Clean can support polymorphic values embedded into dynamics. In other words, types such as $\forall\, a.\, a \to a$ are *Typeable*, unlike in Haskell (Sect. 5.8).

Clean's interface to runtime types is also different from a user perspective. Clean includes runtime type unification using type-pattern variables. This features subsumes both runtime type equality (as in our *eqT*) as type patterns may be nonlinear, and type destruction (as in our *splitApp*). We conjecture that this difference is cosmetic.

OCaml. Leroy and Mauny [LM91] used similar mechanism to Abadi et al. in order to extend the ML language with a dynamic type. However, more recent extensions in support of dynamic typing in OCaml have been proposed in workshop talks [Fri11, HG13]. Like the design presented here, these extensions include a type for type representations `'a ty`, and a comparison operation that returns a GADT-based witness for type equality.

These extensions also include a mechanism to decompose type representations. In this case, every type constructor needs its own a decomposition GADT and function. For example (using Haskell syntax), the pair type constructor would be accompanied by the following:

```
data IsPair a where
  TypePair :: TypeRep a → TypeRep b → IsPair (a, b)
isPair :: TypeRep a → Maybe (IsPair a)   -- Primitive implementation
```

These definitions could then be used to implement *dynFst*.

```
dynFst :: Dynamic → Maybe Dynamic
dynFst (Dyn tab x) = case isPair tab of
  Just (TypePair ta tb) → Just (fst x)
  Nothing → Nothing
```

Our *AppResult* GADT and *splitApp* function uses GHC's kind-polymorphism to generalize these definitions for any type constructor.

9.4 Reflection in Java

In a language with subtyping and down-casting, a maximal supertype acts like a dynamic type. For example, in Java, a reference type, like *String*, can be coerced to *Object* (the maximal supertype) without runtime overhead. Furthermore, Java also supports a simple equivalent of *fromDynamic* using a runtime cast.

Here is another connection to Phil, who introduced generics to Java [BOSW98]. Java's generics are limited in two (related) ways: Java has no notion of higher-kinded types, and type parameters to generic classes (such as $List\langle T\rangle$) and methods are erased, disallowing dynamic checks involving those parameters.

Generics have enhanced Java's reflection feature, which has remarkable parallels to Haskell's. From its first versions, Java had the *Class* type, which was useful for queries about a type, but not for casting. Phil's generics then allowed *Class* to be type-indexed [NW06], just like we are doing with *TypeRep*. For any reference type *T* in modern Java, the expression *T*.**class** is a runtime type representation, of type $Class\langle T\rangle$. With a $Class\langle T\rangle$ in hand, we can cast an arbitrary value to *T*, even if *T* is a type parameter.

9.5 Generic Programming and Type Representations

The type representations described in this paper are not designed for full-blown *datatype generic* programming [Gib07]. *TypeRep* allows one to compare and explore just the shape of a type including names of type constructors and type arguments. But generic programming requires the additional capability to generically explore the structure of the *values* inhabiting a type; for instance by allowing one to iterate over the data constructors of some unknown type without having the actual type definition in scope.

There are many ways to support generic programming; from isomorphisms of types to sums and products (for example the *Generic* class in Haskell [MDJL10]), to providing built-in generic instances for iterators (for example the *Data* class in Haskell [LPJ03]), to advanced variants of *TypeRep* that additionally include type-indexed data structures for describing data constructors and introducing/eliminating values generically [Wei06].

10 Conclusions and Further Work

In this paper, we have designed a powerful API for type reflection and shown that it can be used to implement a flexible and extensible dynamic type. The next major challenge is to provide a better story for polymorphic dynamic values (Sect. 5.8).

Acknowledgements. Thanks to Neil Mitchell both for help in understanding Shake and for a rapid review. Kenneth Foner also provided feedback on a draft. Ben Price's internship at Microsoft Research was helpful in bringing some of the key issues to the surface. This paper was typeset with *lhs2TeX* (http://www.andres-loeh.de/lhs2tex/).

Finally, thanks to Phil, who has spent much of his professional life reflecting on types. Increasingly, Haskell can too. Happy birthday Phil!

References

[ACPP91] Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically-typed language. ACM Trans. Program. Lang. Syst. **13**(2), 237–268 (1991)

[ACPR95] Abadi, M., Cardelli, L., Pierce, B., Rémy, D.: Dynamic typing in polymorphic languages. J. Funct. Program. **5**(1), 111–130 (1995)

[BOSW98] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: adding genericity to the Java programming language. In: Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 183–200. ACM (1998)

[BS02] Baars, A.I., Swierstra, D.: Typing dynamic typing. In: International Conference on Functional Programming, pp. 157–166. ACM (2002)

[CH02] Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Workshop on Haskell, pp. 90–104. ACM (2002)

[EBPJ11] Epstein, J., Black, A.P., Peyton Jones, S.: Towards Haskell in the cloud. In: Haskell Symposium. ACM (2011)

[EW12] Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Haskell Symposium. ACM (2012)

[Fri11] Frisch, A.: Runtime types in OCaml. In: Presentation at Meeting of the Caml Consortium, November 2011

[Gib07] Gibbons, J.: Datatype-generic programming. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 1–71. Springer, Heidelberg (2007)

[GM08] Guillemettte, L.-J., Monnier, S.: A type-preserving compiler in Haskell. In: International Conference on Functional Programming. ACM (2008)

[HG13] Henry, G., Garrique, J.: Dynamic typing in OCaml. Presentation at Nagoya University (2013)

[HHPJW07] Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A history of Haskell: being lazy with class. In: Conference on History of Programming Languages (2007)

[KS04] Kiselyov, O., Shan, C.-C.: Functional pearl: implicit configurations-or, type classes reflect the values of types. In: Workshop on Haskell, pp. 33–44. ACM (2004)

[LM91] Leroy, X., Mauny, M.: Dynamics in ML. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 406–423. Springer, Heidelberg (1991)

[LPJ95] Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp Symb. Comput. **8**(4), 293–341 (1995)

[LPJ03] Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. In: Workshop on Types in Languages Design and Implementation. ACM (2003)

[LPJ05] Lämmel, R., Peyton Jones, S.: Scrap your boilerplate with class: extensible generic functions. In: International Conference on Functional Programming. ACM (2005)

[McB02] McBride, C.: Elimination with a motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 197–216. Springer, Heidelberg (2002)

[MCGN15] McDonell, T., Chakravarty, M., Grover, V., Newton, R.: Type-safe runtime code generation. In: Haskell Symposium, pp. 201–212. ACM (2015)

[MDJL10] Magalhaes, J.P., Dijkstra, A., Jeuring, J., Loeh, A.: A generic deriving mechanism for Haskell. In: Haskell Symposium, pp. 37–48. ACM (2010)

[Mit12] Mitchell, N.: Shake before building: replacing Make with Haskell. In: International Conference on Functional Programming. ACM (2012)

[MPJMR01] Marlow, S., Peyton Jones, S., Moran, A., Reppy, J.: Asynchronous exceptions in Haskell. In: Programming Language Design and Implementation. ACM (2001)

[MR07] Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Workshop on Haskell. ACM (2007)

[NW06] Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly Media, Sebastopol (2006)

[Pil99] Pil, M.: Dynamic types and type dependent functions. In: Hammond, K., Davie, T., Clack, C. (eds.) IFL 1998. LNCS, vol. 1595, pp. 169–185. Springer, Heidelberg (1999)

[PJVWW06] Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: International Conference on Functional Programming, pp. 50–61. ACM (2006)

[PJW93] Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Principles of Programming Languages. ACM (1993)

[PTS02] Pasalic, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. In: International Conference on Functional Programming. ACM (2002)

[SCPJD07] Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: Workshop on Types in Languages Design and Implementation. ACM (2007)

[VW10] Vytiniotis, D., Weirich, S.: Parametricity, type equality, and higher-order polymorphism. J. Funct. Program. **20**, 175–210 (2010)

[Wad89] Wadler, P.: Theorems for free! In: International Conference on Functional Programming Languages and Computer Architecture. ACM (1989)

[Wad90] Wadler, P.: Comprehending monads. In: Conference on LISP and Functional Programming. ACM (1990)

[WB89] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Principles of Programming Languages, pp. 60–76. ACM (1989)

[Wei04] Weirich, S.: Type-safe cast. J. Funct. Program. **14**(6), 681–695 (2004)

[Wei06] Weirich, S.: Replib: a library for derivable type classes. In: Workshop on Haskell. ACM (2006)

[WHE13] Weirich, S., Hsu, J., Eisenberg, R.A.: System FC with explicit kind equality. In: International Conference on Functional Programming, pp. 275–286. ACM (2013)

[XCC03] Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Principles of Programming Languages. ACM (2003)

[Yan98] Yang, Z.: Encoding types in ML-like languages. In: International Conference on Functional Programming, pp. 289–300. ACM (1998)

[YWC+12] Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhaẽs, J.P.: Giving Haskell a promotion. In: Workshop on Types in Language Design and Implementation. ACM (2012)

The Essence of Multi-stage Evaluation in LMS

Tiark Rompf(✉)

Purdue University, West Lafayette, USA
`tiark@purdue.edu`

Abstract. Embedded domain-specific languages (DSLs) are the subject of wide-spread interest, and a variety of implementation techniques exist. Some of them have been invented, and some of them discovered. Many are based on a form of *generative* or *multi-stage* programming, where the host language program builds up DSL terms during its evaluation. In this paper, we examine the execution model of LMS (Lightweight Modular Staging), a framework for embedded DSLs in Scala, and link it to evaluation in a two-stage lambda calculus. This clarifies the semantics of certain ad-hoc implementation choices, and provides guidance for implementing similar multi-stage evaluation facilities in other languages.

Keywords: Multi-stage programming · Scala · Domain-specific languages · LMS (Lightweight Modular Staging) · Partial evaluation

1 Introduction

Embedded domain-specific languages (DSLs) are the subject of wide-spread interest across communities and languages. If a host-language program computes and assembles DSL terms, we are dealing with a form of *generative* or *multi-stage* programming [65]. Exactly how this is done depends on the language and framework that is used. But in any case, the interactions between meta-language and object-language (DSL) semantics can be non-trivial.

In this paper, we examine the evaluation mechanism of LMS (Lightweight Modular Staging) [54]. LMS is a framework for building embedded DSLs and program generators in Scala that uses types to identify staged expressions: any normal Scala expression of type `Int`, `String`, or in general `T`, is evaluated at program generation time, and expressions of type `Rep[Int]`, `Rep[String]`, or in general `Rep[T]`, produce staged expressions that will evaluate to values of the corresponding type `T` when the generated code is run. In general, operations on `Rep[T]` mirror those on `T` but lifted to the `Rep` level. For example, adding two `Rep[Int]` values will produce a `Rep[Int]` result.

While this description succeeds in giving an overall idea of the approach, it leaves many details unspecified. For example, how does one deal with functions? Primitive constants of, e.g., type `Int` can be lifted to `Rep[Int]` values without much effort as constant expressions, but it is less clear how to create a staged function of type `Rep[A=>B]`. The approach chosen by LMS is to

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 318–335, 2016.
DOI: 10.1007/978-3-319-30936-1_17

use an explicit constructor fun that converts a present-stage function of type Rep[A]=>Rep[B] into a staged function of type Rep[A=>B].

Armed with this knowledge, we can build up some intuition on how Rep types enable interesting program transformations. Here is a staged version of Ackermann's function (note the use of the fun constructor):

```
val ack: Int => Rep[Int => Int] = { m: Int => fun { n: Rep[Int] =>
  if (m==0) n+1
  else if (n==0) ack(m-1)(1)
  else ack(m-1)(ack(m)(n-1))
}}
ack(2)
```

What will be the result of this computation? The type of ack is a mixed-stage function type Int => Rep[Int => Int], so ack(m) for a given integer m will have type Rep[Int => Int]: a staged function from Int to Int that corresponds to the body of ack but with all computations that depend on m but not n (nor any other Rep value) readily evaluated.

```
val ack0: Int => Int = { n: Int => n+1 }
val ack1: Int => Int = { n: Int => if (n==0) ack0(1) else ack0(ack1(n-1)) }
val ack2: Int => Int = { n: Int => if (n==0) ack1(1) else ack1(ack2(n-1)) }
ack2
```

The question we would like to address in this paper is: *how does all this work?* To do so, we will link the evaluation strategy of Scala and LMS to a two-stage lambda calculus. Such multi-level calculi have been popularized by Nielson and Nielson [46] and play a key role in the partial evaluation literature to describe *binding-time* information similar to the Rep[T] vs T distinction in LMS.

Our two-level language is derived from Chaps. 8 and 10 in the book by Jones, Gomard, and Sestoft [32]. We will build an evaluator for this calculus from the ground up and show how it computes the specialized ack function above, pointing to similarities and differences with systems from the literature.

While partial evaluation experts will most likely not find much novelty in this paper, we hope that this exposition will be useful in clarifying the semantics of multi-stage evaluation as it is implemented in LMS and that it may serve as guidance for implementing similar facilities in other languages or frameworks.

2 A Two-Stage Lambda Calculus

The term language of our calculus is as follows:

```
Exp ::= Lit(Int)   | Var(Int) | Let(Exp,Exp)
      | Lit2(Exp)
      | Lam(Exp)   | App(Exp,Exp)  | Tic
      | Lam2(Exp)  | App2(Exp,Exp) | Tic2
      | Ifz(Exp,Exp,Exp)  | Plus(Exp,Exp)  | Minus(Exp,Exp)
      | Ifz2(Exp,Exp,Exp) | Plus2(Exp,Exp) | Minus2(Exp,Exp)
```

For ease of implementation, we use DeBrujin levels to represent binders. The term syntax contains integer literals, variables, let expressions, lambda abstractions, function application, simple side effects (Tic), conditionals, and arithmetic. Most (but not all) constructs come in two versions. For example, Lam is the present-stage lambda abstraction and Lam2 is the future-stage lambda abstraction. We will see later that variables and let bindings need not be duplicated.

In what sense does this calculus model Scala and LMS? Since Rep types are just regular Scala types and operations on Rep[T] are implemented within the Scala language using operator overloading, Scala's local type inference mechanism essentially performs a local *binding-time analysis* [32] for us.

Here is the desugared version of ack:

```
val ack: Int => Rep[Int => Int] = { m: Int =>
  fun { n: Rep[Int] =>
    if (m==0) __plus(n, __lit(1))
    else      __ifThenElse(__equals(n, __lit(0)),
                __apply(ack(m-1), __lit(1)),
                __apply(ack(m-1), __apply(ack(m), __minus(n, __lit(1)))))
  }
}
ack(2)
```

We can see that Scala's typechecker has replaced all operations that involve Rep values with method calls like __ plus and __ minus, using standard language facilities like method overloading and implicits. These methods serve as smart constructors that create corresponding expression terms when the program is executed.

If we want to model the same behavior in our calculus, we can directly translate the program above to our two-level language, replacing the primitive Scala constructs with first-level terms and method calls with second-level terms:

```
Let(Lam(
  Lam2(Lam(
    Ifz(m,n+1,
      Ifz2(n,App2(App(ack,m-1),Lit2(Lit(1))),
             App2(App(ack,m-1),App2(App(ack,m),n-1)))))),
  App(ack,Lit(2)))
```

A few expressions have been factored out for clarity:

```
ack = Var(0)   m   = Var(1)              n   = Var(3)
               m-1 = Minus(m,Lit(1))     n-1 = Minus2(n,Lit2(Lit(1)))
                                         n+1 = Plus2(n,Lit2(Lit(1)))
```

Note that we map staged function definitions such as fun { m => ... } to nested expressions Lam2(Lam(...)). For a faithful correspondance, we need to model both the constructor invocation and the Scala function passed as argument. Likewise, we map staged constant literals like __lit(1) to nested terms Lit2(Lit(1)).

Now that we have introduced the two-level term language, how should multi-stage evaluation proceed? Intuitively, each future-stage operator should create a program term at runtime, but a priori it is not clear how these terms are to be composed. For example:

```
Let(Tic2,
  Lit2(Lit(1)))
```

This term could either evaluate to Lit(1), assuming that Tic2 evaluates to Tic, which is then discarded, or it could evaluate to Let(Tic,Lit(1)). Arguably, the second option is the desired one, but it takes some additional work to preserve the evaluation order and sharing behavior across evaluation stages. A well established approach to achieve this is to insert let-bindings eagerly for all "serious" expressions and pass around only atomic identifiers or primitve constants. We will build our multi-stage evaluator in steps, starting from a standard single-stage evaluator, and add the necessary let-insertion logic in a second step, by composing the evaluator with a transformation into administrative normal form (ANF) [23].

3 Single-Stage Evaluation

The starting point for our multi-stage evaluator is a completely standard lambda calculus evaluator with environments and closures:

```
Exp ::= Lit(Int) | Var(Int) | Tic | Lam(Exp) | Let(Exp,Exp) | App(Exp,Exp)
Val ::= Cst(Int) | Clo(Env,Exp)
Env   = List[Val]

var stC = 0
def tick() = { stC += 1; stC - 1 }

def eval(env: List[Val], e: Exp): Val = e match {
  case Lit(n)        => Cst(n)
  case Var(n)        => env(n)
  case Tic           => Cst(tick())
  case Lam(e)        => Clo(env,e)
  case Let(e1,e2)    => eval(env:+eval(env,e1),e2)
  case App(e1,e2)    =>
    val Clo(env3,e3) = eval(env,e1)
    eval(env3:+eval(env,e2),e3)
}
```

In Scala, the *snoc* operator `:+` appends an element to the right of a list, so `env:+eval(env,e1)` will first evaluate `e1` in environment `env`, and then append the result to the right of `env`. Variable lookup uses DeBruijn levels, where `env(n)` returns the `n`th element from the left of a list.

To make matters slightly more interesting, we include a side-effecting operation `Tic` in our language, which we implement using side effects of the meta language: the implicit Scala monad. Of course we could also build a purely functional version based on monads, but for our purposes it is beneficial to trade off this bit of impurity for ease in composition later.

4 ANF Conversion/Let-Insertion

The second building block is ANF conversion [23], which brings programs into a normal form that makes the evaluation order explicit in a program and extracts all intermediate results into let bindings. The ANF syntax is given by the nonterminal `N` as follows:

```
V ::= Lit(Int) | Var(Int)
M ::= V | Lam(N) | App(V,V) | Tic
N ::= V | Let(M,N)
```

We will implement ANF conversion again using side-effects in the meta language of our evaluator. We need two pieces of state: an accumulator for generated let bindings `stBlock` and a counter for the next fresh variable name `stFresh`. We also introduce a function `run` that implements a dynamic scoping discipline for these two variables:

```
var stFresh = 0
var stBlock: List[Exp] = Nil
def run[A](f: => A): A = {
  val sF = stFresh
  val sB = stBlock
  try f finally { stFresh = sF; stBlock = sB }
}
```

With this handling of mutable state in place, we can implement a first cut of our ANF conversion function:

```
def anf(env: List[Exp], e: Exp): Exp = e match {
  case Lit(n) => Lit(n)
  case Var(n) => env(n)
  case Tic    =>
    val s = Tic
    val f = stFresh
    stBlock:+= s
    stFresh += 1
    Var(f)
  case App(e1,e2) =>
    val ex1 = anf(env,e1)
    val ex2 = anf(env,e2)
    val s = App(ex1,ex2)
    val f = stFresh
    stBlock:+= s
    stFresh += 1
    Var(f)
  case Lam(e) =>
    val e1 = run {
      val f = stFresh
      stBlock = Nil
      stFresh += 1
      val res = anf(env:+Var(f),e)
      stBlock.foldRight(res)(Let)
    }
    val f = stFresh
    stBlock:+= Lam(e1)
    stFresh += 1
    Var(f)
  case Let(e1,e2) =>
    val ex1 = anf(env,e1)
    anf(env:+ex1,e2)
}
```

Function `anf` recurses structurally over the given term `e` and emits a let binding for each encountered non-trivial expression. For lambda abstractions, we use the dynamically scoped `run` construct to clear the stateful list of bindings when entering the nested scope and to reset the bindings and the fresh name reservoir when exiting the nested scope.

While this implementation gets the job done, it is not pretty. But it is easy to recognize some patterns: fresh variables are created in several places, and the emitting of let bindings can also be abstracted over. We factor out the repetitive operations as follows:

```
def fresh()          = { stFresh += 1; Var(stFresh-1) }
def reflect(s:Exp)   = { stBlock :+= s; fresh() }
def reify(f: => Exp) = run { stBlock = Nil; val last = f; stBlock.foldRight(last)(Let) }
```

Function `fresh` creates a fresh variable, `reflect` emits a let binding and returns the new identifier, and `reify` accumulates all let bindings created in a given block (note that `f` is a by-name parameter `=>Exp`).

Now we can go ahead and greatly simplify our implementation:

```
def anf(env: List[Exp], e: Exp): Exp = e match {
  case Lit(n)        => Lit(n)
  case Var(n)        => env(n)
  case Tic           => reflect(Tic)
  case Lam(e)        => reflect(Lam(reify(anf(env:+fresh(),e))))
  case App(e1,e2)     => reflect(App(anf(env,e1),anf(env,e2)))
  case Let(e1,e2)     => anf(env:+(anf(env,e1)),e2)
}
```

In fact, we no longer mention mutable state directly in the conversion function anymore, which brings it much closer to common definitions of ANF conversion based on evaluation contexts [23].

Based on this formulation it is easy to convince oneself that ANF conversion preserves semantics:

```
eval(Nil, anf(Nil, e)) = eval(Nil, e)
```

5 Multi-stage Evaluation

We now turn our attention to multi-stage evaluation. We start off with a slightly more uniform term language than the one given above:

```
Exp ::= Lit(Int)  | Var(Int)  | Tic  | Lam(Exp)  | Let(Exp,Exp)  | App(Exp,Exp)
      | Lit2(Int) | Var2(Int) | Tic2 | Lam2(Exp) | Let2(Exp,Exp) | App2(Exp,Exp)
Val ::= Cst(Int) | Clo(Env,Exp) | Code(Exp)
```

For every operation in the term language there is a corresponding future-stage variant that serves as *term constructor*. In addition, the syntax of values now includes a `Code(e)` case for future-stage expressions.

The key idea now is to combine present-stage evaluation with future-stage normalization into ANF. The ANF conversion part is adapted slightly to map present-stage term constructors (`App2`, `Lam2`, ...) to code values containing their normal term equivalents (`App`, `Lam`, ...). This wrapping in `Code` values necessitates a few tweaks to the helper functions:

```
def freshc()           = Code(fresh())
def reflectc(s: Exp)   = Code(reflect(s))
def reifyc(f: => Val) = reify { val Code(r) = f; r }
```

Now we are ready to put together `eval` and `anf` into a single multi-stage evaluator (Fig. 1). As we can see, term constructors like `App2` first evaluate their arguments in the present-stage to code values, from which the argument terms are extracted and the new `App` term is reflected.

It is important to note that there are only three ways in which code values are constructed: (1) via `reflectc`, which creates a variable reference; (2) via `freshc` in the environment, and then accessed through variable lookup; (3) directly from a constant literal. The results of the only call to `reifyc`, which may create non-atomic expressions, are only used to fill in the bodies of `Lam` terms which are itself reflected immediately.

Thus, all code values are atomic expressions, terminating, and side-effect free. They can be freely stored, duplicated and passed around without changing the meaning of the program. Staged expressions will appear in the generated code in the same order they were encountered when running the multi-stage program.

Partial evaluators that support side effects work in a similar way, inserting let-bindings to prevent reordering or duplication of code. Many specializers are implemented in continuation-passing style, which also leads to an elegant on-the-fly ANF conversion. A version that corresponds very closely to the one

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)         => Cst(n)
  case Var(n)         => env(n)
  case Tic            => Cst(tick())
  case Lam(e)         => Clo(env,e)
  case Let(e1,e2)     => evalms(env:+evalms(env,e1),e2)
  case App(e1,e2)     =>
    val Clo(env3,e3)  = evalms(env,e1)
    evalms(env3:+evalms(env,e2),e3)

  case Lit2(n)        => Code(Lit(n))
  case Var2(n)        => env(n)
  case Tic2           => reflectc(Tic)
  case Lam2(e)        => reflectc(Lam(reifyc(evalms(env:+freshc(),e))))
  case Let2(e1,e2)    => evalms(env:+evalms(env,e1),e2)
  case App2(e1,e2)    =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    reflectc(App(s1,s2))
}
```

Fig. 1. Multi-stage evaluation: composing eval and anf

above is shown by Lawall and Thiemann [37,67]. Our method of ANF conversion with direct-style `reflect` and `reify` operators can be seen as an instance of normalization by evaluation [17,22,40] or monadic reflection [21,44].

To obtain the full generated code for a top-level expression `e` one can use `reifyc(evalms(Nil,e))`. Another thing that becomes clear when looking at `evalms` is that `Var2` and `Let2` behave in exactly the same way as `Var` and `Let`. Therefore, we can drop them from our term language, as we had done in Sect. 2. Adding the other constructs, `Ifz`, `Plus`, and `Minus`, is straightforward.

6 Recursion

Where does this multi-stage evaluator leave us? We can almost, but not quite, run our `ack` example. The last missing bit is support for recursion. We could add an explicit `fix` combinator but instead, we chose to pass closures to themselves as first argument when called. Thus, all functions can potentially be recursive.

However, if we leave it at that, we will find that our example (and in fact many programs) will not terminate in the first stage. Here is a simpler example:

```
def f: Rep[Int => Int] = fun { n: Rep[Int] =>
  if (n == 0) 1 else f(n-1)
}
```

If we multi-stage evaluate this code naively, we will end up in a sequence of calls trying to compute the body of `f` — but in order to do that, we must compute the body of `f` again!

We need an additional insight: the body of `fun` in LMS or of `Lam2` in the calculus is itself an expression that will be evaluated. And if two functions have the same body expression, and the same free variables, then they must compute the same results for all inputs: intensional, i.e., structural equality implies extensional equality.

Thus, we can memoize the function expressions we are in the process of evaluating and tie the recursive knot to an enclosing function if we hit a recursive call. We achieve this by adding a small lookup table to the existing dynamically scoped mutable state:

```
var stFun: List[(Int,Env,Exp)] = Nil
def run[A](f: => A): A = {
  ...
  val sN = stFun
  try f finally { ...; stFun = sN }
}
```

In addition, we modify the handling of lambda abstractions to check whether we are trying to expand a given function body inside itself. If that is the case, we just return the already existing parent symbol. Otherwise, the function is new. We create an entry in our memo table and proceed to evaluate the body.

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case App(e1,e2) =>
    val Clo(env3,e3) = evalms(env,e1)
    val v2 = evalms(env,e2)
    evalms(env3:+Clo(env3,e3):+v2,e3)
  ...
  case Lam2(e) =>
    stFun collectFirst { case (n,`env`,`e`) => n } match {
      case Some(n) =>
        Code(Var(n))
      case None =>
        stFun :+= (stFresh,env,e)
        reflectc(Lam(reifyc(evalms(env:+freshc():+freshc(),e))))
    }
  ...
}
```

The code snippet also shows the modified `App` case that passes the closure to itself as first argument. The `Lam` case is updated correspondingly to create two fresh vars as placeholders for the arguments.

With this support for specializing recursive functions in place, we can successfully evaluate our `ack` example. We obtain the following result for `ack(2)`. Term syntax on the left, translated to Scala on the right:

```
Let(Lam(
  Let(Ifz(Var(1),
    Let(Lam(Let(Ifz(Var(3),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(App(Var(4),Lit(1)),Var(5))),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(Minus(Var(3),Lit(1)),Let(App(Var(2),Var(5)),
          Let(App(Var(4),Var(6)),Var(7)))))
      ),Var(4))),
      Let(App(Var(2),Lit(1)),Var(3))),
    Let(Lam(Let(Ifz(Var(3),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(App(Var(4),Lit(1)),Var(5))),
      Let(Lam(Let(Plus(Var(5),Lit(1)),Var(6))),
        Let(Minus(Var(3),Lit(1)),Let(App(Var(2),Var(5)),
          Let(App(Var(4),Var(6)),Var(7)))))
      ),Var(4))),
    Let(Minus(Var(1),Lit(1)),Let(App(Var(0),Var(3)),
      Let(App(Var(2),Var(4)),Var(5)))))
    ),Var(2))),
  Let(App(Var(0),Lit(2)),Var(1)))
```

```
val ack2: Int => Int = { n: Int =>
  if (n==0) {
    val ack1: Int => Int = { n: Int =>
      if (n==0) {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(1)
      } else {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(ack1(n-1))
      }
    }
    ack1(1)
  } else {
    val ack1: Int => Int = { n: Int =>
      if (n==0) {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(1)
      } else {
        val ack0: Int => Int = { n: Int => n+1 }
        ack0(ack1(n-1))
      }
    }
    ack1(ack2(n-1))
  }
}
ack2
```

We can see that this code does the right thing and corresponds to the right recursion pattern, but it contains duplicated function definitions for `ack0` and `ack1`. These are easily reclaimed by a code motion and common subexpression elimination or global value numbering algorithm. Hoisting functions and removing duplicates yields exactly the desired result:

```
val ack0: Int => Int = { n: Int => n+1 }
val ack1: Int => Int = { n: Int => if (n==0) ack0(1) else ack0(ack1(n-1)) }
val ack2: Int => Int = { n: Int => if (n==0) ack1(1) else ack1(ack2(n-1)) }
ack2
```

In the partial evaluation literature such patterns are known as polyvariant specialization [8]: one function in the source yields multiple specialized variants in the residual generated code. Sophisticated partial evaluators such as Similix or PGG implement similar techniques [6,28,37,67].

7 Relation to LMS

We have seen how we can model and describe a simple multi-stage evaluation algorithm for a small language. But how can we relate it to the actual implementation of LMS? Essentially we take the `evalms` function and turn it inside out, from an interpreter over an initial term language to an evaluator in tagless-final [10] style. But there is a problem: since we are working inside a *running* Scala program we cannot get our hands on *unevaluated* Scala expressions, as would be required by our implementation of staged literals and staged lambdas:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(n)          => Code(Lit(n))
  case Lam2(e)          => reflectc(Lam(reifyc(evalms(env:+freshc(),e))))
}
```

The argument `n` for `Lit2` would correspond to a literal constant in the Scala source code, and the argument `e` for `Lam2` corresponds to an unevaluated Scala expression, which we would evaluate at our own leisure, in an extended context, on the right hand side. The fact that tools like LMS operate as libraries inside a general-purpose host language is a key difference to offline partial evaluation systems that operate on the program text.

We generalize our implementation in such a way that `Lit2` takes an expression that *evaluates* to a constant, and `Lam2` takes an expression that *evaluates* to a function. The intention is that `Lit2(Lit(n))` and `Lam2(Lam(e))` correspond to the previous behavior. The construct `Lit2` now implements a *lifting* facility, which enables embedding of arbitrary values computed in the present stage as constants in the generated code. Note, however, that lifting (or *cross-stage-persistance*) is not automatically valid for all possible types. It is easy to see, for example, that a present-stage Scala closure does not necessarily have a valid representation as C source code. For this reason, we restrict `Lit` to integers here, and handle functions separately using `Lam2`. The implementation of `Lam2` can be seen as a pretty standard use of higher-order abstract syntax.

We modify our evaluator as follows to immediately evaluate all subexpressions and introduce the required indirection for functions:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(e)         => val Cst(n) = evalms(env,e); Code(Lit(n))
  case Tic2            => reflectc(Tic)
  case Lam2(e)         =>
    val f = evalms(env,e) // memoization elided
    reflectc(Lam(reifyc(evalms(env,App(f,freshc())))))
  case App2(e1,e2)     =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    reflectc(App(s1,s2))
}
```

As the last step towards our tagless-final interpreter, we refactor the evaluator to introduce constructor methods for each kind of term:

```
def evalms(env: Env, e: Exp): Val = e match {
  ...
  case Lit2(e)         => val Cst(n) = evalms(env,e); Code(lit(n))
  case Tic2            => Code(tic())
  case Lam2(e)         =>
    val f = evalms(env,e)
    val f1 = { x => val Code(r) = evalms(env,App(f,Code(x))); r }
    Code(lam(f1))
  case App2(e1,e2)     =>
    val Code(s1) = evalms(env,e1)
    val Code(s2) = evalms(env,e2)
    Code(app(s1,s2))
}
```

These term constructors are defined as follows to abstract over explicit environments, recursive calls to `evalms`, and `Code` values:

```
type Rep[T] = Exp
def lit(n: Int): Rep[Int]                      = Lit(n)
def tic()                                      = reflect(Tic)
def app[A,B](f:Rep[A=>B],x:Rep[A]): Rep[B]     = reflect(App(f,x))
def lam[A,B](f:Rep[A]=>Rep[B]): Rep[A=>B]      = reflect(Lam(reify(f(fresh()))))
                                                 // memoization elided
```

At this point, we no longer need the explicit evaluator, and we can just use the standard Scala evaluation to construct staged terms.

This API corresponds almost directly to the actual implementation in LMS. Behind the `reflect` and `reify` API, the internal representation of LMS is different though. Rather than as simple expression trees, LMS has a graph-based intermediate representation (IR) that directly supports code motion, common subexpression elimination, dead code elimination and a range of other optimizations [14, 54, 55]. LMS also makes pervasive use of smart constructors to perform rewriting while the IR is constructed.

But can we still handle recursion correctly in this setting? By switching away from our explicit evaluator we lose control over the representation of closures, which was key for detecting recursive calls through memoization. Fortunately, Scala implements closures as objects on the JVM, with free references stored as fields that can be queried via runtime reflection. In practice, LMS uses Java serialization to convert closures to byte arrays, and then takes this binary fingerprint as a key for memoization. This method neatly takes care of cyclic references, too. To implement a framework similar to LMS in other languages, a similar facility would likely be needed, or an alternative strategy for supporting recursive functions would need to be used (e.g. explicit staged fixpoint combinators).

8 Case Study: Regular Expression Matchers

While the `ack` example in the previous sections served as a good example for exposition, it is arguable not one that has a lot of practical uses. But the pattern directly translates to more interesting applications.

In this section, we give a short overview of a fast regular expression matcher that was developed using essentially the same technique by Nada Amin and the author. Part of the code and description below has appeared in the author's Ph.D. thesis [53] and in a paper at POPL 2013 [55]. Specializing string matchers and parsers is a popular benchmark in the partial evaluation and supercompilation literature [1,15,58,59,61,69].

We consider regular expression matchers that are "multi-threaded" and spawn a new conceptual thread to process alternatives in parallel. Of course these matchers do not actually spawn OS-level threads, but rather need to be advanced manually by client code. Thus, they are similar to coroutines.

From these non-deterministic matchers, we generate a set of mutually tail-recursive functions that implement a deterministic finite automaton (DFA) that recognizes the same regexp pattern without backtracking.

8.1 Regexp Matchers as Nondeterministic Finite Automata (NFA)

Here is a simple example for the fixed regular expression '.*AAB':

```
def findAAB(): NIO = {
  guard(Set('A')) {
    guard(Set('A')) {
      guard(Set('B'), Found)) {
        stop()
  }}} ++
  guard(None) { findAAB() } // in parallel...
}
```

We can easily add combinators on top of the core abstractions that take care of producing matchers from textual regular expressions. However the point here is to demonstrate how the implementation works. The given matcher uses an API that models nondeterministic finite automata (NFA):

```
type NIO = List[Trans]    // state: many possible transitions
case class Trans(c: Set[Char], x: Flag, s: () => NIO)
def guard(cond: Set[Char], flag: Flag)(e: => NIO): NIO =
  List(Trans(cond, flag, () => e))
def stop(): NIO = Nil
```

An NFA state consists of a list of possible transitions (type `NIO`). Each transition may be guarded by a set of characters and it may have a flag to be signaled if the transition is taken. It also knows how to compute the following state. We use `Chars` for simplicity, but of course we could use generic types as well. From a given NFA transition, we can obtain the following state, and thus recursively unfold the state space. Thus, it is sometimes useful to identify the notions of NFA and NFA state. NFA states, and thus NFAs, can be composed in parallel through list concatenation. Method `guard` implements sequential composition. It creates a new NFA state with a single start transition and "the rest of the automaton" given as a by-name expression (type `=>NIO`) that computes the following state. Note that the API does not mention where input is obtained from (files, streams, etc.).

8.2 From NFA to DFA Using Staging

We will translate NFAs to DFAs using LMS. This is the unstaged DFA API that will be used by the generated code:

```
abstract class DfaState {
  def hasFlag(x: Flag): Boolean
  def next(c: Char): DfaState
}
def dfaFlagged(flag: Flag, link: DfaState) = new DfaState {
  def hasFlag(x: Flag) = x == flag || link.hasFlag(x)
  def next(c: Char) = link.next(c)
}
def dfaState(f: Char => DfaState) = new DfaState {
  def hasFlag(x: Flag) = false
  def next(c: Char) = f(c)
}
```

The staged API is just a thin wrapper:

```
type DIO = Rep[DfaState]
def dfa_flag(x: Flag)(link: DIO): DIO
def dfa_trans(f: Rep[Char] => DIO): DIO
```

Translating an NFA to a DFA is accomplished by creating a DFA state for each encountered NFA configuration (removing duplicate states via `canonicalize`):

```
def convertNFAtoDFA(state: NIO): DIO = {
  val cstate = canonicalize(state)
  dfa_trans { c: Rep[Char] =>
    exploreNFA(cstate, c)(dfa_flag) { next =>
      convertNFAtoDFA(next)
    }
  }
}
convertNFAtoDFA(findAAB())
```

Since LMS memoizes functions in the same was as we have shown in Sect. 6 (here, the argument to `dfa_trans`), the framework ensures termination if the NFA is indeed finite. We use a separate function to explore the NFA space (see below), advancing the automaton by a symbolic character `cin` to invoke its continuations `k` with a new automaton, i.e. the possible set of states after consuming `cin`. The given implementation assumes that character sets contain either zero or one characters, the empty set `Set()` denoting a wildcard match. More elaborate cases such as character ranges are easy to add. The algorithm tries to remove as many redundant checks and impossible branches as possible. This only works because the character guards are staging-time values.

```
def exploreNFA[A](xs: NIO, cin: Rep[Char])(flag: Flag => Rep[A] => Rep[A])
                                           (k: NIO => Rep[A]):Rep[A] = xs match {
  case Nil => k(Nil)
  case Trans(Set(c), e, s)::rest =>
    if (cin == c) {
      // found match: drop transitions that look for other chars and
      // remove redundant checks
      val xs1 = rest collect { case Trans(Set(`c`)|None,e,s) => Trans(Set(),e,s) }
      val maybeFlag = e map flag getOrElse (x=>x)
      maybeFlag(exploreNFA(xs1, cin)(acc => k(acc ++ s())))
    } else {
      // no match, drop transitions that look for same char
      val xs1 = rest filter { case Trans(Set(`c`),_,_) => false case _ => true }
      exploreNFA(xs1, cin)(k)
    }
  case Trans(Set(), e, s)::rest =>
    val maybeFlag = e map flag getOrElse (x=>x)
    maybeFlag(exploreNFA(rest, cin)(acc => k(acc ++ s())))
}
```

8.3 Generated State Machine Code

The generated matcher code for regular expression `.*AAB` is shown below, with some slight syntactic changes to make it more readable. Each function corresponds to one DFA state. Note how negative information has been used to prune the transition space: given input such as `...AAB` the automaton jumps back to the initial state, i.e. it recognizes that the last character B cannot also be A, and it starts looking for two As after the B.

```
def stagedFindAAB(): DfaState = {
  val found = dfaFlagged(Found, matched_nothing)
  val matched_AA = dfaState { c: (Char) =>
    if (c == B) found
    else if (c == A) matched_AA
    else matched_nothing
  }
  val matched_A = dfaState { c: (Char) =>
    if (c == A) matched_AA
    else matched_nothing
  }
  val matched_nothing = dfaState { c: (Char) =>
    if (c == A) matched_A
    else matched_nothing
  }
}
```

9 Related Work

Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [65] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [9] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [54] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code. LMS draws inspiration from earlier work such as TaskGraph [4], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [7,38,55,56,63].

Partial Evaluation. Partial evaluation [32] is an automatic program specialization technique. Despite their automatic nature, most partial evaluators also provide annotations to guide specialization decisions. Some notable systems include DyC [26], an annotation-directed specializer for C, JSpec/Tempo [57], the JSC Java Supercompiler [34], and Civet [58].

Partial evaluation has addressed higher-order languages with state using similar let-insertion techniques as discussed here [6,28,37,67]. Further work has studied partially static structures [43] and partially static operations [66], and compilation based on combinations of partial evaluation, staging and abstract interpretation [16,33,60]. Two-level languages are frequently used as a basis for describing binding-time annotated programs [32,47].

Embedded DSLs. Embedded languages have a long history [36]. Hudak introduced the concept of embedding DSLs as pure libraries [30,31]. Steele proposed the idea of "growing" a language [62]. The concept of linguistic reuse goes back to Krishnamurthi [35]; Language virtualization to Chafi et al. [12]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [10] and Hofer et al. [29], going back to much earlier work by Reynolds [52]. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [39] and Elliot et al. [20]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [25,64], examples being Accelerate [42], Feldspar [3], Nikola [41]. Recent work presents new approaches around quotation and normalization for DSLs [13,45]. Other performance oriented DSLs include Firepile [48] (Scala), Terra [18,19] (Lua). Copperhead [11] (Python). Rackets macros [68] provide full control over the syntax and semantics.

Program Generators. A number of high-performance program generators have been built, for example ATLAS [70] (linear algebra), FFTW [24] (discrete fourier transform), and Spiral [49] (general linear transformations). Other systems include PetaBricks [2], CVXgen [27] and Halide [50,51].

References

1. Ager, M.S., Danvy, O., Rohde, H.K.: Fast partial evaluation of pattern matching in strings. ACM Trans. Program. Lang. Syst. **28**(4), 696–714 (2006)
2. Amarasinghe, S.P.: Petabricks: a language and compiler based on autotuning. In: Katevenis, M., Martonosi, M., Kozyrakis, C., Temam, O. (eds.) Proceedings of the 6th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC 2011, Heraklion, Crete, Greece, 24–26 January 2011, p. 3. ACM (2011)
3. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar: an embedded language for digital signal processing. In: Hage, J., Morazán, M.T. (eds.) IFL. LNCS, vol. 6647, pp. 121–136. Springer, Heidelberg (2011)
4. Beckmann, O., Houghton, A., Mellor, M.R., Kelly, P.H.J.: Runtime code generation in C++ as a foundation for domain-specific optimisation. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 291–306. Springer, Heidelberg (2004)
5. Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013. ACM (2013)
6. Bondorf, A.: Self-applicable partial evaluation. Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen (1990)
7. Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: PACT (2011)
8. Bulyonkov, M.A.: Polyvariant mixed computation for analyzer programs. Acta Inf. **21**, 473–484 (1984)
9. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Pfenning, F., Macko, M. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 57–76. Springer, Heidelberg (2003)

10. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters forsimpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)
11. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 47–56. ACM, New York (2011)
12. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: Onward! (2010)
13. Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA, 25–27 September 2013, pp. 403–416. ACM (2013)
14. Click, C., Cooper, K.D.: Combining analyses, combining optimizations. ACM Trans. Program. Lang. Syst. **17**, 181–196 (1995)
15. Consel, C., Danvy, O.: Partial evaluation of pattern matching in strings. Inf. Process. Lett. **30**(2), 79–86 (1989)
16. Consel, C., Khoo, S.-C.: Parameterized partial evaluation. ACM Trans. Program. Lang. Syst. **15**(3), 463–493 (1993)
17. Danvy, O.: Type-directed partial evaluation. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 367–411. Springer, Heidelberg (1999)
18. DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: a multi-stage language for high-performance computing. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 105–116. ACM (2013)
19. DeVito, Z., Ritchie, D., Fisher, M., Aiken, A., Hanrahan, P.: First-class runtime generation of high-performance types using exotypes. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, p. 11. ACM (2014)
20. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. J. Funct. Program. **13**(3), 455–481 (2003)
21. Filinski, A.: Representing monads. In: Boehm, H., Lang, B., Yellin, D.M. (eds.) 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Conference Record of POPL 1994, Portland, Oregon, USA, 17–21 January 1994, pp. 446–457. ACM Press (1994)
22. Filinski, A.: Normalization by evaluation for the computational Lambda-Calculus. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 151–165. Springer, Heidelberg (2001)
23. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, 23–25 June 1993, pp. 237–247. ACM (1993)
24. Frigo, M.: A fast fourier transform compiler. In: PLDI, pp. 169–180 (1999)
25. Gill, A.: Domain-specific languages and code synthesis using haskell. Queue **12**(4), 30:30–30:43 (2014)
26. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: Dyc: an expressive annotation-directed dynamic compiler for c. Theoret. Comput. Sci. **248**(1–2), 147–199 (2000)

27. Hanger, M., Johansen, T.A., Mykland, G.K., Skullestad, A.: Dynamic model predictive control allocation using CVXGEN. In: 9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, 19–21 December 2011, pp. 417–422. IEEE (2011)
28. Hatcliff, J., Danvy, O.: A computational formalization for partial evaluation. Math. Struct. Comput. Sci. **7**(5), 507–541 (1997)
29. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: Smaragdakis, Y., Siek, J.G. (eds.) GPCE, pp. 137–148. ACM (2008)
30. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. **28**, 196 (1996)
31. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse, pp. 134–142, June 1998
32. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall Inc., Upper Saddle River (1993)
33. Kiselyov, O., Swadi, K.N., Taha, W.: A methodology for generating verified combinatorial circuits. In: Buttazzo, G.C. (ed.) EMSOFT, pp. 249–258. ACM (2004)
34. Klimov, A.V.: A Java supercompiler and its application to verification of cache-coherence protocols. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 185–192. Springer, Heidelberg (2010)
35. Krishnamurthi, S.: Linguistic reuse. Ph.D. thesis, Computer Science, Rice University, Houston (2001)
36. Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3), 157–166 (1966)
37. Lawall, J.L., Thiemann, P.: Sound specialization in the presence of computational effects. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 165–190. Springer, Heidelberg (1997)
38. Lee, H., Brown, K.J., Sujeeth, A.K., Chafi, H., Rompf, T., Odersky, M., Olukotun, K.: Implementing domain-specific languages for heterogeneous parallel computing. IEEE Micro **31**(5), 42–53 (2011)
39. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: DSL, pp. 109–122 (1999)
40. Lindley, S.: Normalisation by evaluation in the compilation of typed functional programming languages (2005)
41. Mainland, G., Morrisett, G.: Nikola: embedding compiled GPU functions in Haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell 2010, pp. 67–78. ACM, New York (2010)
42. McDonell, T.L., Chakravarty, M.M., Keller, G., Lippmeier, B.: Optimising purely functional GPU programs. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 49–60. ACM, New York (2013)
43. Mogensen, T.A.: Partially static structures in a self-applicable partial evaluator (1988)
44. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
45. Najd, S., Lindley, S., Svenningsson, J., Wadler, P.: Everything old is new again: quoted domain specific languages. Technical report, University of Edinburgh (2015)
46. Nielson, F., Nielson, H.R.: Code generation from two-level denotational meta-languages. In: Ganzinger, H., Jones, N.D. (eds.) Programs as Data Objects. LNCS, vol. 217, pp. 192–205. Springer, Heidelberg (1986)
47. Nielson, F., Nielson, H.R.: Multi-level Lambda-Calculi: an algebraic description. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation. LNCS, vol. 1110, pp. 338–354. Springer, Heidelberg (1996)

48. Nystrom, N., White, D., Das, K.: Firepile: run-time compilation for GPUs in scala. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, GPCE, pp. 107–116. ACM, NewYork (2011)
49. Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J., Padua, D.A., Veloso, M.M., Johnson, R.W.: Spiral: a generator for platform-adapted libraries of signal processing alogorithms. IJHPCA **18**(1), 21–45 (2004)
50. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S.P., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. **31**(4), 32 (2012)
51. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.P.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 519–530. ACM (2013)
52. Reynolds, J.: User-defined types and procedural data structures as complementary approaches to data abstraction (1975)
53. Rompf, T.: Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming. Ph.D. thesis, EPFL (2012)
54. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. Commun. ACM **55**(6), 121–130 (2012)
55. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing data structures in high-level programs. In: POPL (2013)
56. Rompf, T., Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Odersky, M., Olukotun, K.: Building-blocks for performance oriented DSLs. In: DSL (2011)
57. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for java. ACM Trans. Program. Lang. Syst. **25**(4), 452–499 (2003)
58. Shali, A., Cook, W.R.: Hybrid partial evaluation. In: OOPSLA, pp. 375–390 (2011)
59. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. J. Funct. Program. **6**(6), 811–838 (1996)
60. Sperber, M., Thiemann, P.: Realistic compilation by partial evaluation. In: PLDI, pp. 206–214 (1996)
61. Sperber, M., Thiemann, P.: Generation of LR parsers by partial evaluation. ACM Trans. Program. Lang. Syst. **22**(2), 224–264 (2000)
62. Steele, G.: Growing a language. High.-Order Symbolic Comput. **12**(3), 221–236 (1999)
63. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and reuse with compiled domain-specific languages. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 52–78. Springer, Heidelberg (2013)
64. Svensson, B.J., Sheeran, M., Newton, R.: Design exploration through code-generating DSLs. Queue **12**(4), 40:40–40:52 (2014)
65. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoret. Comput. Sci. **248**(1–2), 211–242 (2000)
66. Thiemann, P.: Partially static operations. In: Albert, E., Mu, S.-C. (eds.) PEPM, pp. 75–76. ACM (2013)
67. Thiemann, P., Dussart, D.: Partial evaluation for higher-order languages with state. Technical report (1999)

68. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 132–141. ACM, New York (2011)
69. Turchin, V.F.: The concept of a supercompiler. ACM Trans. Program. Lang. Syst. **8**(3), 292–325 (1986)
70. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001)

1ML with Special Effects
F-ing Generativity Polymorphism

Andreas Rossberg(✉)

Google, München, Germany
rossberg@mpi-sws.org

Abstract. We take another look at 1ML, a language in the ML tradition, but with core and modules merged into one unified language, rendering all modules first-class values. 1ML already comes with a simple form of effect system that distinguishes pure from impure computations. Now we enrich it with effect polymorphism: by introducing effect declarations and, more interestingly, abstract effect specifications, effects can be parameterised over, and treated as abstract or concrete in the type system, very much like types themselves. Because type generativity in 1ML is controlled by (im)purity effects, this yields a somewhat exotic novel notion of *generativity polymorphism* – that is, a given functor can be asked to behave as either "generative" or "applicative". And this time, we even get to define an interesting (poly)monad for that!

1 Introduction

In a recent paper [13] we introduced 1ML, a reboot of ML where modules are first-class values. In this language, there no longer is any distinction between core and modules, records or structures, functions or functors. Morally, every expression denotes a module.

One peculiar feature of 1ML is its distinction between *pure* and *impure* functions and computations, through a – very simple – form of effect system. Although this is primarily motivated by the semantics of functors (as we will see below), it obviously is an interesting distinction to make for "core-like" functions as well. Effect systems [3,9,18] are regularly proposed as a complement to functional type systems. They serve a similar role as monads [12,20], which have been successfully incorporated into languages like Haskell, but with a little extra flexibility – Wadler & Thiemann showed that both are essentially equivalent [21].

Telling pure from impure statically allows much better control over side effects. Like any other form of type discipline, it allows more accurate specification of interfaces, and prevents certain classes of errors. For example, a module can specify that some of its interface functions are free of side effects. Or not. Higher-order functions can shield against unwanted side effects when invoking an argument function (or "callback", as laymen say nowadays). All useful capabilities, and in principle it is possible to put them to good use in 1ML already.

What makes the use of effects cumbersome in 1ML, however, is the lack of support for *effect polymorphism*. In a higher-order language that is a real

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 336–355, 2016.
DOI: 10.1007/978-3-319-30936-1_18

bummer, because the purity of a higher-order function will typically depend on the purity of its argument. For example, consider

```
map f xs = if null xs then [] else f (head xs) :: map f (tail xs)
```

Is this a pure or impure function? It depends on f. We can imagine (at least) two different signatures for map:

```
map : ∀ a b. (a →P b) →P list a →P list b
map : ∀ a b. (a →I b) →P list a →I list b
```

Here, we use "$\rightarrow_\mathsf{P}$" to denote pure function types, and "$\rightarrow_\mathsf{I}$" for impure ones. Unfortunately however, map may only have one of these types in 1ML. If we needed both, we would need to write different functions for the different signatures. (Notably, 1ML's pure arrows are subtypes of impure ones, but such subtype polymorphism is insufficient to handle this case.)

Or we introduce the ability to abstract over (im)purity via polymorphism. Both map and its argument could be polymorphic in their (joint) effect:

```
map : ∀ a b e. (a →e b) →P list a →e list b
```

In this signature, we assume that e is a (universally quantified) *effect variable*. By instantiating it with either P or I, we can use the same implementation of map on either a pure or an impure argument, and get a corresponding pure or impure computation in return.

Okay, you might think, that looks like a fairly trivial and standard extension. Others proposed this long ago [9]. Well, what makes it interesting in 1ML is that the (im)purity effect is intimately tied to the notion of *type generativity*.

Functions that return abstract types – "functors" in ML speak – are interesting beasts. There are two traditional schools of semantics: "*generative*" functors (the SML way) and "*applicative*" functors (the OCaml way). In 1ML, both notions coexist, and depend on whether a functor is pure or impure. Consider:

```
F : (a : type) →P type
```

A pure functor like this is "applicative" [8,15], meaning that all applications to equivalent types yield equivalent types:

```
type t = F bool
type u = F bool
```

defines types t and u to be equivalent. (Regardless, they are still abstract!)

An impure functor is "generative", however:

```
G : (a : type) →I type
```

generates a fresh type with every application. So,

```
type v = G bool
type w = G bool
```

are different types! This is important in the face of impurity, since the types produced by an impure computation may e.g. depend on state, and thus equating them would be plain unsound. (More reasons why you all want both applicative *and* generative functors are disclosed in Sects. 7 and 8 of [15].)

Now, what would happen if we allowed such a functor to be polymorphic over its effect? Say,

$$\mathsf{H} : (\mathsf{a} : \textbf{type}) \rightarrow_{\mathsf{P}} (\mathsf{e} : \textbf{effect}) \rightarrow_{\mathsf{e}} \textbf{type}$$

Depending on the choice of e, it would be either applicative or generative: that is, $\mathsf{H\ bool\ P} = \mathsf{H\ bool\ P}$, while $\mathsf{H\ bool\ I}$ is not reflexive (or even a well-formed type expression).

In other words, introducing polymorphism over effects implies a notion of *generativity polymorphism*. Is that actually a thing?

It is! In this paper, we show that this – rather esoteric – notion can actually be defined. And we don't even have to leave the familiar grounds of System F_ω for that. Well, except for one small extension. We also show that generativity polymorphism is the essence of MacQueen & Tofte & Kuan's (arguably equally esoteric) notion of "true" higher-order functors [6,7,10].

2 The Language

Figure 1 presents the syntax of 1ML extended with explicit effects, separated into core syntax and various syntactic sugar. For space reasons, we focus on the explicitly typed fragment $1\mathrm{ML}_{\mathrm{ex}}$ [13] here (though complementing this with effect inference in full 1ML is an easy addition). The effect-related constructs that are new relative to the original formulation of $1\mathrm{ML}_{\mathrm{ex}}$ are highlighted in the figure.

As in the original paper, the syntax is normalised to require named subterms in most constructs. The general forms, as well as many other familiar module or core-level constructs, can easily be defined as syntactic sugar. See [13] for enough sugar to upset your stomach.

2.1 Basic Use of Effects

The extended language incorporates three main additions.

Effect Annotations. First, function types acquire an explicit annotation F, that specifies the effect that is released when calling the function. Because real programming language syntax better works as plain text, we use the notation "F/T" instead of making F a subscript on the arrow, as we did in the introduction.

There are two basic effect constants: **pure** and **impure**. To ease notation in the common cases, we allow abbreviating pure function types with a plain arrow "$\rightarrow$", and impure ones with a squiggly arrow "$\rightsquigarrow$".[1]

[1] I apologise for any confusion this may cause with the original 1ML paper, where "$\rightarrow$" is spelled "$\Rightarrow$" and "$\rightsquigarrow$" is spelled "$\rightarrow$". I chose to change the syntax for the extended system because pure arrows are the more common case now, and I like them to be represented by their natural operator.

$$
\begin{array}{lrll}
\text{(identifiers)} & X & & \\
\text{(types)} & T & ::= & E \mid \mathbf{bool} \mid \{D\} \mid (X{:}T){\to}F/T \mid \mathbf{type} \mid \mathbf{effect} \mid {=}E \mid T\ \mathbf{where}\ (.\overline{X}{:}T) \\
\text{(effects)} & F & ::= & E \mid \mathbf{pure} \mid \mathbf{impure} \mid F{,}F \\
\text{(declarations)} & D & ::= & X : T \mid \mathbf{include}\ T \mid D;D \mid \epsilon \\
\text{(expressions)} & E & ::= & X \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ X\ \mathbf{then}\ E\ \mathbf{else}\ E{:}T \mid \{B\} \mid E.X \mid \\
 & & & \mathbf{fun}\ (X{:}T){\Rightarrow}E \mid X\ X \mid \mathbf{type}\ T \mid \mathbf{effect}\ F \mid X{:>}T \\
\text{(bindings)} & B & ::= & X{=}E \mid \mathbf{include}\ E \mid B;B \mid \epsilon
\end{array}
$$

Abbreviations:

$$
\begin{array}{lll}
\text{(types)} & & \\
(X{:}T_1) \to T_2 & := & (X{:}T_1) \to \mathbf{pure}/T_2 \\
(X{:}T_1) \leadsto T_2 & := & (X{:}T_1) \to \mathbf{impure}/T_2 \\
T_1 \overset{\to}{\leadsto} T_2 & := & (X{:}T_1) \overset{\to}{\leadsto} T_2
\end{array}
$$

where: (parameter) $P ::= (X{:}T)$

$$
\begin{array}{lll}
\text{(declarations)} & & \\
\mathbf{effect}\ X\ \overline{P} & := & X : \overline{P \to}\ \mathbf{effect} \\
\mathbf{effect}\ X\ \overline{P}{=}F & := & X : \overline{P \to}\ ({=}\ \mathbf{effect}\ F) \\
\text{(bindings)} & & \\
\mathbf{effect}\ X\ \overline{P}{=}F & := & X = \mathbf{fun}\ \overline{P} \Rightarrow \mathbf{effect}\ F
\end{array}
$$

Fig. 1. Syntax of $1\mathrm{ML}_{\mathrm{ex}}$ with effect polymorphism (see [13] for more abbreviations)

For example, the type of the polymorphic identity function,

$$\mathsf{id} = \mathbf{fun}\ \mathsf{a} \Rightarrow \mathbf{fun}\ (\mathsf{x} : \mathsf{a}) \Rightarrow \mathsf{a}$$

can be denoted as

$$\mathsf{id} : (\mathsf{a} : \mathbf{type}) \to \mathbf{pure}/(\mathsf{x} : \mathsf{a}) \to \mathbf{pure}/\mathsf{a}$$

or shorter, as just

$$\mathsf{id} : (\mathsf{a} : \mathbf{type}) \to (\mathsf{x} : \mathsf{a}) \to \mathsf{a}$$

Similarly, we can add impure operators to the language; for example, an ML-style type $\mathsf{ref}\ T$ of mutable references with operators

$$
\begin{array}{l}
\mathsf{new} : (\mathsf{a} : \mathbf{type}) \to (\mathsf{x} : \mathsf{a}) \leadsto \mathsf{ref\ a} \\
\mathsf{rd}\ \ : (\mathsf{a} : \mathbf{type}) \to (\mathsf{r} : \mathsf{ref\ a}) \leadsto \mathsf{a} \\
\mathsf{wr} : (\mathsf{a} : \mathbf{type}) \to (\mathsf{r} : \mathsf{ref\ a}) \to \mathsf{a} \leadsto \{\}
\end{array}
$$

Note how these types mix pure and impure arrows. A type parameter – expressing (explicit) polymorphism – is best considered a pure function: "instantiating" a polymorphic type should not have any effect. Similarly, an impure function with curried value parameters (like wr) releases its effect only when the last argument is provided.

Effect Values. Second, what makes the new syntax for function types interesting instead of just verbose is that effects can now be "computed": F can not just refer to the two effect constants, it can also consist of an expression E. This has to be a (pure) expression of the new type **effect**, a type that is inhabited by effect "values". Those values are formed by the corresponding expression **effect** F. Just like types in 1ML, effects can thus be named or passed around as if they were first-class values.

For example, we can write an effect-polymorphic apply combinator:

```
apply = fun (a : type) (b : type) (e : effect) (f : a → e/b) (x : a) ⇒ f x
```

The type of this function is:

```
apply : (a : type) → (b : type) → (e : effect) → (a → e/b) → a → e/b
```

To apply it, we have to provide the right effect argument:

```
t = apply bool bool pure id true
f = apply (ref bool) bool impure (rd bool) (new false)
```

Similarly, the map function from the introduction can be given the (explicitly) polymorphic type

```
map : (a : type) → (b : type) → (e : effect) → (a → e/b) → list a → e/list b
```

Like any other value, effects can also be named by a binding:

```
effect e = pure
t = apply bool bool e id true
```

Analogous to type bindings in 1ML, this effect binding is just sugar for the value binding "e = (**effect** pure)". Admittedly, such bindings are a bit boring with just effect constants, but they become more interesting with the following feature.

Effect Composition. Finally, how are we going to type the following function?

```
compose = fun (a : type) (b : type) (c : type) (e1 : effect) (e2 : effect)
                    (f : b → e2/c) (g : a → e1/b) (x : a) ⇒ f (g x)
```

To give a type to that, we need the ability to compose effects. That is the role of the effect operator ",":

```
compose : (a : type) → (b : type) → (c : type) → (e1 : effect) → (e2 : effect) →
                    (b → e2/c) → (a → e1/b) → a → (e1,e2)/c
```

"F_1,F_2" denotes the least upper bound of effects F_1 and F_2 in a lattice where **pure** is the bottom element and **impure** is top (the direction of this lattice is consistent with the natural notion of subtyping on effects, where pure $\leq$ impure; see Sect. 3). Consequently, an invocation of compose will be impure if at least one of the effect parameters is instantiated with impure; only if both are pure the result is pure as well. That should come as no surprise.

2.2 Generativity Polymorphism

So far, we have only looked at effects of simple functions. If you are already familiar with effect polymorphism, and much of the above looked boring, then you can wake up now – we are now turning to modules and functors.

First-Order Generativity Polymorphism. Let us start with the simplest possible functor – essentially, the identity type constructor:

```
Id = fun (a : type) ⇒ a
```

The most specific type derivable for Id in 1ML is the fully transparent type

```
type ID_TRANS = (a : type) → (= a)
```

where the *singleton type* (= a) indicates that the function returns the argument a itself. This *transparent* signature is a subtype of two possible *opaque* signatures that do not reveal the identity of their resulting type:

```
type ID_PURE = (a : type) → type
type ID_IMPURE = (a : type) ⇝ type
```

We can *seal* Id with either of these signatures:

```
Id_pure = Id :> ID_PURE
Id_impure = Id :> ID_IMPURE
```

Both these functors now return an abstract type, but the first is "applicative", i.e., always returns the same type for equivalent arguments, while the second is "generative", i.e., always returns a fresh type, regardless of the argument.

But in our extended 1ML, there now is a third choice:

```
type ID_POLY (e : effect) = (a : type) → e/type
Id_poly (e : effect) = Id :> ID_POLY e
```

We have just created our first "poly-generative" functor! This one can be applied to our choice of effect: for example, Id_poly pure bool = Id_poly pure bool are equivalent types, while Id_poly impure bool is a generative (and thus impure) expression that cannot be used as a type without binding it to a name. This is exactly the functor H we wondered about in the introduction.

Higher-Order Generativity Polymorphism. Okay, that was a contrived example. Generativity polymorphism becomes more relevant in the higher-order case. Because we can now write the kind of functors that MacQueen always wanted to write, under his slogan of "true higher-order functors" [6,7,10]. Recast in 1ML, the problem he is concerned with, as formulated by Kuan & MacQueen [7], boils down to the ability to define a generic Apply functor over types:

```
Apply (F : type → e/type) (a : type) = F a
```

Kuan & MacQueen want to be able to use such a functor transparently (i.e., such that type identities are fully propagated) in both of the following cases:

```
Id (a : type) = a
t = Apply Id bool
f (x : t) = (x : bool) ;; type-checks because t = bool


Const (a : type) = int
u = Apply Const bool
g (x : u) = (x : int) ;; type-checks because u = int
```

As it turns out, these examples already type-check in plain 1ML: both applications are well-typed, provided one picks e = pure in the parameter type of F when defining Apply (which is equivalent to a plain 1ML pure function type). In fact, even the application to an abstract functor works, as long as that is pure as well:

```
Abs = Id :> type → type
v = Apply Abs bool
h (x : v) = (x : Abs bool) ;; type-checks because v = Abs bool
```

However, Kuan & MacQueen didn't deal with a language of (only) applicative functors (and neither do we), so the above isn't quite a fair answer to their challenge. What they really meant in particular is that Apply should also be applicable to a *generative* functor, like here:

```
Gen (a : type) = a :> type
w = Apply Gen bool
```

That also works in plain 1ML, but only if the parameter F is typed as an impure functor in the definition of Apply, equivalent to picking e = impure in our extended language. That is, in plain 1ML, we can write Apply such that either the former three examples type-check, or the last, but not all at the same time.

Effect polymorphism to the rescue! You already guessed it: in extended 1ML we can escape that dilemma by making e into a parameter itself:

```
Apply (e : effect) (F : type → e/type) (a : type) = F a
```

Now all examples are expressible:

```
t = Apply pure Id bool
u = Apply pure Const bool
v = Apply pure Abs bool
w = Apply impure Gen bool
```

The extra argument is a bit more tedious to write than Kuan & MacQueen would want, but it could easily be inferred (though we don't discuss that here).

Existential Effect Polymorphism. Just for completeness, we mention that effects can also – like types – be *sealed*, introducing the notion of an "abstract effect":

```
M = {effect e = pure; f (g : int → e/bool) = ...} :> {effect e; f : (int → e/bool) →...}
let g (h : int → M.e/bool) = ... in g M.f
```

Honestly, this does not look like it would be a particularly useful feature, but it falls out from 1ML's design naturally and for free. Ruling it out would be more complicated than allowing it. Maybe there even is some crazy use case that we don't foresee yet... Phantom effects, anyone?

3 Type System

Here's how we build a type system for 1ML$_{\text{ex}}$ with generativity polymorphism. (Unfortunately, for the lack of space, we have to focus on the novelties, and refer the interested reader to [13] for many basic details and rules omitted here.)

Semantic Types. Following the F-ing modules approach [15], 1ML's type system [13] is not defined in terms of its own syntactic types. Instead, it is defined by translating those types into *semantic types*. (The short story behind that approach is that syntactic module types are not expressive enough to accurately account for all details of type abstraction and functorisation, especially the problem of local types, a.k.a. the *avoidance problem*. See [15] for the long story.)

$$\begin{array}{lll}
\text{(computation)} & \varPhi & ::= \varXi \mid (\varPhi + \varPhi)^{\eta} \\
\text{(abstracted)} & \varXi & ::= \exists \overline{\alpha}.\varSigma \\
\text{(large)} & \varSigma & ::= \pi \mid \mathsf{bool} \mid [= \varXi] \mid [= \eta] \mid \{\overline{l{:}\varSigma}\} \mid \forall \overline{\alpha}.\varSigma \rightarrow_{\eta} \varPhi \\
\text{(small)} & \sigma & ::= \pi \mid \mathsf{bool} \mid [= \sigma] \mid [= \eta] \mid \{\overline{l{:}\sigma}\} \mid \sigma \rightarrow_{\eta} \sigma \\
\text{(paths)} & \pi & ::= \alpha \mid \pi\, \overline{\sigma} \\
\text{(effects)} & \eta & ::= \mathtt{P} \mid \mathtt{I} \mid \iota \mid \eta \vee \eta
\end{array}$$

Notation:

$$\begin{array}{rlrlrll}
\mathtt{P} \vee \eta := \eta \vee \mathtt{P} & := \eta & \varXi\,! & := \varXi & (\varPhi_1 \oplus \varPhi_2)^{\mathtt{P}} & := \varPhi_1 & \\
\mathtt{I} \vee \eta := \eta \vee \mathtt{I} & := \mathtt{I} & (\varPhi_1 + \varPhi_2)^{\mathtt{P}}! & := \varPhi_1! & (\varPhi_1 \oplus \varPhi_2)^{\mathtt{I}} & := \varPhi_2 & \\
\eta(\varSigma) & := \mathtt{P} & & & (\varPhi_1 \oplus \varPhi_2)^{\eta} & := \varPhi_1 & \text{if } \varPhi_1 = \varPhi_2 \\
\eta(\exists \alpha \overline{\alpha}.\varSigma) & := \mathtt{I} & & & (\varPhi_1 \oplus \varPhi_2)^{\eta} & := (\varPhi_1 + \varPhi_2)^{\eta} & \text{otherwise}
\end{array}$$

Fig. 2. Semantic types

Figure 2 shows the grammar of semantic types needed for 1ML$_{\text{ex}}$ with effect polymorphism (plus some auxiliary notation we'll get to). They are written in the style of System F types with explicit quantifiers (and as we will see later they actually *are* System F types). Once more, additions to the original 1ML system are highlighted. Type variables can be higher-order in these types, but we assume they are kinded implicitly, and we use the notation κ_α if we need to talk about the kind of α.

Let us recap the main intuitions behind those F-ing module types. The central idea is introducing explicit quantifiers to remove all dependencies within a type.

Following Mitchell & Plotkin [11], signature types containing abstract types (i.e., ADTs) are represented as *existential* types. For example, the signature

$$\{\textbf{type}\ \mathsf{t};\ \textbf{type}\ \mathsf{u};\ \textbf{type}\ \mathsf{v} = \mathsf{t} \rightarrow \mathsf{t};\ \mathsf{f} : \mathsf{t} \rightarrow \mathsf{u}\}$$

where components v and f depend on t and u, corresponds to the semantic type

$$\exists \alpha_1 \alpha_2.\{\mathsf{t} : [= \alpha_1], \mathsf{u} : [= \alpha_2], \mathsf{v} : [= \alpha_1 \rightarrow \alpha_1], \mathsf{f} : \alpha_1 \rightarrow \alpha_2\}$$

where α_1 and α_2 are the local names for type components t and u, respectively, and there are no dependencies inside the record. The notation $[= \tau]$ denotes the type of τ reified "as a value", i.e., represents the type **type** (recall that a component specification "**type** t" is just short for "t: **type**" in 1ML).

Functors, on the other hand, correspond to *universal* types. Any abstract type from their domain signature becomes a universal type variable with scope widened to include the codomain. For example, the functor type

$$(\mathsf{X} : \{\textbf{type}\ \mathsf{t};\ \mathsf{v} : \mathsf{t}\}) \to \textbf{pure}/\{\textbf{type}\ \mathsf{u} = \mathsf{X.t} \to \mathsf{X.t};\ \mathsf{x} : \mathsf{X.t}\}$$

with dependencies from its codomain on the type X.t from the domain, maps to

$$\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_{\mathsf{P}} \{\mathsf{u} : [= \alpha \to \alpha], \mathsf{x} : \alpha\}$$

More interesting are cases where an abstract type is bound in the codomain:

$$(\mathsf{X} : \{\textbf{type}\ \mathsf{t};\ \mathsf{v} : \mathsf{t}\}) \to \mathsf{e}/\{\textbf{type}\ \mathsf{u};\ \mathsf{f} : \mathsf{X.t} \to \mathsf{u}\}$$

If e is impure, then this is a generative functor, which is modeled in the semantic types by returning an existential package:

$$\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_{\mathtt{I}} \exists\beta.\{\mathsf{u} : [= \beta], \mathsf{f} : \alpha \to \beta\}$$

If e is pure, however, then the functor is supposed to behave applicatively, i.e., return the same abstract type on each application. That is modeled by lifting the existential out of the function:

$$\exists\beta.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_{\mathsf{P}} \{\mathsf{u} : [= \beta\,\alpha], \mathsf{f} : \alpha \to \beta\,\alpha\}$$

In fact, it as an invariant of our semantic types that a pure arrow never has an existential quantifier to the right – a pure functor cannot generate types. There is one subtlety involved with the above type, though: in the implementation of a functor matching this signature, the definition of u may depend on the parameter type t. Following Biswas [1] and Russo [16], all uses of β in the semantic interpretation are hence skolemised over α accordingly. Consequently, β needs to have higher kind $\kappa_\beta = \Omega \to \Omega$ here (other variables had base kind Ω so far).

As we can see, the structure of the semantic type modeling the functor type above is quite different depending on the choice of e. It differs in terms of where the quantifier goes (inside vs. outside), its variable kind (Ω vs. $\Omega \to \Omega$), and how the abstract type u is denoted (β vs. $\beta\,\alpha$). How can we reconcile these structural differences to support a parametric choice of effect?

We can't. Not really, anyway. We need something new.

The natural trick is to use sums: we generalise function types such that their codomain is a "computation" type Φ that allows arbitrary many alternative results. That is, if e is not statically known to be either pure or impure, then the functor signature above will be represented as

$$\begin{aligned}&\exists\beta_1.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_\eta \\ &\qquad (\{\mathsf{u} : [= \beta_1\,\alpha], \mathsf{f} : \alpha \to \beta_1\,\alpha\} + \exists\beta_2.\{\mathsf{u} : [= \beta_2], \mathsf{f} : \alpha \to \beta_2\})^\eta\end{aligned}$$

This encodes both possibilities: the left side of the sum is for the pure choice, the right for the impure one. In this type, η is the representation of the effect e as a semantic type. If η is not a constant $\mathtt{P}$ or $\mathtt{I}$, then it either is an *effect variable* ι, or the least upper bound of several of those, represented by the "$\vee$" operator. Since the two effect constants are the top and bottom elements of the effect lattice, least upper bounds that contain those constants can always be simplified to either ones that don't, or directly to $\mathtt{I}$. We assume that this simplification is always performed implicitly, according to the notational rules given in Fig. 2.

The effect η appears twice in the above type: once as an annotation on the arrow, and once as an *index* on the sum, which is written $(\Phi_1 + \Phi_2)^\eta$. The former occurrence tracks the effect of the function, in a generalisation of what we already had in plain 1ML. The latter tracks the choice of the sum: if, at some point, the free effect variables of η get substituted such that the result normalises to an effect constant, then the choice is statically determined – this basically is a binary GADT. We say that a Φ with all effect indices being constants is *unique*. As we will see below, this static extra knowledge is important for figuring out when an expression of type **type** unambiguously denotes a type, such that it is legal to project it.

It may be surprising that we need η twice. The reason is that, in general, the computation type Φ in the codomain of an arrow can consist of many nested sums, indexed by different effects – e.g., when a functor itself invokes several other functors with variable effects. The effect on the arrow then only is an upper bound of all these individual indices (and not necessarily the least); for example, we could construct a functor of type $\Sigma \rightarrow_{\eta_1 \vee \eta_2} ((\Xi_1 + \Xi_2)^{\eta_1} + \Xi_3)^{\eta_2}$. In a pure function, however, this upper bound is $\mathtt{P}$, so all effect indices in Φ must be pure as well, such that Φ is already guaranteed to be unique.

Types and Effects. The new and modified rules for translating syntactic types T into semantic types Ξ are collected in Fig. 3 (please see [13] for the others). The individual modifications over plain 1ML are again highlighted.

The new rule TEFFECT for the type **effect** is analogous to the rule for **type**, in that it introduces a fresh type variable to name the abstract effect. We use ι to range over type variables that encode effects. We assume that these type variables are a subcategory of general type variables α, such that they can be uniformly written as α wherever we don't care about the distinction.

The rule TFUN for function types effectively merges the previous rules TFUN and TPFUN from 1ML, covering both impure and pure functors, but also effect-polymorphic ones. It refers to the simple effect elaboration judgement also shown in the figure. The auxiliary operator "$\oplus$" (defined in Fig. 2) avoids the sum in those cases where the effect is statically known or does not change the type. Likewise, we write "$\exists\overline{\alpha}^{\iota \neq \mathtt{I}}$" to say that a quantifier is to be empty if $\eta = \mathtt{I}$.

Another change is in rules TPATH and TSING (and inherited by FPATH): the notation $\Phi!$ (also defined in Fig. 2) requires E's type Φ to be unique – it selects Φ's unique summand and is undefined otherwise. It is here where we rely on the effect index on sums: only sums whose indices are constant are unique, and can be used for unambiguous projection of static information.

Types $\boxed{\Gamma \vdash T \rightsquigarrow \Xi}$

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \rightsquigarrow e \qquad \Phi! = [= \Xi]}{\Gamma \vdash E \rightsquigarrow \Xi}\,\text{TPATH}$$

$$\frac{\kappa_\alpha = \Omega}{\Gamma \vdash \mathsf{type} \rightsquigarrow \exists \alpha.[= \alpha]}\,\text{TTYPE} \qquad \frac{}{\Gamma \vdash \mathsf{effect} \rightsquigarrow \exists \iota.[= \iota]}\,\text{TEFFECT}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \\ \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2 \qquad \Gamma \vdash F \rightsquigarrow \eta \qquad \overline{\kappa_{\alpha'_2} = \overline{\kappa_{\alpha_1}} \rightarrow \kappa_{\alpha'_2}}\end{array}}{\Gamma \vdash (X{:}T_1) \rightarrow F/T_2 \rightsquigarrow \exists \overline{\alpha}'^{\,\eta \neq \mathtt{I}}_2.\forall \overline{\alpha}_1.\, \Sigma_1 \rightarrow_\eta ((\exists \overline{\alpha}_2.\Sigma_2) \oplus (\Sigma_2[\overline{\alpha'_2\,\overline{\alpha}_1}/\overline{\alpha}_2]))^\eta}\,\text{TFUN}$$

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \rightsquigarrow e \qquad \Phi! = \Sigma}{\Gamma \vdash (=E) \rightsquigarrow \Sigma}\,\text{TSING}$$

Effects $\boxed{\Gamma \vdash F \rightsquigarrow \eta}$

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \rightsquigarrow e \qquad \Phi! = [= \eta]}{\Gamma \vdash E \rightsquigarrow \eta}\,\text{FPATH}$$

$$\frac{}{\Gamma \vdash \mathsf{pure} \rightsquigarrow [= \mathtt{P}]}\,\text{FPURE} \qquad \frac{}{\Gamma \vdash \mathsf{impure} \rightsquigarrow [= \mathtt{I}]}\,\text{FIMPURE}$$

$$\frac{\Gamma \vdash F_1 \rightsquigarrow \eta_1 \qquad \Gamma \vdash F_2 \rightsquigarrow \eta_2}{\Gamma \vdash F_1{,}F_2 \rightsquigarrow \eta_1 \vee \eta_2}\,\text{FJOIN}$$

Fig. 3. Elaboration of types and effects (new and modified rules)

Expressions and Bindings. Figure 4 shows selected typing rules for expressions E and bindings B, novelties once more highlighted. Before we go into details, let us recap the main idea of typing modules using the F-ing modules approach.

As we saw before, the main trick in interpreting module types is introducing quantifiers for abstract types. That is reflected in the typing of expressions: an expression that defines new abstract types will have an "abstracted" type Ξ with existential quantifiers, one quantifier for each new type.

When such an expression is nested into a larger expression then the rules have to propagate these quantifiers accordingly. For example, for a projection $E.\mathsf{x}$ to be well-typed, E obviously needs to have a type of the form $\{\mathsf{x} : \Sigma, \dots\}$; the resulting type would be Σ then. However, if E creates abstract types locally, then its type will be of the form $\exists\overline{\alpha}.\{\mathsf{x} : \Sigma, \dots\}$ instead. The central idea of F-ing modules is to handle such types implicitly by extruding the existential quantifier automatically: that is, the projection $E.\mathsf{x}$ is well-typed and assigned type $\exists\overline{\alpha}.\Sigma$, with the same sequence of quantifiers. And so on for other constructs.

As we pointed out in [15], this handling of existential types is akin to a *monad* – the monad of type generation! More precisely, it is a stack of nested monads, one for each generated type. By extending 1ML with generativity polymorphism, however, there no longer necessarily is a unique quantifier sequence for a given expression. Expressions are now classified by "computation" types Φ, which are sums over heterogeneous existentials. Our monad just became more interesting!

Expressions $\boxed{\boldsymbol{\Gamma \vdash E :_\eta \Phi \rightsquigarrow e}}$

$$\frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \mathsf{\mathbf{type}}\ T :_{\mathrm{P}} [= \Xi] \rightsquigarrow [\Xi]}\,\textsc{EType} \qquad \frac{\Gamma \vdash F \rightsquigarrow \eta}{\Gamma \vdash \mathsf{\mathbf{effect}}\ F :_{\mathrm{P}} [= \eta] \rightsquigarrow [\eta]}\,\textsc{EEffect}$$

$$\frac{\Gamma \vdash E :_\eta M\lambda\overline{\alpha}.\{\overline{X':\Sigma'}\} \rightsquigarrow e \qquad X{:}\Sigma \in \overline{X':\Sigma'}}{\Gamma \vdash E.X :_\eta M\lambda\overline{\alpha}.\Sigma \rightsquigarrow \mathsf{do}_M\ \overline{\alpha}, y \leftarrow e\ \mathsf{in}\ y.X}\,\textsc{EDot}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash E :_\eta \Phi \rightsquigarrow e}{\Gamma \vdash \mathsf{\mathbf{fun}}\,(X{:}T) \Rightarrow E :_{\mathrm{P}} \forall\overline{\alpha}.\,\Sigma \to_\eta \Phi \rightsquigarrow \lambda\overline{\alpha}.\lambda_\eta X{:}\Sigma.e}\,\textsc{EFun}$$

$$\frac{\begin{array}{ll} \Gamma \vdash X_1 :_{\mathrm{P}} (\forall\overline{\alpha}.\,\Sigma_1 \to_\eta \Phi) \rightsquigarrow e_1 & \\ \Gamma \vdash X_2 :_{\mathrm{P}} \Sigma_2 \rightsquigarrow e_2 & \Gamma \vdash \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f \end{array}}{\Gamma \vdash X_1\,X_2 :_\eta \delta\Phi \rightsquigarrow (e_1\,(\delta\overline{\alpha})\,(f\,e_2)).\mathsf{val}}\,\textsc{EApp}$$

Bindings $\boxed{\boldsymbol{\Gamma \vdash B :_\eta \Phi \rightsquigarrow e}}$

$$\frac{\Gamma \vdash E :_\eta M\lambda\overline{\alpha}.\Sigma \rightsquigarrow e}{\Gamma \vdash X{=}E :_\eta M\lambda\overline{\alpha}.\{X{:}\Sigma\} \rightsquigarrow \mathsf{do}_M\ \overline{\alpha}, x \leftarrow e\ \mathsf{in}\ \{X{=}x\}}\,\textsc{BVar}$$

$$\frac{\begin{array}{rl} \Gamma \vdash B_1 :_{\eta_1} M_1\lambda\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \rightsquigarrow e_1 & \overline{X'_1} = \overline{X_1} - \overline{X_2} \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash B_2 :_{\eta_2} M_2\lambda\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \rightsquigarrow e_2 & \overline{X'_1{:}\Sigma'_1} \subseteq \overline{X_1{:}\Sigma_1} \end{array}}{\begin{array}{l} \Gamma \vdash B_1;B_2 :_{\eta_1\vee\eta_2} M_1M_2\lambda\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X'_1{:}\Sigma'_1}, \overline{X_2{:}\Sigma_2}\} \\ \quad \rightsquigarrow \mathsf{join}_{M_1M_2}\ \mathsf{do}_{M_1}\ \overline{\alpha}_1, y_1 \leftarrow e_1\ \mathsf{in\ let}\ \overline{X_1 = y_1.X_1}\ \mathsf{in} \\ \qquad\qquad\quad \mathsf{do}_{M_2}\ \overline{\alpha}_2, y_2 \leftarrow e_2\ \mathsf{in}\ \{\overline{X'_1 = y_1.X'_1}, \overline{X_2 = y_2.X_2}\} \end{array}}\,\textsc{BSeq}$$

Fig. 4. Elaboration of expressions (selected rules)

Fortunately, the sums we are dealing with are not arbitrary. Ultimately, they all originate, directly or indirectly, from uses of the rule TFUN introducing effect-polymorphic function types. And a quick look at this rule reveals that the inner structure of the type is the same in both cases, up to the presence of quantifiers and the internal naming of the abstract types introduced.

That allows to factor computation types Φ such that we separate their inner structure from their quantification scheme. To that end, Fig. 5 defines an auxiliary syntactic class of monadic type constructors M. Any computation type Φ can be expressed β-equivalently as an application of such an M to a suitable type constructor defining the inner structure of the result. For example, the effect-polymorphic functor type from earlier can be written equivalently as

$$\exists\beta_1.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_\eta (\exists[\beta_1\,\alpha] + \exists\beta_2[\beta_2])^\eta\,(\lambda\beta.\{\mathsf{u} : [= \beta], \mathsf{f} : \alpha \to \beta\})$$

That is, an application of $(M_1 + M_2)^\eta$ with $M_1 = \exists[\beta_1\,\alpha]$ (which has an empty existential quantifier) and $M_2 = \exists\beta_2[\beta_2]$ to the structural template $\lambda\beta.\{\mathsf{u} : [= \beta], \mathsf{f} : \alpha \to \beta\}$, with β being mapped to either $\beta_1\,\alpha$ or β_2, accordingly.

The set of all monadic types M forms a *polymonad* [5] or *productoid* [19], with $\exists[]$ as its identity element. We can define the composition M_1M_2 as given

$$\text{(monadic type)} \quad M ::= \exists\alpha[\pi] \mid (M+M)^{\eta}$$

with:

$$\begin{array}{lcl}
\exists\overline{\alpha}[\overline{\pi}]_{\overline{\kappa}} & := & \lambda c{:}(\overline{\kappa} \to \Omega).\exists\overline{\alpha}.c\,\overline{\pi} \\
(M_1+M_2)^{\eta}_{\overline{\kappa}} & := & \lambda c{:}(\overline{\kappa} \to \Omega).(M_1 c + M_2 c)^{\eta}
\end{array}$$

$$(M_1M_2)_{\overline{\kappa}_1\overline{\kappa}_2} := \begin{cases} \exists\overline{\alpha}_1\overline{\alpha}_2[\overline{\pi}_1\overline{\pi}_2] & \text{if } M_1 = \exists\overline{\alpha}_1[\overline{\pi}_1] \text{ and } M_2 = \exists\overline{\alpha}_2[\overline{\pi}_2] \\ (M_1M_{21} + M_1M_{22})^{\eta_2} & \text{if } M_1 = \exists\overline{\alpha}_1[\overline{\pi}_1] \text{ and } M_2 = (M_{21}+M_{22})^{\eta_2} \\ (M_{11}M_2 + M_{12}M_2)^{\eta_1} & \text{if } M_1 = (M_{11}+M_{12})^{\eta_1} \end{cases}$$

$$\mathsf{do}_M\ \overline{\alpha}, x \leftarrow e_1 \text{ in } e_2 := \begin{cases} \mathsf{unpack}\ \langle\overline{\alpha}', x\rangle = e_1\ \mathsf{in\ pack}\ \langle\overline{\alpha}', e_2\rangle & \text{if } M = \exists\overline{\alpha}'[\overline{\pi}] \\ \mathsf{let}\ f = \lambda\overline{\alpha}.\lambda x.e_2\ \mathsf{in\ case}\ e_1\ \mathsf{of} & \text{if } M = (M_1+M_2)^{\eta} \\ \quad \mathsf{inl}\ x_1.(\mathsf{do}_{M_1}\ \overline{\alpha}, x \leftarrow x_1\ \mathsf{in\ inl}\ e_2\,\overline{\alpha}\,x) \mid & \\ \quad \mathsf{inr}\ x_2.(\mathsf{do}_{M_2}\ \overline{\alpha}, x \leftarrow x_2\ \mathsf{in\ inr}\ e_2\,\overline{\alpha}\,x) & \end{cases}$$

$$\mathsf{join}_{M_1M_2}\ e := \begin{cases} \mathsf{case}\ e\ \mathsf{of} & \text{if } M_1 = (M_{11}+M_{12})^{\eta_1} \\ \quad \mathsf{inl}\ x.\mathsf{inl}\ (\mathsf{join}_{M_{11}M_2}\ x) \mid & \\ \quad \mathsf{inr}\ x.\mathsf{inr}\ (\mathsf{join}_{M_{12}M_2}\ x) & \\ \mathsf{unpack}\ \langle\overline{\alpha}_1, x\rangle = e\ \mathsf{in\ case}\ x\ \mathsf{of} & \text{if } M_1 = \exists\overline{\alpha}_1[\overline{\pi}_1] \\ \quad \mathsf{inl}\ y.\mathsf{inl}\ (\mathsf{join}_{M_1M_{21}}\ \mathsf{pack}\ \langle\overline{\alpha}_1, y\rangle) \mid & \text{and } M_2 = (M_{21}+M_{22})^{\eta_2} \\ \quad \mathsf{inr}\ y.\mathsf{inr}\ (\mathsf{join}_{M_1M_{22}}\ \mathsf{pack}\ \langle\overline{\alpha}_1, y\rangle) & \\ e & \text{otherwise} \end{cases}$$

Fig. 5. Polymonad notation for computation types

in Fig. 5. We won't go into details here, but leave proving the polymonad laws as an exercise (see also Sect. 4).

It suffices to observe that we can use this notation to uniformly access the structure of all alternatives of a computation type. Moreover, we can construct a new computation type that lives in the same monadic envelope M. In particular, this happens in rules EDOT and BVAR, which project and inject a value from/into a structure, respectively. The notation is put to more interesting use in rule BSEQ, where two nested monadic computations in M_1 and M_2 are lifted to their composition M_1M_2.

On the term level, the polymonad is witnessed by suitably defined do-notation (which expresses a mapping over some M) and a join operator. The definition of these operators, indexed by M, is given in Fig. 5.

Subtyping. The existing rules for 1ML subtyping don't change, but subtyping now needs to be generalised to computation types Φ. Figure 6 shows how.

Basically, the six new rules inductively express that $\Phi_1 \leq \Phi_2$ holds if each Ξ_1 from Φ_1 is a subtype of each Ξ_2 from Φ_2. Except that in the case of a constant effect index, the excluded alternative can be ignored.

The most interesting case is rule SR. It coerces a unique type Ξ' into a sum indexed by an effect η. Since η may force later which alternative to pick, the coercion has to perform a case distinction over η. To enable that, effects need

Subtyping $\quad \varGamma \vdash \varPhi' \leq \varPhi \rightsquigarrow f \; := \; \varGamma \vdash \varPhi' \leq_\epsilon \varPhi \rightsquigarrow \mathrm{id}; f \qquad \boxed{\boldsymbol{\varGamma \vdash \varPhi' \leq_\pi \varPhi \rightsquigarrow \delta; f}}$

$$\frac{\varGamma \vdash \varPhi'_1 \leq_\pi \varPhi \rightsquigarrow \delta; f_1 \qquad \varGamma \vdash \varPhi'_2 \leq_\pi \varPhi \rightsquigarrow \delta; f_2}{\varGamma \vdash (\varPhi'_1 + \varPhi'_2)^{\eta'} \leq_\pi \varPhi \rightsquigarrow \delta; \lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y.f_1 y \mid \mathsf{inr}\ y.f_2 y}\textsc{Sl}$$

$$\frac{\varGamma \vdash \varPhi'_1 \leq_\pi \varPhi \rightsquigarrow \delta; f}{\varGamma \vdash (\varPhi'_1 + \varPhi'_2)^{\mathtt{P}} \leq_\pi \varPhi \rightsquigarrow \delta; \lambda x.f\,(\mathsf{asl}\ x)}\textsc{Slp} \qquad \frac{\varGamma \vdash \varPhi'_2 \leq_\pi \varPhi \rightsquigarrow \delta; f}{\varGamma \vdash (\varPhi'_1 + \varPhi'_2)^{\mathtt{I}} \leq_\pi \varPhi \rightsquigarrow \delta; \lambda x.f\,(\mathsf{asr}\ x)}\textsc{Sli}$$

$$\frac{\varGamma \vdash \varPhi' \leq_\pi \varPhi_1 \rightsquigarrow \delta; f_1 \qquad \varGamma \vdash \varPhi' \leq_\pi \varPhi_2 \rightsquigarrow \delta; f_2}{\varGamma \vdash \varXi' \leq_\pi (\varPhi_1 + \varPhi_2)^{\eta} \rightsquigarrow \delta; \lambda x.\mathsf{case}\ \eta\ \mathsf{of}\ \mathsf{inl}\ y.\mathsf{inl}\ (f_1 x) \mid \mathsf{inr}\ y.\mathsf{inr}\ (f_2 x)}\textsc{Sr}$$

$$\frac{\varGamma \vdash \varPhi' \leq_\pi \varPhi_1 \rightsquigarrow \delta; f}{\varGamma \vdash \varXi' \leq_\pi (\varPhi_1 + \varPhi_2)^{\mathtt{P}} \rightsquigarrow \delta; \lambda x.\mathsf{inl}\ (f\ x)}\textsc{Srp} \qquad \frac{\varGamma \vdash \varPhi' \leq_\pi \varPhi_2 \rightsquigarrow \delta; f}{\varGamma \vdash \varXi' \leq_\pi (\varPhi_1 + \varPhi_2)^{\mathtt{I}} \rightsquigarrow \delta; \lambda x.\mathsf{inr}\ (f\ x)}\textsc{Sri}$$

$$\frac{}{\eta \leq \mathtt{I}}\textsc{Ftop} \qquad \frac{}{\mathtt{P} \leq \eta}\textsc{Fbot} \qquad \frac{\overline{\iota}' \supseteq \overline{\iota}}{\bigvee \overline{\iota}' \leq \bigvee \overline{\iota}}\textsc{Fjoin} \qquad \boxed{\boldsymbol{\eta' \leq \eta}}$$

Fig. 6. Elaboration of subtyping (new rules)

to be reified as terms in the elaboration. We refer to the Appendix for details. There, we also explain the operators asl and asr used in the elaboration of rules Slp and Sli, which are akin to a one-armed case over the binary "+" GADT.

Metatheory. For space reasons, we have banished all metatheory to the Appendix, where we define the encoding of semantic types into System F_ω, and state the obvious soundness results for the elaboration.

4 Related Work

There has been a broad range of work on effect systems and effect polymorphism, starting from Gifford & Lucassen's original work [3,9] and Talpin & Jouvelot's refinements [18]. But as noted in the introduction, the implications of effect polymorphism that we have investigated in this paper is rather esoteric – to the best of our knowledge, there is no other work on effect systems for modules, or generativity polymorphism of the kind we introduced here.

"True" Higher-Order Modules. The idea most closely related hence actually is MacQueen's notion of "true" higher-order modules, as originally introduced by MacQueen & Tofte [10], implemented in SML of New Jersey, and later recast by Kuan & MacQueen [6,7]. In this semantics, every functor type is implicitly "generativity polymorphic" as much as possible.

However, the formal details are rather involved, defining a specialised operational calculus of type name creation, path trees, and explicit environment manipulation (named the "entity calculus" in Kuan & MacQueen's more recent work). This semantics has so far escaped a more type-theoretic treatment, and consequently, none of the other formalisations of higher-order modules on the market [2,4,8,13–15,17] has followed its lead.

The system we presented is coming from a completely different angle. Yet, as we show in Sect. 2, it has similar expressiveness, while maintaining most of the relative simplicity of the 1ML semantics. One could argue that effect polymorphism is what was hiding in MacQueen's system all along, and that our system makes that explicit and gives it a foundation in standard type theory.

Monads, Polymonads and Productoids. Moggi [12] suggested monads as a means for semantic modeling of effectful computations. Wadler [20] recognised their broader value for language design, as an immensely viable user-facing feature, which became a cornerstone of Haskell.

Our paper on F-ing modules [15] already pointed out that existentials behave "like a monad" in our semantics, encapsulating the underlying "effect" of type generation. However, we never formally investigated the connection. A slightly more careful look reveals that it's not really a single monad, but a whole stack of them: one for each abstract type generated.

In the current paper, this interpretation as nested monads is no longer sufficient. Computation types are sums of existentials. In order to maintain this invariant under composition, composition can no longer be just nesting. Consequently, they give rise to a more general, more heterogeneous structure.

Hicks et al. [5] have recently investigated a generalisation of this kind of structure under the name *polymonad*. One way to describe it is as a *set* of monadic type constructors with heterogeneous bind (or join) operators. Independently, Tate [19] introduced a similar, slightly more general notion he calls *productoids*. In both cases, these formal structures were motivated by the desire to model certain forms of effects (though both works only investigate classical term-level effects).

Our computation types Φ, when factored into monadic constructors M, are an instance of this general structure. However, they are higher-kinded: they take a(nother) type constructor as argument, to allow transmitting the choice of type names to the "value" type. We leave a closer investigation of their exact relation to polymonads and productoids, and their formal properties, to future work.

5 Future Work

The current paper is primarily a sketch of a basic system. As always, there are many future roads to go. To mention only a few:

Implementation. We would like to integrate effect polymorphism into our 1ML prototype interpreter (mpi-sws.org/~rossberg/1ml/), to gather some practical experience from more experiments with the system.

Effect Inference. In the current paper we have only investigated the explicitly-typed fragment of 1ML. We believe that it is straightforward to incorporate *implicit functions* over effects to full 1ML, and enable the inference of effect parameters and arguments, just like for types.

More Effects. Our little language provides "impurity" (or partiality, if you prefer) as the only effect. That is as coarse as it can get. While already useful, it would be interesting to refine it to distinguish different concrete effects.

Abstract Effects. We have not yet explored what kind of abstractions might be enabled by the notion of abstract effect that our system introduces. Is it useful?

A Elaboration

A.1 Internal Language

As in the F-ing modules semantics [15] and the original 1ML paper [13], we define the semantics of the extended language by elaborating 1ML$_{\text{ex}}$ types and terms into types and terms of a (call-by-value, impredicative) variant of System F$_\omega$ with simple record types – see the syntax in Fig. 7.

$$\begin{array}{llll}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa \to \kappa \\
\text{(types)} & \tau & ::= & \alpha \mid \tau \to \tau \mid \{\overline{l{:}\tau}\} \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \top \mid \bot \mid (\tau+\tau)^\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau \\
\text{(terms)} & e,f & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid \{\overline{l{=}e}\} \mid e.l \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \\
& & & \mathsf{pack}\ \langle\tau,e\rangle_\tau \mid \mathsf{unpack}\ \langle\alpha,x\rangle{=}e\ \mathsf{in}\ e \mid \\
& & & \mathsf{inl}_\tau\ e \mid \mathsf{inr}_\tau\ e \mid \mathsf{asl}\ e \mid \mathsf{asr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.e \mid \mathsf{inr}\ x.e
\end{array}$$

$$\frac{}{\Gamma\vdash\top:\Omega}\qquad\frac{}{\Gamma\vdash\bot:\Omega}\qquad\frac{\Gamma\vdash\tau_1:\Omega\qquad\Gamma\vdash\tau_1:\Omega\qquad\Gamma\vdash\tau:\Omega}{\Gamma\vdash(\tau_1+\tau_2)^\tau:\Omega}$$

$$\frac{\Gamma\vdash e:\tau_1\qquad\Gamma\vdash\tau_2:\Omega}{\Gamma\vdash\mathsf{inl}_{\tau_2}\ e:(\tau_1+\tau_2)^\top}\qquad\frac{\Gamma\vdash e:\tau_2\qquad\Gamma\vdash\tau_1:\Omega}{\Gamma\vdash\mathsf{inr}_{\tau_1}\ e:(\tau_1+\tau_2)^\bot}$$

$$\frac{\Gamma\vdash e:(\tau_1+\tau_2)^\top}{\Gamma\vdash\mathsf{asl}\ e:\tau_1}\qquad\frac{\Gamma\vdash e:(\tau_1+\tau_2)^\bot}{\Gamma\vdash\mathsf{asr}\ e:\tau_2}$$

$$\frac{\Gamma\vdash e:(\tau_1+\tau_2)^{\tau'}\qquad\Gamma,x_1{:}\tau_1\vdash e_1:\tau\qquad\Gamma,x_2{:}\tau_2\vdash e_2:\tau}{\Gamma\vdash\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.e_1\mid\mathsf{inr}\ x_2.e_2:\tau}$$

Fig. 7. Syntax and non-standard typing rules of F$_\omega$ with binary GADTs

However, this time we need a small extension over plain F$_\omega$: to track generativity effects and the different forms of quantification they can cause in an indexed sum, we require a simple notion of GADT. We add this to F$_\omega$ in the simplest possible form: an indexed binary sum $(\tau_1 + \tau_2)^\tau$. Operationally, it is equivalent to an ordinary sum, with injections inl e and inr e and a case construct for elimination. However, the index τ allows deducing the choice statically: if it is equivalent to the type $\top$, then the value is known to be a left injection;

analogously, $\bot$ implies a right injection. Accordingly, there are two additional elimination constructs, $\mathsf{asl}\ e$ and $\mathsf{asr}\ e$, that are only valid on operands with known index, and perform the respective projection (they correspond to a one-armed case in more general forms of GADTs).

Figure 7 shows the syntax for this IL. The non-standard extensions just described are highlighted. The figure also includes the typing and kinding rules for those additional constructs. We omit reduction rules, because they are straightforward. The semantics is otherwise completely standard, and reuses the formulation from [15].

We write $\Gamma \vdash e : \tau$ for the F_ω typing judgement, and let $e \hookrightarrow e'$ denote (one-step) reduction. As one would hope, the language enjoys the standard soundness properties:

Theorem 1 (Preservation). *If $\cdot \vdash e : \tau$ and $e \hookrightarrow e'$, then $\cdot \vdash e' : \tau$.*

Theorem 2 (Progress). *If $\cdot \vdash e : \tau$ and e is not a value, then $e \hookrightarrow e'$ for some e'.*

To establish soundness of 1ML it suffices to ensure that elaboration always produces well-typed F_ω terms (Sect. A.3).

(kinds)		
Eff	$:=$	$\Omega \to \Omega \to \Omega$
$\mathsf{Mon}_{\overline{\kappa}}$	$:=$	$(\overline{\kappa} \to \Omega) \to \Omega$
(types)		
$[=\tau]$	$:=$	$\{\mathsf{typ} : \tau \to \{\}\}$
$[=\eta]$	$:=$	$\{\mathsf{typ} : \eta \to \{\}\}$
$\tau_1 \to_\eta \tau_2$	$:=$	$\tau_1 \to \{\mathsf{val} : \tau_2, \mathsf{eff} : [=\eta]\}$
$(\Phi_1 + \Phi_2)^\eta$	$:=$	$(\Phi_1 + \Phi_2)^{(\eta \top \bot)}$
$\forall\iota.\tau$	$:=$	$\forall\alpha_\iota.\mathsf{eff}\,\alpha_\iota \to \tau$
$\exists\iota.\tau$	$:=$	$\exists\alpha_\iota.\{\mathsf{eff} : \mathsf{eff}\,\alpha_\iota, \mathsf{val} : \tau\}$
eff	$:=$	$\lambda\alpha{:}\mathsf{Eff}.(\{\} + \{\})^{(\alpha \top \bot)}$
$\mathtt{I}$	$:=$	$\lambda\alpha_1\alpha_2.\alpha_1$
$\mathtt{P}$	$:=$	$\lambda\alpha_1\alpha_2.\alpha_2$
ι	$:=$	α_ι
$\eta_1 \vee \eta_2$	$:=$	$\lambda\alpha_1\alpha_2.\eta_1\,\alpha_1\,(\eta_2\,\alpha_1\,\alpha_2)$

(environments)		
Γ, ι	$:=$	$\Gamma, \alpha_\iota, x_\iota{:}\,\mathsf{eff}\,\alpha_\iota$
(terms)		
$[\tau]$	$:=$	$\{\mathsf{typ} = \lambda x{:}\tau.\{\}\}$
$[\eta]$	$:=$	$\{\mathsf{eff} = \lambda x{:}(\eta \top \bot).\{\}, \mathsf{val} = \eta\}$
$\lambda_\eta x{:}\tau.e$	$:=$	$\lambda x{:}\tau.\{\mathsf{val} = e, \mathsf{eff} = [\eta]\}$
$\lambda\iota.e$	$:=$	$\lambda\alpha_\iota.\lambda x_\iota{:}(\mathsf{eff}\,\alpha_\iota).e$
$\mathsf{pack}\,\langle\eta, e\rangle$	$:=$	$\mathsf{pack}\,\langle\eta, \{\mathsf{eff} = \eta, \mathsf{val} = e\}\rangle$
$\mathtt{I}$	$:=$	$\mathsf{inl}\,\{\}$
$\mathtt{P}$	$:=$	$\mathsf{inr}\,\{\}$
ι	$:=$	x_ι
$\eta_1 \vee \eta_2$	$:=$	$\mathsf{case}\ \eta_1\ \mathsf{of\ inl}\ x.\mathtt{I} \mid \mathsf{inr}\ x.$ $\mathsf{case}\ \eta_2\ \mathsf{of\ inl}\ x.\mathtt{I} \mid \mathsf{inr}\ x.\mathtt{P}$

Fig. 8. Encoding of semantic types in F_ω

A.2 Encoding Semantic Types

Elaboration translates $1\mathrm{ML}_{\mathrm{ex}}$ types directly into "equivalent" System F_ω types. The shape of these *semantic* types was given by the grammar in Fig. 2.

Not all the forms in that grammar are unadorned F_ω types, however. Some are auxiliary forms that are definable as syntactic sugar. Figure 8 shows how they can be encoded, along with respective term-level constructs for defining the evidence terms of the elaboration.

Several tricks in the elaboration are new relative to plain 1ML:

- Computation types are represented using the indexed binary sums introduced in the previous section. Their index is (the desugaring of) an effect type applied to $\top$ and $\bot$.
- To this end, effect types are represented as type constructors representing a simple Church-style encoding of Booleans. The kind of effect types hence is $\mathsf{Eff} = \Omega \rightarrow \Omega \rightarrow \Omega$, and their application behaves like a conditional.
- Effects also need to be reified on the term level, however, in order to enable reflection as needed in rule Sr. We use a binary sum (over units) for that purpose that is indexed by the corresponding type-level effect encoding – terms of type $\mathsf{eff}\ \eta$ are term-level encodings of effect type η.
- Since effects η can contain effect variables ι, this in turn requires those to be represented on both the type and term level. We use the trick of "twinning" each effect variable ι in the environment Γ as both a type variable α_ι and a term variable x_ι, assuming appropriate namespace injections. Likewise, every abstraction and quantification over effect variables consistently happens on both levels.

These encodings make sense, because the following consistency properties hold (note how we overload effects η as notation for both types and terms):

Proposition 1 (Derivable Rules for Effect Encodings). *Let Γ be a well-formed System F_ω environment.*

1. $\Gamma \vdash \mathtt{I} : \mathsf{Eff}$.
2. $\Gamma \vdash \mathtt{P} : \mathsf{Eff}$.
3. *If* $\iota \in \Gamma$*, then* $\Gamma \vdash \alpha_\iota : \mathsf{Eff}$.
4. *If* $\Gamma \vdash \eta_1 : \mathsf{Eff}$ *and* $\Gamma \vdash \eta_2 : \mathsf{Eff}$*, then* $\Gamma \vdash \eta_1 \vee \eta_2 : \mathsf{Eff}$
5. $\Gamma \vdash \mathtt{I} : \mathsf{eff}\ \mathtt{I}$.
6. $\Gamma \vdash \mathtt{P} : \mathsf{eff}\ \mathtt{P}$.
7. *If* $\iota \in \Gamma$*, then* $\Gamma \vdash x_\iota : \mathsf{eff}\ \alpha_\iota$.
8. *If* $\Gamma \vdash \eta_1 : \mathsf{eff}\ \eta_1$ *and* $\Gamma \vdash \eta_2 : \mathsf{eff}\ \eta_2$*, then* $\Gamma \vdash \eta_1 \vee \eta_2 : \mathsf{eff}\ (\eta_1 \vee \eta_2)$.

Here, we write "$\iota \in \Gamma$ as a shorthand for "$\Gamma(\alpha_\iota) = \mathsf{Eff} \wedge \Gamma(x_\iota) = \mathsf{eff}\ \alpha_\iota$", in correspondence to the twinning for effect variables explained above.

With the notation $\mathsf{Mon}_{\overline{\kappa}}$ to denote the kind of a monadic computation constructor mentioning abstract types of kinds $\overline{\kappa}$, similar consistency properties can be shown for the polymonad notation from Fig. 5:

Proposition 2 (Derivable Rules for Polymonads). *Let Γ be a well-formed System F_ω environment.*

1. *If* $\overline{\Gamma, \overline{\alpha} \vdash \pi : \kappa}$*, then* $\Gamma \vdash \exists\overline{\alpha}[\overline{\pi}] : \mathsf{Mon}_{\overline{\kappa}}$.

2. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}}$ *and* $\Gamma \vdash \eta : \mathsf{Eff}$, *then* $\Gamma \vdash (M_1 + M_2)^\eta : \mathsf{Mon}_{\overline{\kappa}}$.
3. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}_1}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}_2}$, *then* $\Gamma \vdash M_1 M_2 : \mathsf{Mon}_{\overline{\kappa}_1\overline{\kappa}_2}$.
4. *If* $\Gamma \vdash M : \mathsf{Mon}_{\overline{\kappa}_\alpha}$ *and* $\Gamma \vdash e_1 : M\lambda\overline{\alpha}.\tau_1$ *and* $\Gamma, \overline{\alpha}, x{:}\tau_1 \vdash e_2 : \tau_2$, *then* $\Gamma \vdash (\mathsf{do}_M\ \overline{\alpha}, x \leftarrow e_1\ \mathsf{in}\ e_2) : M\lambda\overline{\alpha}.\tau_2$.
5. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}_1}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}_2}$ *and* $\Gamma \vdash e : M_1\lambda\overline{\alpha}_1.M_2\lambda\overline{\alpha}_2.\tau$, *then* $\Gamma \vdash \mathsf{join}_{M_1M_2}\ e : M_1M_2\lambda\overline{\alpha}_1\overline{\alpha}_2.\tau$.

Note that our use of do-notation is a slight abuse (if you're coming from Haskell, anyway): it actually is a map, not a monadic bind – both are instances of polymonadic binds, however.

A.3 Meta-Theory

With the previous propositions we can verify that elaboration is correct:

Proposition 3 (Correctness of Elaboration). *Let* Γ *be a well-formed* F_ω *environment.*

1. *If* $\Gamma \vdash T/D \rightsquigarrow \Xi$, *then* $\Gamma \vdash \Xi : \Omega$.
2. *If* $\Gamma \vdash F \rightsquigarrow \eta$, *then* $\Gamma \vdash \eta : \mathsf{Eff}$.
3. *If* $\Gamma \vdash E/B :_\eta \Phi \rightsquigarrow e$, *then* $\Gamma \vdash e : \Phi$ *and* $\Gamma \vdash \eta : \mathsf{Eff}$. *Furthermore, if* $\eta = \mathsf{P}$ *then* $\Phi ! = \Sigma$.
4. *If* $\Gamma \vdash \Phi' \leq_{\overline{\alpha\alpha'}} \Phi \rightsquigarrow \delta; f$ *and* $\Gamma \vdash \Phi' : \Omega$ *and* $\Gamma, \overline{\alpha} \vdash \Phi : \Omega$, *then* $\mathrm{dom}(\delta) = \overline{\alpha}$ *and* $\Gamma \vdash \delta : \Gamma, \overline{\alpha}$ *and* $\Gamma \vdash f : \Phi' \to \delta\Phi$.

Together with the standard soundness result for F_ω we can tell that the extension of $1\mathrm{ML}_{\mathrm{ex}}$ is still sound:

Theorem 3 (Soundness of $1\mathrm{ML}_{\mathrm{ex}}$ with Effect Polymorphism). *If* $\cdot \vdash E : \Phi \rightsquigarrow e$, *then either* $e \uparrow$ *or* $e \hookrightarrow^* v$ *such that* $\cdot \vdash v : \Phi$ *and* v *is a value.*

References

1. Biswas, S.K.: Higher-order functors with transparent signatures. In: POPL (1995)
2. Dreyer, D., Crary, K., Harper, R.: A type system for higher-order modules. In: POPL (2003)
3. Gifford, D., Lucassen, J.: Integrating functional and imperative programming. In: LFP (1986)
4. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: POPL (1994)
5. Hicks, M., Bierman, G., Guts, N., Leijen, D., Swamy, N.: Polymonadic programming. In: MSFP (2014)
6. Kuan, G.: A true higher-order module system. Ph.D. thesis, University of Chicago (2010)
7. Kuan, G., MacQueen, D.: Engineering higher-order modules in SML/NJ. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 218–235. Springer, Heidelberg (2010)

8. Leroy, X.: Applicative functors and fully transparent higher-order modules. In: POPL (1995)
9. Lucassen, J., Gifford, D.: Polymorphic effect systems. In: POPL (1988)
10. MacQueen, D., Tofte, M.: A semantics for higher-order functors. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788. Springer, Heidelberg (1994)
11. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM TOPLAS **10**(3), 470–502 (1988)
12. Moggi, E.: Computational lambda calculus and monads. In: LICS (1989)
13. Rossberg, A.: 1ML - Core and modules united. In: ICFP (2015)
14. Rossberg, A., Dreyer, D.: Mixin' up the ML module system. ACM TOPLAS **35**(1) (2013)
15. Rossberg, A., Russo, C., Dreyer, D.: F-ing modules. JFP **24**(5), 529–607 (2014)
16. Russo, C.: Types for modules. In: ENTCS, vol. 60 (2003)
17. Shao, Z.: Transparent modules with fully syntactic signatures. In: ICFP (1999)
18. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. JFP **2**(3), 245271 (1992)
19. Tate, R.: The sequential semantics of producer effect systems. In: POPL (2013)
20. Wadler, P.: The essence of functional programming. In: POPL (1992)
21. Wadler, P., Thiemann, P.: The marriage of effects and monads. TOCL, **4**(1) (2003)

The Computer Scientist Nightmare
My Favorite Bug

Manuel Serrano(✉)

Inria Sophia Méditerranée, 2004 route des Lucioles - BP 93,
06902 Sophia Antipolis, Cedex, France
Manuel.Serrano@inria.fr

Hop has recently been used by a small French company, which we will call AIMM in the rest of this paper, to implement a new widget for a web multimedia application. This widget contained two parts: a music selector that lets end users browse a database of artists and music, and an HTML5 music player supporting on-demand conversion from one music format to another.

Figure 1 presents a screenshot of the widget. The music player is on the top, the database browser on the bottom. Clicking an artist name pushes a new panel presenting the songs that artist has produced. Clicking the small black arrow to the left of the name restores the previous panel. Graphical effects improve user experience. Depending on the speed of the platform and the web browser used, a new panel *slides* from left to right or *blends* into the previous one.

More than a real application, this was an endeavor, or an evaluation in order to understand how Hop fits in the context of realistic multimedia web application. The development was *a priori* easy because all the elements needed to implement the widget were provided by the Hop development kit off the shelf. I was in charge of the development that I estimated would take only a handful of days. With the AIMM engineers, we agreed on a common API that Hop could use to access the actual database; then I started to develop the widget. After a couple of days, everything went as we wished. The prototype was operational. Early tests showed that it was reliable enough to enter the second stage: to be tested on the AIMM server.

The AIMM server was a classical Linux Debian hosted by an x86/64 processor. A usual setting, almost the same as the one I used for developing the widget. After having installed additional Linux packages required by Hop, I installed the development kit and the prototype. Then I started to test the widget. At first, it seemed to be doing perfectly well. Everything seemed to be working as it should, until I realized that, on some browsers, clicking the artist names on the selector produced no effect. The new panel never showed up. Plagued with this erroneous behavior, the application was utterly useless. So began the battle I fought to understand and eliminate that bug.

1 A Multitier Bug

Mainly Firefox appeared to be affected by the disease. Other browsers, such as Chrome, Midori, or Opera, were doing well. At that early moment the logical

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 356–366, 2016.
DOI: 10.1007/978-3-319-30936-1_19

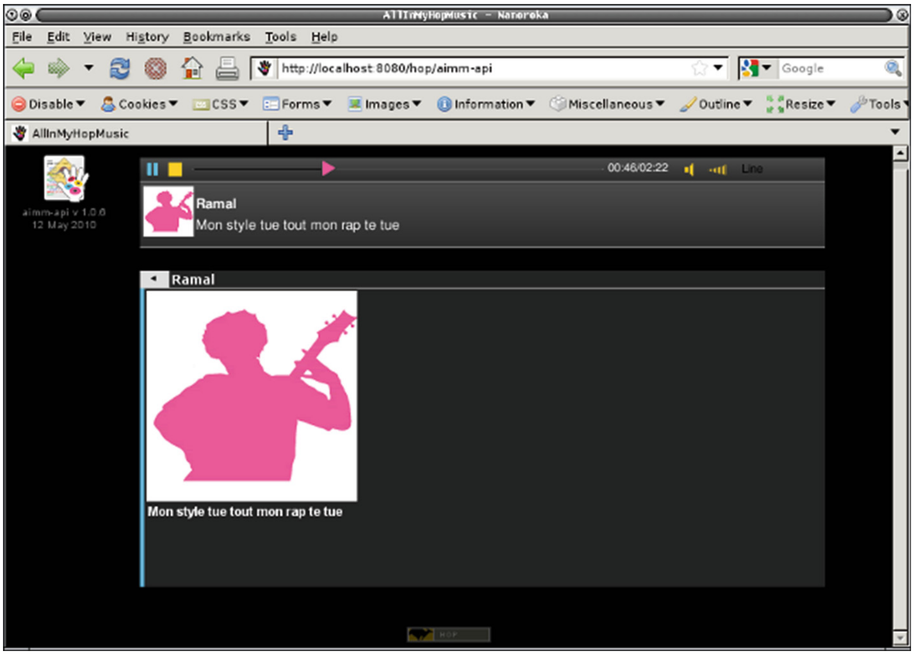

AllInMyHopMusic – Navoroka
File Edit View History Bookmarks Tools Help
http://localhost:8080/hop/aimm-api
Google
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools
AllInMyHopMusic
aimm-api v 1.0.0
12 May 2010
00:46/02:22
Line
Ramal
Mon style tue tout mon rap te tue
Ramal
Mon style tue tout mon rap te tue

Fig. 1. The Hop web widget

suspicion was that something specific to Firefox prevented it from executing the application correctly. However, this first intuition was contradicted by an obvious observation: the same Firefox was doing well when the web page was served by a server running on my machine, that is, when the application was executed entirely locally. This was particularly shocking because the client-side code served by the two servers, the AIMM server and my local machine, were supposed to be identical. My first investigation drove me to inspect the implementation of the *client-side* part of the application in order to spot which Firefox specificity was breaking its execution.

Hop is a multitier language. A single formalism is used to program the server-side and the client-side of web applications. The web page *visualized* on a browser is first *elaborated* on a server as an abstract syntax tree that is eventually compiled on-the-fly into Html. Both ends of the application use the same *Document Object Model* to reify Html trees as first class values. In Hop, Html tags are standard functions. Hop extends the official set of Html tags with numerous new widgets that are implemented by composing elementary objects. The artist selector was implemented using one of these objects: the SPAGE widget.

A SPAGE, which stands for *sliding page*, is an Html container (*i.e.*, a box) augmented with a title and a stack of inner elements. Only the top-most element is visible at a time. A visual effect gives the feeling that new elements *slide* over the previous ones. The source code using the SPAGE in the AIMM application was similar to:

```
(<SPAGE> :title "languages" :id "lang"
   (<DIV>
      (<SPAGE> :title "functional" :parent "lang"
         (<UL>
            (<LI> "Lisp")
            (<LI> "ML")
            (<LI> "Haskell"))))
   (<DIV>
      (<SPAGE> :title "sequential" :parent "lang"
         (<UL>
            (<LI> "Fortran")
            (<LI> "Pascal")
            (<LI> "C")))))
```

The functions <DIV>, <UL>, and <LI> are the Hop functions for the eponymous Html tags[1]. The function <SPAGE> is the constructor for the *sliding page*. In this example, the first page displays the words functional and sequential. When one is clicked, the corresponding page *slides* onto the screen and the associated enumeration is displayed. The default graphical configuration of SPAGE is defined by the following CSS rule set:

```
spage {
  background: white;
  border: 1px solid darkorange;
  font-size: 12px;
  border-radius: 0.2em;
  effect: auto;
}
```

The AIMM widget overrides these rules to use a gray background gradient and white texts, as shown in Fig. 1. The effect field lets applications choose the graphical effect they desire when a new element is pushed or popped. The four possible values are "fade", "slide", "auto", or "none". It defaults to "auto", which tells Hop to select the most suitable effect according to the characteristic of the browser. The CSS declaration is used by the client-side function spage-effect which implements the graphical effect on the browser. It is defined as:

[1] Note that in Hop, identifiers may contain the special characters "<", ">", "!", or "+". Hence, "<DIV>" is a regular identifier as is "string->integer", "remq!", or even "+".

```
(define (spage-effect node width step)
   (let ((e (node-computed-style-get node :effect)))
      (cond
         ((eq? e 'fade)
          (spage-fade node (if (> step 0) -0.3 0.2)))
         ((eq? e 'slide)
          (spage-slide node width step))
         ((eq? e 'none)
          (spage-none node (- step)))
         ((< (hop-config :browser-speed) 80)
          (spage-fade node (if (> step 0) -0.3 0.2)))
         (else
          (spage-slide node width step)))))
```

When the `effect` is “`auto`”, it chooses the *best* effect according to the browser graphical animation speed. If the browser graphical animation is slow, it *fades* elements. Otherwise, it *slides* them. At the time that problem occurred, browsers based on Gecko (Firefox) were not able to animate smoothly the visual effect, while browsers based on Webkit (Google-Chrome, Midori, Safari) could do it. So, the configuration `effect: auto` of the `SPAGE` widget yield two different visual effects on the two browser families. This explained why Firefox and Google-Chrome behaved differently: they were simply not using the same code for the animation. A brief additional examination of the client-side code unveiled that the very call to `spage-fade` was the reason for the problem.

The central piece of the Hop development kit is the web broker. It is a full-fledged web server which embeds the Hop compilers. When a program runs on the broker, *i.e.,* the server-side of the application, it generates the program which is installed on the web browser, *i.e.,* the client-side of the application. This client-side part is compiled on-the-fly into JavaScript, the universal language of web browsers, by the Hop broker. As it turned out, the call to `spage-fade` was erroneously compiled into JavaScript. Instead of:

```
SpageFade( node, step > 0 ? -0.3 : 0.2 );
```

it was compiled into:

```
SpageFade( node, step > 0 ? -0.0 : 0.0 );
```

These zeroes were of course the reason for the problem: the step of the fading being 0, the element stayed invisible forever. So, the problem was not due to Firefox executing incorrectly the code it received but to Hop delivering this incorrect code! Firefox being innocent, the question remained: *what was so special with the compilation of the `SpageFade` call that drove Hop to generate these wrong zeroes?*

2 Read, Compile, Print

Although Hop and JavaScript share many similarities (they are both fully polymorphic functional languages, using dynamic type checking, and automatic memory management), the compilation from Hop to JavaScript is not straightforward. First, JavaScript is not fully lexically scoped. Hence, a closure analysis is needed to compile Hop closures into JavaScript closures. Second, JavaScript is not properly tail-recursive, so static analyses are needed to efficiently compile Hop tail recursion into JavaScript loops. Third, because on the web the client-side code first traverses the network, its size matters. The client-side Hop compiler uses fancy features to generate code as compact as possible. All in all, this makes the implementation of the JavaScript compiler about 17KLOC of Hop code (the system is fully bootstrapped). The reason for the erroneous generation of "`0.0`" was likely to be located somewhere in these lines.

After thorough investigations I ended up with the conclusion that the Hop reader, that is the *server-side function in charge of reading Hop files*, was at some point broken. At the beginning of the application, the reader was correct and then, after a while, it was returning 0 for all floating point numbers. The compiler was then probably not broken *per se*, only the reader was wrong.

The Hop syntax is defined by a regular language. It can then be parsed efficiently by a finite state automaton such as those generated by the Hop form "`regular-grammar`". The actual implementation of the Hop parser looks like:

```
(regular-grammar ((float (or (: (* digit) "." (+ digit))
                             (: (+ digit) "." (* digit))))
                  ...)
   ...
   ((: (* digit)
       (or letter special)
       (* (or letter special digit (in ))))
    (the-symbol))
   ((: (? (in "-+"))
       (or float
           (: (or float (+ digit))
            (in "eE") (? (in "+-")) (+ digit))))
    (the-flonum))
   ...)
```

The first part defines variables (`float` in the code snippet above) which bind regular expressions used in the *rules*. These rules bind regular expressions, defined in a infix syntax, to expressions that are executed when a pattern matches. Regular grammars are compiled on the fly into Hop procedures. The implementation of regular grammars is about 3.5KLOC of Hop code for the compiler, seconded by 1.1KLOC of native code for the IO layer.

Compiling the Hop reader produces a function such as:

```
(lambda (ip)
   (define (the-flonum::double)
      (rgc_buffer_flonum ip))
   (define (the-fixnum::long)
      (rgc_buffer_fixnum ip))
   (define (the-symbol::symbol) ...)
   ...
   (define (STATE0 ip) ...)
   (case (STATE0 ip)
      ...
      ((10) (the-flonum))
      ...))
```

The function `rgc_buffer_flonum` extracts from the input buffer the corresponding floating point number. It is part of the native runtime system. The Hop native compiler (73KLOC of Hop code) can produce either native code, JVM bytecode, or .NET bytecode. The function `rgc_buffer_flonum` thus exists for the three backends. C was the backend used for the AIMM experiment. Its implementation is:

```
#define rgc_buffer_flonum( ip ) \
  strtod( RGC_INPUT_PORT_BUFFER( ip ), 0 );
```

As the reader at some point was no longer able to read numbers correctly, either `RGC_INPUT_PORT_BUFFER` or the libc function `strtod` had to be wrong! This was very shocking. I could not believe that Hop IO buffers were wrong. The overall size of the Hop implementation is about 325KLOC of Hop code which are all read by the Hop reader. If the buffers were not correctly managed, the bootstrap which reads all the 12,371,080 characters composing the source code could not reasonably succeed! I could not either believe that `strtod` was wrong. The Gnu libc is too widely used and too many applications depend on it for such an obvious error to occur. However, the Linux version running on the AIMM server being rather old, I spent half an hour googling to check if that particular version of the library was known to have a wrong `strtod` implementation. It was not.

3 The Usual Suspects

At this point of the paper, it is probably useful to make a short point. I was chasing a bug that *(i)* only appeared with Firefox but Firefox was innocent; *(ii) appeared* in the client-side of the application but it was *located* in the server-side of the application; *(iii)* was due to an error in the server-side code in charge of producing the client-side; *(iv)* was due to server-side code that seemed to first behave correctly and then, at some point of the execution, started to behave badly; *(v)* was not observable when executed on my machine but systematic when executed on the AIMM server. The obvious conclusion of all these observations was that something specific to the AIMM server was responsible for the wrong behavior.

When portability across a single operating system seems broken, the first idea that usually comes to mind is to check if there could be a potential confusion between pointers size. In our case we had an ideal suspect: my personal machine was using 32 bit pointers while the AIMM server was using 64 bit pointers! This suspicion was even strengthened by my previous investigation. As said before, I was more or less suspecting an IO buffer overflow and a confusion between 32 bit pointers and 64 bit pointers is very prone to buffer overflows. A quick test conducted on another x86/64 Linux machine of our own showed that Hop and the AIMM widget were running like a charm on that other 64 bit pointers computer. The problem was elsewhere.

The Hop broker (75KLOC) is deeply multi-threaded. When started, it spawns several preemptive threads (usually 20) that wait for connections. Each client request is handled in a separate thread. So, several client-side compilations may occur simultaneously. Multi-threaded applications are known to be a nightmare to program and an even worse nightmare to debug, in particular because multi-threading may break sequentially correct code. The C standard library is plagued with several non-thread-safe functions. In particular, all those that use static buffers, such as `strtok`. Would it be possible that `strtod` was also using a static buffer that made it non *thread-safe*? An excerpt of the Linux man page is given Fig. 2. It tells nothing about multi-threading and it proposes no thread-safe alternative. So, at least with the Gnu libc, it is very likely that `strtod` is thread safe. Another observation mostly invalidates the theory of a potential multi-threading problem. The bug was far too reproducible! The charm of multi-threading errors is their non-deterministic behavior. Here, the bug always occurred, at the same moment, on the same machine. Not something that could have happened with an error in the implementation of the parallelism.

After all, since the bug was reproducible, it was an ideal candidate for a debugger. I recompiled everything in debug mode and tried to learn more. No success. Firstly, the debugger was not easy to use because the error did not show up until the function had been called several thousand times. Secondly, it merely validated some already known facts: yes, the problem also showed up when the application was running inside the debugger; yes, after a while `the-flonum` was returning 0. Nothing else to learn. In addition to the debugger, I tried the read-eval-print loop Hop can spawn. Same result.

Since the debugger and the interactive loop were mostly useless, I instrumented the application to log all the calls to `strtod` and I wrote another program for replaying the logs. Surprisingly, this new program ran well, as all the calls to `strtod` completed correctly. So, if `strtod` was broken, it was broken in a weird context-dependent way. This experiment taught me another important fact: by observing the log player running correctly, I realized that many Hop instances were running *well* on the AIMM server. Something specific to the AIMM widget made it run incorrectly on the AIMM server!

```
STRTOD(3)              Linux Programmer's Manual              STRTOD(3)



NAME
       strtod, strtof, strtold - convert ASCII string to floating-point number

SYNOPSIS
       #include <stdlib.h>

       double strtod(const char *nptr, char **endptr);
       ...

DESCRIPTION
       The strtod(), strtof(), and strtold()  functions  convert  the  initial
       portion  of  the  string  pointed to by nptr to double, float, and long
       double representation, respectively.

       The expected form of the (initial portion of the)  string  is  optional
       leading white space as recognized by isspace(3), an optional plus ('+')
       or minus sign ('-') and then either (i) a decimal  number,  or  (ii)  a
       hexadecimal number, or (iii) an infinity, or (iv) a NAN (not-a-number).

       A decimal number consists of a nonempty sequence of decimal digits pos-
       sibly containing a radix character  (decimal  point,  locale-dependent,
       usually  '.'),  optionally  followed  by a decimal exponent.  A decimal
       exponent consists of an 'E' or 'e', followed by  an  optional  plus  or
       minus  sign,  followed  by  a  nonempty sequence of decimal digits, and
       indicates multiplication by a power of 10.

       ...

CONFORMING TO
       C89 describes strtod(), C99 describes the other two functions.
```

Fig. 2. The Linux man page for strtod

4 The Solution

At a moment, for no particular reason, by luck, or because I was unconsciously examining all the possible directions, the small detail that I had in front of my eyes since the beginning and that I had neglected so far attracted my attention: if `strtod` was wrong by returning 0.0 how come it returned −0.0 and not just 0.0 for "`-0.3`"? Where did this "`-`" come from? After all, it seemed that something was correct in the result since the sign was correct! A further experiment showed that actually the result was only wrong for the decimal part of the numbers. That is "`0.2`" was read as 0.0 but "`1.2`" was read as 1.0. After this observation everything went fast and easy.

As stated in the Linux man page (see above), `strtod` is locale-dependent. In French, the radix character is "`,`", contrary to English that uses "`.`".

Hence, parsing `-0.3` in English *must* return -0.3 while parsing it in French *must* return -0.0, because the french parsing of the number stops after parsing the characters `-` and `0`, ignoring the rest of the string. My computer and the AIMM server were both located in France but my machine was using an English setting while the AIMM server was using a French setting. On the AIMM server, the Linux global environment variable `LOCALE` was set to "`fr_FR`". This was the reason for the bug. The only problem yet to be solved was to understand why the behavior of `strtod` changed during the execution. Why did it start with an English setting and at some point switch to French?

The Hop server starts in a bare minimal core image. When an application is loaded it dynamically loads the libraries it depends on. Many multimedia Hop applications depend on the widely used Gstreamer library to manipulate multimedia material. In the AIMM application, it was used to encode MP3 into OGG on the fly (while Google-Chrome supports MP3, Firefox only supports OGG). So, when the AIMM application started it loaded Gstreamer.

The Gstreamer documentation specifies that an application should first call `gst_init` to initialize the library but it says nothing about the locale. However inspecting the source code, and in particular, the file gst/gst.c, I spotted the following:

```
static gboolean
init_pre (GOptionContext * context, GOptionGroup * group,
          gpointer data, GError ** error)
{
  ...
#ifdef ENABLE_NLS
  setlocale (LC_ALL, "");
  bindtextdomain (GETTEXT_PACKAGE, LOCALEDIR);
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
#endif /* ENABLE_NLS */
 ...
}
```

This clearly meant that Gstreamer allowed itself to change the locale of the application, with the side effect of changing the behavior of `strtod`, and in that case, breaking the Hop client-side compiler. The fix was straightforward and it took exactly two lines of code. It merely required saving the current locale before calling the `gst_init` function and to restore it afterward as in:

```
char *locale = setlocale( LC_ALL, 0 );
gst_init( &argc, &argv );
setlocale( LC_ALL, locale );
```

5 Concluding Remarks

This article is *unacademic* on purpose. It is written in a first-person narrative mode. it contains no citations, no mathematical formulae of any kind, and no algorithms, but it does contain code! Readers will eventually judge, but I think that in spite of containing no theorem, it demonstrates a flaw in the *principles* we use to design our languages and to write our programs because the bug that is described in this paper could probably occur in many different contexts and with most programming languages. *How many readers of this paper can be sure that none of their programs are affected by a similar problem?*

This paper tells a story of a program that apparently broke none of our commonly accepted rules but that still went wrong. The program suffered no memory leak. It was type safe, with respect to the common understanding of what a type should be. It suffered no data race. But still it went wrong!

Who is to blame for the error then? This is a legitimate question. Someone has to be blamed! Obviously I have my share of responsibility. It might be argued that the Hop implementation of `the-flonum` is incorrect and this is the reason for the whole problem. It definitively is and the function should be re-written. My knowledge of the C function `strtod` was too imprecise and it had yielded to the problem described in the paper. I'm the main culprit. However, it is inconceivable to re-implement all the low level functions used by Hop. It is likely that some other functions rely on a hidden state that might be changed externally, in particular amongst the functions used to implement the network and DNS APIs. However, re-writing these functions would take an unreasonable amount of work and independently of the feasibility of the task it is also worth wondering if this approach makes any sense. The purpose of Hop is to provide an infrastructure to help re-using code written elsewhere by other people, maybe using other formalisms. Re-writing all the core APIs in Hop would then contradict the overall goal of the whole system.

If I'm not the only one to blame, who else? Clearly the C functions `strtod` and `atof` have a peculiar design that is partly responsible for the problem. First, relying on external information, the locale, they make programs that use them unsealed. Second, using a weak definition for the external representation of numbers, they are prone to incorrectly parse numbers. Even worse, they offer no means to isolate the code relying on them. The locale used by this function is *always* a kind of oracle that cannot be controlled by the program.

In my opinion, the Gstreamer library also shares a part of the responsibility. First, it might be argued that no API should ever change the global context in which programs that use them execute. Second, if such a modification cannot be avoided, then it should be stated very explicitly in the documentation. In that particular case, Gstreamer seemed to call `setlocale` for no particularly good reason (parsing ID3 tags potentially expressed in the locale of the host is a weak motivation). Second, as reported, the documentation of the `gst_init` function does not even mention the change of the locale. This might be the biggest fault.

What tool could have helped to prevent the error or to detect it? Usual debuggers were of no help because, first, the execution was correct although with an unintended behavior (after all, `strtod` was configured to return -0.0), second, the lines of code that were incorrect were not implemented in the main language but in a hidden implementation language that the system tries hard to make invisible to the end programmer.

It is frequently argued that strong static type checking is a powerful tool that helps detecting errors soon. In this particular case it would have been of no help. With respect to the type system found in general purpose languages, everything in the execution was correct. Maybe this tells us that the types we are used to are inadequate? Maybe this tells that we should rely on types that denote concrete entities and not abstract mathematical things? Maybe the type of `strtod` should not be a function that takes a *string of characters* and that returns a real but a function that takes *a floating point number represented as a string of characters* and that returns a real? Maybe the type of `gst_init` should not only be *a function that accepts an integer and an array of strings and that returns an integer* but also something that denotes *the effect on the current locale*?

I think the real reason for the error presented in this paper is that the languages we have designed so far, and maybe even the way we think a program should be written, do not allow programs to compose safely. The error presented in this article came from an apparently innocuous function of a widely used library that was permitted to modify the whole behavior of the application. As long as the systems we design will be liable to such behaviors this kind of error will keep occurring. I think this story shows the limit of the current direction we are following in designing our languages and programs. I also think that it shows that we should explore different paths where safe composition is the starting point of the design. In the era we are to enter where programs need to use heterogeneous sets of resources and data, this will be vital for the reliability of our yet to be invented brave new applications.

A Branding Strategy for Business Types

Avraham Shinnar and Jérôme Siméon(✉)

IBM Research, 1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
{shinnar,simeon}@us.ibm.com

Abstract. In the course of building a compiler from business rules to a database run-time, we encounter the need for a type system that includes a class hierarchy and subtyping in the presence of complex record operations. Since our starting point is based on structural typing and targets a data-centric language, we develop an approach inspired by Wadler's work on XML types (Siméon and Wadler 2003). Our proposed type system has strong similarities with branded or tagged objects, combining nominal and structural typing, and is designed to support a rich set of operations on records commonly found in database languages. We show soundness of the type system and illustrate its use on two of the intermediate languages involved in our compiler for business rules: a calculus for pattern matching with aggregation (CAMP) that captures rules semantics, and the nested relational algebra (NRA) used for optimization and code generation. We show type soundness for both languages. The approach and correctness proofs are fully mechanized using the Coq proof-assistant.

1 Introduction

In order to support the growing amount of data routinely processed by applications, many programming languages are being developed (Cooper et al. 2007; Green et al. 2012) or extended (Cheney et al. 2013; Meijer et al. 2006) with query-like capabilities. Existing database techniques can then be leveraged for scalability through compilation to a database query language such as SQL (Grust et al. 2009) or a cloud library such as Spark (Armbrust et al. 2015). When extending object-oriented languages with such query capabilities, one of the key challenges is type system integration: database languages (e.g., SQL or the relational algebra) most often rely on structural types and feature rich operations on records (e.g., record concatenation), while object-oriented languages most often rely on nominal types and feature rich forms of subtyping. Inspired by Wadler's work on XML types (Siméon and Wadler 2003) as well as recent work on brands (Jones et al. 2015) or tagged objects (Lee et al. 2015), we propose a new type system that provides an elegant and natural way to combine nominal and structural typing in the presence of complex record operations, and illustrate its use in the context of a compiler for business rules.

Business Rules. Our work takes place as part of META (Arnold et al. 2016), a project in which rules are used to program applications that combine event processing and analytics. The use of rules is common in business intelligence

© Springer International Publishing Switzerland 2016
S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 367–387, 2016.
DOI: 10.1007/978-3-319-30936-1_20

```
public class Entity          public class Client
{                              extends Entity
  Int id;                    {
}                              String name;
                             }


public class Marketer        public final class DirectMarketer
  extends Entity               extends Marketer
{                            {
  String name;                 List<String> targets;
  Int client;                }
}


public class ProductRep      public final class C2Ps
  extends Marketer           {
{                              Int client;
  List<String> products;       List<String> reps;
}                            }
```

Fig. 1. Business object model.

applications as they can encode complex data-centric policies in a flexible manner. Languages for business rules go back to production rules (Forgy 1981), and modern specimens include Drools (Bali 2009) and JRules (Boyer and Mili 2011), the latter being our focus. Rules are written against collection(s) of objects described in a Business Object Model (BOM), which is essentially an object-oriented class hierarchy. Figure 1 shows an example BOM for a marketing application. A general `Entity` class is used to describe objects with an `id` attribute which serves as a primary key. Other classes derive from `Entity` and include `Clients` with an attribute `name` of type string, and `Marketers` with attributes `name` of type string and `client` of type `Client`. Two additional classes corresponding to categories of marketers are also defined: `DirectMarketer` with an attribute `targets` and `ProductRep` with an attribute `products`, both of which of type list of strings. The `extends` clause indicates the derivation hierarchy for each class, and when omitted means the class derives from the built-in `Object` class. Finally, class `C2Ps` is used to materialize a mapping from clients to their product reps, computed through the rule shown in Fig. 2.

That rule consists of a condition (`when`, Lines 2–7) and an action (`then`, Line 8). In effect, expressions in such rules perform pattern matching over an implicit value, in the context of an environment (variable bindings). The initial implicit value is not set and the initial environment is a single variable containing the collection of objects against which the rule is being matched. The `rule` construct sets the implicit value to be that input collection and the rule condition is matched against any of those objects and "fires" when a match is found. In our example, the condition binds variable `C` to a `Client`, and then aggregates all `ProductRep` objects `P` for whom `C`'s `id` is the same as `P`'s `client`. The aggregation expression is essentially an embedded query, with a syntax similar to that

```
1  rule FindProductReps {
2    when {
3      C: Client();
4      Ps: aggregate {
5            P: ProductRep(client == C.id);
6          } do { List<String> {P.name}; }
7    }
8    then { insert new C2Ps(C, Ps); }
9  }
```

Fig. 2. Rule computing a reverse mapping from clients to product reps.

of comprehensions (Trinder and Wadler 1988). The `List<String> { P.name }` expression builds a new collection containing the names of all matching elements resulting from the aggregate, and that collection is bound to variable `Ps`. The action part of the rule in Line 8 creates a new object of class `C2Ps` which materializes the mapping from `C` to `Ps`. Environment manipulation (e.g., binding variable `P`) is reflected in the compiler as record operations (e.g., concatenating a record with field `P` to the record for the current environment).

A Branding Strategy. A calculus and compilation scheme for business rules targeting the nested relational algebra (NRA) for optimization and scalability was proposed in (Shinnar et al. 2015). Proper typing is essential in order to exploit optimization techniques for bulk data processing developed by the database community (Beeri and Kornatzky 1993; Cluet and Moerkotte 1993; Fegaras and Maier 2000; May et al. 2006). However, most existing type systems for database languages, including the one we used in (Shinnar et al. 2015), rely on a purely structural approach which does not account for the nominal aspects of object-oriented data models and types. In addition, those typically assume homogeneous collections of objects and do not account for the subtyping relation implicit in the class hierarchy. In the context of Object-Oriented Databases, OQL relies on a purely nominal type system (Alagic 1999), while underlying algebras or calculi for optimization rely on a purely structural type system (Fegaras and Maier 2000; Cluet and Moerkotte 1993). A notable exception is Wadler's work on a type system for XML and XQuery presented in (Siméon and Wadler 2003), which integrates nominal and structural typing by annotating values and types with type names related through a derivation hierarchy. However, the focus on XML types and lack of support for records means it cannot be directly applied to classic object models or to operators from the nested relational algebra.

Inspired by this work, we propose a novel type system that can handle collections of objects in a class hierarchy that we believe is well suited for integration with database languages or language-integrated query systems (Cooper et al. 2007; Grust et al. 2009; Cheney et al. 2013). The approach relies on a notion of branded types and objects, where brands can be used to annotate values and types. Relationship between brands is captured by a derivation relation which encodes the nominal part of the semantics. From a structural typing stand point, we support open and closed records along with the corresponding subtyping relation, in the presence of a rich set of record operations that include projection and concatenation. From a nominal typing stand point, we support inheritance and (up/down) casting. Most of the treatment is language-agnostic, in that we define a set of core operators over brands independently from the host language. We then illustrate how those core operations can be integrated within two languages used in our business rules compiler: a calculus which captures rules' pattern matching semantics, and the nested relational algebra.

Paper Outline. The rest of the paper is organized as follows. In Sect. 2, we provide an overview for the proposed branding approach through examples. In Sect. 3, we describe the data model and type system. In Sect. 4, we introduce the subtyping relation and a notion of well-formed models which captures object-oriented schemas. In Sect. 5, we define key operators on records, bags and branded values along with their typing rules. Finally, in Sect. 6, we apply the proposed type system to a compiler from rules to NRA. Section 7 reviews related work and Sect. 8 concludes the paper with a brief report on our implementation and a discussion of future work.

2 Overview

Even though our work is motivated by business rules, most of the approach can be explained in terms of core operations over an object model. We first explain how classes and objects are encoded with brands, present key operations on those, then illustrate their use as part of our rules compiler.

Branded Model. In essence, branded types are types annotated with a name (a brand). Figure 3 shows the branded model corresponding to the Business Object Model from Fig. 1. Each class is described by a brand (the class name) and a type (the type of values of that class). For instance, the `Client` class corresponds to *Brand* Client which contains values of type record with attributes *id* of type `INT` and *name* of type `STRING`. We use [ ] to denote records, and { } to denote collection types[1]. The '[..]' (resp. '[|]') notation indicates open (resp. closed) records. An open record can be extended with new attributes while a closed one cannot. In our example, all branded types use open record, except for `C2Ps` and `DirectMarketer` which are declared as `final` in the source object model.

The brand derivation hierarchy is listed at the bottom of Fig. 3 and captures the nominal component of the type hierarchy. For instance, brand ProductRep derives from brand Marketer which itself derives from brand Entity. We assume the presence of an Object brand whose type is that of all possible records and which encodes the built-in Object class. We use Any to denote the top of the brand hierarchy. Compared to Fig. 1, we eliminate the `extends` relation, i.e., we fully expand the type for each class, and expose the inheritance relation in the

[1] We rely on a bag semantics but support for other collection types can be added easily, e.g., along the lines of (Fegaras and Maier 2000).

```
Brand Object [ ..]                     Brand Client
                                          [ id : INT;
Brand Entity [ id : INT ..]                 name : STRING ..]

Brand Marketer                         Brand DirectMarketer
    [ id : INT;                           [ id : INT;
      name : STRING;                        name : STRING;
      client : INT ..]                      client : INT;
                                            targets : {STRING} |]

Brand ProductRep                       Brand C2Ps
    [ id : INT;                           [ client : INT;
      name : STRING;                        reps : {STRING} |]
      client : INT;
      products : {STRING} ..]

    Object derives from Any            DirectMarketer derives from Marketer
    Entity derives from Object         ProductRep derives from Marketer
    Client derives from Entity         C2Ps derives from Object
    Marketer derives from Entity
```

Fig. 3. Branded object model.

derivation hierarchy[2]. This separation follows directly from (Siméon and Wadler 2003) and is central to our approach in that it allows to clearly separate the nominal and structural aspects of the type system. The branded model along with its derivation hierarchy is well-formed on the condition that the types of two brands related through (nominal) derivation be properly related through (structural) subtyping. For instance, adding the relation C2Ps *derives from* Entity would be invalid since the type for brand C2Ps does not have and *id* attribute.

Note that the model allows any type to be branded. For instance, `INT` is the type of integer values but we could give integers an object-like treatment by defining a brand for them. We could then define a special kind of integer for ids through derivation, as illustrated by the following declarations.

Brand Int `INT`
Brand Id `INT`
Id *derives from* Int

Through subtyping, every operation taking a branded value of type *Brand* Int also takes a value of type *Brand* Id, but not the converse.

Branded Values. In essence, branded values are values annotated with a name (a brand). Figure 4 shows examples of values in our data model. In each case we provide a name for the value (variable), a type and the value itself. All the

[2] The formal model actually relies on the reflexive and transitive closure of the derivation hierarchy which must be a partial order (i.e., without cycles).

```
M1Id : INT = 1

C1 : [ id : INT; name : STRING |] = [ id : 1 ; name : “Disney” ]
C2 : [ id : INT ..] = [ id : 1 ; name : “Disney” ]

M1 : Brand DirectMarketer =
  brand DirectMarketer ([ id : 101 ; name : “John”; client : 1;
                          targets : { “iOS”, “Gawker” } ])

M2 : Brand Marketer =
  brand DirectMarketer ([ id : 101 ; name : “John”; client : 1;
                          targets : { “iOS”, “Gawker” } ])

Clients : {Brand Client} =
  { brand Client ([ id : 1 ; name : “Disney” ]),
    brand Client ([ id : 2 ; name : “Netflix” ]),
    brand Client ([ id : 3 ; name : “HBO” ]) }

Marketers : {Brand Marketer} =
  { brand DirectMarketer ([ id : 101 ; name : “John”; client : 1;
                            targets : { “iOS”; “Gawker” } ]),
    brand ProductRep ([ id : 102 ; name : “Julie”; client : 1;
                        products : { “Star Wars” } ]),
    brand DirectMarketer ([ id : 103 ; name : “Jack”; client : 3;
                            targets : { “Instagram” } ])}

WM : {Brand Entity} = Clients ∪ Marketers
```

Fig. 4. Branded values.

examples are well-typed, i.e., the given value has the given type. The first three are simple values without brands: M1Id is an integer value, C1 and C2 are records. Note that C1 (resp. C2) is declared as having a closed (resp. open) record type. M1 is a value with brand DirectMarketer whose content is a record with id 101 and name *“John”*, working for client with id 1 and targeting *“iOS”* and *“Gawker”*. Value M2 is the same value but declared with type *Brand* Marketer, which is also valid since DirectMarketer derives from Marketer. The remaining of Fig. 4 shows examples of collections. Clients is an homogeneous collection of values with brand Client, and Marketers a collection of object in the sub-brand hierarchy for brand Marketer. Finally the last collection WM is the union of Clients and Marketers and can be typed as a collection of values with brand Entity.

Brand Operations. We provide three core operations on brands: branding, unbranding, and casting. The first two are constructors/destructors for branded values. For instance, the following operations create a value with brands Id, Client and Entity. (We use the notation: Var : $Type = Expr \Downarrow Value$ to indicate that variable Var has type $Type$ and value $Value$ which is the result of $Expr$).

$$\begin{aligned}
\texttt{M1Id'} &: \mathit{Brand}\ \mathsf{Id} = \mathsf{brand\ Id}\ (\texttt{M1Id}) \Downarrow \mathsf{brand\ Id}\ (1)\\
\texttt{C1'} &: \mathit{Brand}\ \mathsf{Client} = \mathsf{brand\ Client}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“\textit{Disney}”}])\\
\texttt{C1''} &: \mathit{Brand}\ \mathsf{Entity} = \mathsf{brand\ Client}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“\textit{Disney}”}])\\
\texttt{C1'''} &: \mathit{Brand}\ \mathsf{Entity} = \mathsf{brand\ Entity}\ (\texttt{C1}) \Downarrow\\
&\qquad \mathsf{brand\ Entity}\ ([\mathit{id} : 1; \mathit{name} : \text{“\textit{Disney}”}])
\end{aligned}$$

Note that branding checks that the value being branded has the type declared for the corresponding brand. In the previous examples, C1 is declared as a closed record with attributes *id* and *name* and is indeed a subtype of the corresponding open record for both brands Client and Entity. However, the following two operations are not well typed, since the type for C1 (resp. M1Id) isn't a subtype of the type for brand Marketer (resp. Entity).

$$\begin{gathered}
\mathsf{brand\ Marketer}\ (\texttt{C1})\\
\mathsf{brand\ Entity}\ (\texttt{M1Id})
\end{gathered}$$

Unbranding, denoted !, always succeeds on a branded value and returns its content. Unbranding uses the static type of the brand (which can be changed with a cast) to determine the type of the returned contents. Here are a few examples, also illustrating record operations such as field access (denoted $.A$ where A is an attribute) and record projection (denoted $\pi_{\overline{A_i}}$ where $\overline{A_i}$ is a set of attributes). Note that the typing rule for projection yields a closed record type.

$$\begin{aligned}
\texttt{Id1} &: \texttt{INT} = !\texttt{M1Id'} \Downarrow 1\\
\texttt{C1id} &: \texttt{INT} = !\texttt{C1'}.\mathit{id} \Downarrow 1\\
\texttt{C1proj} &: [\mathit{id} : \texttt{INT}\ |] = \pi_{\mathit{id}}(!\texttt{C1'}) \Downarrow [\mathit{id} : 1]
\end{aligned}$$

Unbranding is typed using the statically known brand type, which means that !C1''.*name* is not well typed. In other words, the fields that are present in C1'' but not declared for the brand Entity are hidden, unless casting is first applied to the value. This once again corresponds to the standard behavior in an OO context. At run-time, casting checks if a given brand is a supertype (or the same) as a brand's dynamic type and returns the original branded value or fails[3]. The typing rule for casting asserts that the returned brand has the provided brand type. This allows safe downcasting on brands, allowing code to enrich the static type of a brand at the cost of a runtime check. Here are some examples of successful casts applied in the context of the nested relational algebra, in which success (resp. failure) is represented as a singleton (resp. empty) bag.

$$\begin{aligned}
\texttt{UpM1Id} &: \{\mathit{Brand}\ \mathsf{Int}\} = (\mathsf{Int})\ \texttt{M1Id'} \Downarrow \{\mathsf{brand\ Id}\ (1)\}\\
\texttt{UpC1} &: \{\mathit{Brand}\ \mathsf{Entity}\} = (\mathsf{Entity})\ \texttt{C1'} \Downarrow\\
&\qquad \{\mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“\textit{Disney}”}])\}\\
\texttt{DownC1} &: \{\mathit{Brand}\ \mathsf{Client}\} = (\mathsf{Client})\ \texttt{C1''} \Downarrow\\
&\qquad \{\mathsf{brand\ Client}\ ([\mathit{id} : 1; \mathit{name} : \text{“\textit{Disney}”}])\}
\end{aligned}$$

[3] What failure means may vary as we will show when applying casting in the context the two intermediate languages (CAMP and NRA) involved in our rules compiler.

The first two examples are upcasts: from *Brand* Id to *Brand* Int, and from *Brand* Client to *Brand* Entity. The third example is a downcast from *Brand* Entity to *Brand* Client. We now give some examples of cast failures, all of which return the empty bag in the context of the nested relational algebra.

$$\begin{aligned}
&\texttt{Fail1} : \{\mathit{Brand}\ \mathsf{Entity}\} = (\mathsf{Entity})\ \texttt{M1Id'} \Downarrow \{\}\\
&\texttt{Fail2} : \{\mathit{Brand}\ \mathsf{Client}\} = (\mathsf{Client})\ \texttt{C1'''} \Downarrow \{\}\\
&\texttt{Fail3} : \{\mathit{Brand}\ \mathsf{ProductRep}\} = (\mathsf{ProductRep})\ \texttt{M1} \Downarrow \{\}
\end{aligned}$$

Note that the specific failure semantics for NRA is unusual from a classic OO perspective, but it enables easy integration with other relational operators. For instance, the following combination of flatten, map (χ) and casting can be used to select all objects in variable `WM` that are product reps. In that example, **In** stands for the current element in the input collection processed by the map. The output of the map is a bag of either singleton or empty bags which can then be flattened. The final type is a bag of *Brand* ProductRep as expected.

$$\begin{aligned}
\texttt{Reps} : \{\mathit{Brand}\ \mathsf{ProductRep}\} = \mathit{flatten}(\chi_{\langle(\mathsf{ProductRep})\ \textbf{In}\rangle}(\texttt{WM})) \Downarrow&\\
\{\mathsf{ProductRep}([\mathit{id} : 102; \mathit{name} : \text{“}\mathit{Julie}\text{”}; \mathit{client} : 1;&\\
\mathit{products} : \{\text{“}\mathit{Star\ \ Wars}\text{”}\}])\}&
\end{aligned}$$

Rules Compiler. With that notion of branded values and types in hand, we extend the rules calculus and compiler proposed in (Shinnar et al. 2015) to provide proper support for the class hierarchy. Going back to the example rule from Fig. 2, the binding within the aggregate expression on Line 5 can be translated to the nested-relational algebra as follows:

$$\bowtie^d_{\langle[P:\textbf{In}.it]\rangle}\Big(\sigma_{\langle!(\textbf{In}.C).id=!(\textbf{In}.it).client\rangle}\Big(\rho_{it/\{br\}}\Big(\bowtie^d_{\langle[br:(ProductRep)\ (\textbf{In}.it)]\rangle}(\textbf{In})\Big)\Big)\Big) \tag{1}$$

From an input bag of records (rightmost **In**) each containing a working memory elements (in attribute *it*) and a binding for variable `C` (in attribute C), select those whose brand is ProductRep. Then, select those whose *client* attribute is the same as the client id ($!(\textbf{In}.C).id$). Then, add the attribute P containing the current value in attribute *it* to each remaining record. Note that the notion of current value (**In**) depends on context and its scope changes within every sub-operator between angle brackets $\langle..\rangle$. In the first operand of the select (σ) operator, it is bound to each record from the bag returned by its second operand. The $\bowtie^d$ operator (traditionally called dependent join in the database literature (Cluet and Moerkotte 1993)) performs record concatenation, adding attribute P to every record in its input. The expression makes use of *casting* and *unbranding*. The first expression within the right-most $\bowtie^d$ uses a cast to filter the input objects with the right brand ($(\mathit{ProductRep})\ (\textbf{In}.it)$) and adds a temporary field *br* in the input record that contains values of type $\{\mathit{Brand}\ \mathsf{ProductRep}\}$. The next NRA operator (ρ) is used to unnest the content of attribute *br*, then re-bind it to attribute *it*, which now has type *Brand* ProductRep. The subsequent selection operator can now access the *client* attribute safely by unbranding the input value $(!(\textsc{In}.it).client)$.

Remarks. Before we move to a formal treatment of the proposed model, a few aspects are worth pointing out. Firstly, support for both open and closed records is a key requirement: open records allow to model classes and inheritance, while closed records are essential when using relational operations such as Cartesian product or (dependent) join both of which perform record concatenation. We handle both by combining: (i) subtyping between open and closed records, (ii) record operations, notably projection, that allow to coerce an open record into a closed record. Secondly, the combination of operations on brands and records enables a rich feature set that includes classic features from OO language, many of which can be expressed as combinations of our core operators. The presentation here focuses on data-centric languages and notably, we do not attempt to model features such as methods or object identity.

3 Data and Types

In this section, we first define the data model and core type system with the notion of brands. Note that both rely only on a hierarchy of brand names.

3.1 Data Model

Values in our data model, the set $\mathcal{D}$, are atoms, records, bags or branded values. We assume a sufficiently large set of atoms $a, b, ...$ including integers in $\mathcal{Z}$, strings in $\mathcal{S}$, the Boolean values true and false, and a null value written nil. A bag is a multiset of values in $\mathcal{D}$; we write $\emptyset$ for the empty bag and $\{d_1, ..., d_n\}$ for the bag containing the values $d_1, ..., d_n$.

A record is a mapping from a finite set of attributes to values in $\mathcal{D}$, where attribute names are drawn from a sufficiently large set $A, B, ...$ We write $[\,]$ for the empty record and $[\overline{A_i : d_i}]$ for the record mapping A_i to d_i. We define the concatenation $x * y$ of two records as a mapping from $\mathrm{dom}(x) \cup \mathrm{dom}(y)$ to values, such that $[x * y](A) = x(A)$ if $A \in \mathrm{dom}(x)$ and $[x * y](A) = y(A)$ if $A \in \mathrm{dom}(y) \setminus \mathrm{dom}(x)$. Records x and y are $\mathsf{compatible}$ if $\forall A \in \mathrm{dom}(x) \cap \mathrm{dom}(y), x(A) = y(A)$. We define the sum $[x + y]$ of compatible records as the union $x \cup y$, and leave it undefined for non-compatible records.

A branded value is a pair of a brand name and a value, where brand names are drawn from a sufficiently large set $\mathsf{A}, \mathsf{B}, ...$ We write $\mathsf{brand}\ \mathsf{A}\ (d)$ for the value d branded with A. We assume a derivation hierarchy, which is a partial order relation δ between brands, with a most general brand denoted Any. We write $\delta(\mathsf{A}, \mathsf{A}')$ to denote that A *derives from* A' (through reflexive and transitive closure). For every brand A, we assume that $\delta(\mathsf{A}, \mathsf{Any})$. The fact that δ is a partial order is essential for the soundness of the type system. We allow a brand name to derive from several brand names which accounts for multiple inheritance.

Finally, we assume structural equality between values, denoted $d_1 \doteq d_2$, and defined inductively as follows. Two atomic values are equal if their underlying built-in equality (between two booleans, two strings, two integers) holds. Two bags are equal if they have the same values (compared through equality) modulo permutation. Two record are equal if they have the same set of attributes and the values associated to corresponding attribute are equal. Branded values are equal if they have the same brand and their contents are equal.

3.2 Types

Our types include primitive types `NIL`, `INT`, `BOOL`, and `STRING`. The type of a (homogeneous) bag with elements of type τ is written $\{\tau\}$. We also assume the existence of a top (resp. bottom) type denoted $\top$ (resp. $\bot$).

We define two kinds of record types: closed and open. $[\overline{A_i : \tau_i} \;|]$ is the type of a *closed* record with attributes A_i of type τ_i. $[\overline{A_i : \tau_i} \;..]$ is the type of an *open* record with attributes A_i of type τ_i. When there is no ambiguity, we sometimes write $[\overline{A_i : \tau_i}]$ for a record type that can be either open or closed.

As with data records, we define a notion of compatibility for record types. Two open (resp. closed) record types are considered compatible if all their overlapping attributes (resp. known overlapping attributes) have the same type. Note that two records can have compatible types but incompatible data. Record concatenation, $[\overline{A_i : \tau_{A_i}} * \overline{B_j : \tau_{B_j}} |]$ concatenates two record types, favoring the types of A's attributes in case of conflict and is only defined for closed records. Record merge, $[\overline{A_i : \tau_{A_i}} + \overline{B_j : \tau_{B_j}}]$, also concatenates the record types and is defined for both open and closed records, but only if they are compatible.

Finally, *Brand* A is the type of values with brand A.

4 Subtyping and Models

In this section, we define structural subtyping and the notion of well-formed model used to capture class hierarchies. We then describe what it means for a value to be well-typed and show this notion is consistent with subtyping.

4.1 Subtyping

We can now define the subtyping relation, which is still purely structural, depending only on the given derivation hierarchy δ over brand names. Figure 5 defines $\tau \preccurlyeq_\delta \tau'$, the subtyping relation between two types given δ. The first two rules simply place $\top$ (resp. $\bot$) at the top (resp. bottom) of the type hierarchy. A type is a subtype of itself (SRefl). A bag of type τ is a subtype of a bag of type τ' iff, τ is a subtype of τ' (SBag). A type with brand A is a subtype of a type with brand A' iff A *derives from* A' in δ (SBrand).

A record is a subtype of a closed record iff, it is also closed, has the same domain (same attributes), and each of its attributes' type is a subtype of the corresponding attribute in the super type (SClosed). The subtype of an open record can be open or closed, and it must have at least the same attributes, with the proper subtyping relationship between the attribute in common (SOpen). In

$$\frac{}{\tau \preccurlyeq_\delta \top}\ (\text{STop}) \qquad \frac{}{\bot \preccurlyeq_\delta \tau}\ (\text{SBottom}) \qquad \frac{}{\tau \preccurlyeq_\delta \tau}\ (\text{SRefl})$$

$$\frac{\tau \preccurlyeq_\delta \tau'}{\{\tau\} \preccurlyeq_\delta \{\tau'\}}\ (\text{SBag}) \qquad \frac{\delta(\mathsf{A}, \mathsf{A}')}{\mathit{Brand}\ \mathsf{A} \preccurlyeq_\delta \mathit{Brand}\ \mathsf{A}'}\ (\text{SBrand})$$

$$\frac{\tau_1 \preccurlyeq_\delta \tau_1' \quad \dots \quad \tau_n \preccurlyeq_\delta \tau_n'}{[A_1 : \tau_1, ..., A_n : \tau_n\ |] \preccurlyeq_\delta [A_1 : \tau_1', ..., A_n : \tau_n'\ |]}\ (\text{SClosed})$$

$$\frac{\tau_1 \preccurlyeq_\delta \tau_1' \quad \dots \quad \tau_n \preccurlyeq_\delta \tau_n'}{[A_1 : \tau_1, ..., A_n : \tau_n, \overline{B_i : \tau_i''}] \preccurlyeq_\delta [A_1 : \tau_1', ..., A_n : \tau_n'\ ..]}\ (\text{SOpen})$$

Fig. 5. Subtyping. $\boxed{\tau \preccurlyeq_\delta \tau'}$

other words, open records support both width and depth subtyping, while closed records support only depth subtyping.

Figure 5 does not include rules for transitivity and antisymmetry which are both consequences of the definition. Subtyping is indeed a partial order over types, as stated by the following theorem[4].

Theorem 1 (Subtype is a Partial Order). *Given a derivation hierarchy δ, for all types τ_1, τ_2, τ_3:*

$$(\tau_1 \preccurlyeq_\delta \tau_1)$$
$$\textit{if}\ (\tau_1 \preccurlyeq_\delta \tau_2) \wedge (\tau_2 \preccurlyeq_\delta \tau_3)\ \textit{then}\ (\tau_1 \preccurlyeq_\delta \tau_3)$$
$$\textit{if}\ (\tau_1 \preccurlyeq_\delta \tau_2) \wedge (\tau_2 \preccurlyeq_\delta \tau_1)\ \textit{then}\ (\tau_1 = \tau_2)$$

4.2 Models

A model μ_δ is a relation from brands to types wrt a given derivation hierarchy δ. We write $\langle\rangle_\delta$ for the empty model and $\langle\overline{\mathsf{A}_i \mapsto \tau_i}\rangle_\delta$ for the model mapping A_i to type $\mu_\delta(\mathsf{A}_i) = \tau_i$.

Definition 1 (Well-Formed Model). A model μ_δ is well-formed with respect to a brand derivation hierarchy δ, denoted $\models \mu_\delta$, iff:

$$\forall \mathsf{A}, \mathsf{A}',\ \text{if}\ \delta(\mathsf{A}, \mathsf{A}')\ \text{then}\ \mu_\delta(\mathsf{A}) \preccurlyeq_\delta \mu_\delta(\mathsf{A}')$$

4.3 Typed Data

We use the notation $\mu_\delta \vdash d : \tau$ to mean that data d has type τ in the context of model μ_δ. Figure 6 defines the typing rules for data. The first few rules are trivial: every data has type $\top$ (TTop), and atoms have the type corresponding to the kind of data they belong to (TNil, TInt, TString, TTrue, TFalse). As expected, no data can have the type $\bot$.[5] Bags are values of uniform types (TEmpty, TBag).

[4] All theorems in the paper have been verified using Coq.

[5] Note that $\bot$ is still useful to have in the type system, for instance the type $\{\bot\}$ is the most precise type for the empty bag.

$$\frac{}{\mu_\delta \vdash d : \top}\ (\text{TTop}) \qquad \frac{}{\mu_\delta \vdash \mathsf{nil} : \mathtt{NIL}}\ (\text{TNil}) \qquad \frac{d \in \mathcal{Z}}{\mu_\delta \vdash d : \mathtt{INT}}\ (\text{TInt})$$

$$\frac{d \in \mathcal{S}}{\mu_\delta \vdash d : \mathtt{STRING}}\ (\text{TString}) \qquad \frac{}{\mu_\delta \vdash \mathsf{true} : \mathtt{BOOL}}\ (\text{TTrue}) \qquad \frac{}{\mu_\delta \vdash \mathsf{false} : \mathtt{BOOL}}\ (\text{TFalse})$$

$$\frac{}{\mu_\delta \vdash \emptyset : \{\tau\}}\ (\text{TEmpty}) \qquad \frac{\mu_\delta \vdash d_1 : \tau \quad ... \quad \mu_\delta \vdash d_n : \tau}{\mu_\delta \vdash \{d_1, ..., d_n\} : \{\tau\}}\ (\text{TBag})$$

$$\frac{\mu_\delta \vdash d_1 : \tau_1 \quad ... \quad \mu_\delta \vdash d_n : \tau_n}{\mu_\delta \vdash [A_1 : d_1, ..., A_n : d_n] : [A_1 : \tau_1, ..., A_n : \tau_n \ ||]}\ (\text{TClosed})$$

$$\frac{\mu_\delta \vdash d_1 : \tau_1 \quad ... \quad \mu_\delta \vdash d_n : \tau_n \quad \mu_\delta \vdash d'_1 : \tau'_1 \quad ... \quad \mu_\delta \vdash d'_k : \tau'_k}{\mu_\delta \vdash [A_1 : d_1, ..., A_n : d_n, \overline{B_i : d'_i}] : [A_1 : \tau_1, ..., A_n : \tau_n \ ..]}\ (\text{TOpen})$$

$$\frac{\mu_\delta(A) = \tau \quad \delta(A, A') \quad \mu_\delta \vdash d : \tau}{\mu_\delta \vdash \mathsf{brand\ A}\ (d) : \mathit{Brand}\ \mathsf{A'}}\ (\text{TBrand})$$

Fig. 6. Typed data. $\boxed{\mu_\delta \vdash d : \tau}$

For a closed record type, a record value belongs to that type if it has the same attributes as its type and if each attribute value has the corresponding attribute type (TClosed). For an open record type, a record value belongs to that type if it has at least the attributes defined in its type and for those, each attribute value has the corresponding attribute type (TOpen). Those attributes not declared in the open record type are only assumed to be well typed. The rule for brands simply state that the content of a branded value must be of the type declared for that brand in the model μ_δ (TBrand).

The key property of the typing rules for data is that they are sound with respect to subtyping, as expressed by the following theorem.

Theorem 2 (Soundness of Typing Rules for Data). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\textit{if}\ (\mu_\delta \vdash d : \tau) \wedge (\tau \preccurlyeq_\delta \tau')\ \ \textit{then}\ \ (\mu_\delta \vdash d : \tau')$$

I.e., if data d has type τ in the context of μ_δ, and τ is a subtype of τ', then data d has type τ' in the context of μ_δ.

5 Operators

We now define operations over the proposed data model. Besides operations on records and bags typically found in data languages (notably record concatenation and projection), we include operations for branding and unbranding. We leave casting for the next section, as the semantics of cast depends on the specific ways failure is handled in the host language.

5.1 Syntax and Semantics

Unary or binary *operators* are basic operations over the data model defined as functions. Most of those are only defined for data with a specific type. We provide an informal dynamic semantics, followed by typing rules which gives the precise conditions under which those operators are well-typed. A small denotational semantics is given for reference in Appendix A.

Definition 2 (Operators).

$$\begin{array}{lll} (\text{uops}) & \oplus d & ::= \mathit{ident}\, d \mid \{d\} \mid \mathit{flatten}\, d \mid \neg d \mid [A\!:\!d] \mid d.A \\ & & \quad\mid\ d{-}A \mid \pi_{\overline{A_i}} \mid \mathsf{brand\ A}\ (d) \mid\ !d \\ (\text{bops}) & d_1 \otimes d_2 & ::= d_1 = d_2 \mid d_1 \in d_2 \mid d_1 \cup d_2 \mid d_1 * d_2 \mid d_1 + d_2 \end{array}$$

In order of presentation, the unary operators do the following:

ident d	returns d.
$\{d\}$	constructs a singleton bag containing the value d.
flatten d	flattens a bag of bags.
$\neg d$	negates a Boolean.
$[A\!:\!d]$	constructs a record with a single attribute A and value d.
$d.A$	accesses the value associated with attribute A in record d.
$d{-}A$	returns a record with all attributes of d except A.
$\pi_{\overline{A_i}}(d)$	returns the projection of record d over attributes $\overline{A_i}$.
brand A (d)	returns a value with brand A containing d.
$!d$	returns the content of branded value d.

In order of presentation, the binary operators do the following:

$d_1 = d_2$	compares two values for equality.
$d_1 \in d_2$	returns true if and only if d_1 is an element of bag d_2.
$d_1 \cup d_2$	returns the union of two bags.
$d_1 * d_2$	concatenates two records, favoring d_1 for overlapping attributes.
$d_1 + d_2$	returns a singleton bag with the concatenation of the two records if they are compatible, and returns $\emptyset$ otherwise.

Operators can be easily extended (e.g., for arithmetics or aggregation). *flatten* corresponds to a single-level flattening of a nested bag. Record operations are sufficient to support all standard relational and nested relational operators. The last two unary operators correspond to branding and unbranding. Branding, brand A (d), creates a new value with brand A and content d. Unbranding, $!d$, returns the content of a branded value d. The definition for those does not make any assumption about the model of the corresponding brands, but consistency with a given model results from their typing rules which are given next.

5.2 Typing

Given a well-formed model μ_δ, the type signatures of unary operators, written $\mu_\delta \vdash \oplus d : \tau \rightarrow \tau'$ (i.e., operator $\oplus$ applied to a value d of type τ has return type τ'), are as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\mu_\delta \vdash \mathit{ident}\, d & : & \tau \rightarrow \tau & \mu_\delta \vdash [A\!:\!d] & : & \tau \rightarrow [A : \tau\ |] \\
\mu_\delta \vdash \neg d & : & \texttt{BOOL} \rightarrow \texttt{BOOL} & \mu_\delta \vdash d.A & : & [A : \tau] \rightarrow \tau \\
\mu_\delta \vdash \{d\} & : & \tau \rightarrow \{\tau\} & \mu_\delta \vdash d{-}A & : & [A : \tau, \overline{B_i : \tau_i}] \rightarrow [\overline{B_i : \tau_i}] \\
\mu_\delta \vdash \mathit{flatten}\, d & : & \{\{\tau\}\} \rightarrow \{\tau\} & \mu_\delta \vdash \pi_{\overline{A_i}} & : & [\overline{A_i : \tau_i}] \rightarrow [\overline{A_i : \tau_i}\ |]
\end{array}$$

$$\begin{array}{rcl}
\mu_\delta \vdash \mathsf{brand\ A}\ (d) & : & \mu_\delta(\mathsf{A}) \rightarrow \mathit{Brand}\ \mathsf{A} \\
\mu_\delta \vdash\ !d & : & \mathit{Brand}\ \mathsf{A} \rightarrow \mu_\delta(\mathsf{A})
\end{array}$$

Note that record construction ($[A : d]$) and projection ($\pi_{\overline{A_i}}$) always return a closed record, and that field access ($d.A$) and field removal ($d{-}A$) work on both closed and open records. The typing rules for branding (resp. unbranding) takes as input (resp. returns) values of the type associated with its brand in μ_δ.

Given a well-formed model μ_δ, the type signatures of binary operators, written $\mu_\delta \vdash d_1 \otimes d_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ (i.e., operator $\otimes$ applied to values d_1 of type τ_1 and d_2 of type τ_2 has return type τ_3), are as follows:

$$\begin{array}{l}
\mu_\delta \vdash d_1 = d_2 : \tau \rightarrow \tau \rightarrow \texttt{BOOL} \\
\mu_\delta \vdash d_1 \in d_2 : \tau \rightarrow \{\tau\} \rightarrow \texttt{BOOL} \\
\mu_\delta \vdash d_1 \cup d_2 : \{\tau\} \rightarrow \{\tau\} \rightarrow \{\tau\} \\
\mu_\delta \vdash d_1 * d_2 : [\overline{A_i : \tau_{A_i}}\ |] \rightarrow [\overline{B_j : \tau_{B_j}}\ |] \rightarrow [\overline{A_i : \tau_{A_i}} * \overline{B_j : \tau_{B_j}}\ |] \\
\mu_\delta \vdash d_1 + d_2 : [\overline{A_i : \tau_{A_i}}\ |] \rightarrow [\overline{B_j : \tau_{B_j}}\ |] \rightarrow \{[\overline{A_i : \tau_{A_i}} + \overline{B_j : \tau_{B_j}}\ |]\} \\
\mu_\delta \vdash d_1 + d_2 : [\overline{A_i : \tau_{A_i}}] \rightarrow [\overline{B_j : \tau_{B_j}}] \rightarrow \{[\overline{A_i : \tau_{A_i}} + \overline{B_j : \tau_{B_j}}\ ..]\}\ \mathit{otherwise}
\end{array}$$

Note that record concatenation is well typed only for closed records, while record merge is well typed for both closed and open records. If both records in a merge are closed the type of its record output is closed, otherwise it is open.

We end this section with a theorem showing operators are total under typing conditions i.e., given value(s) of the correct input type(s), an operator always returns a value of the expected output type. The formulation relies on the evaluation judgments ($\oplus d \Downarrow d$ for unary operators and $d \otimes d \Downarrow d$ for binary operators) defined in Appendix A.

Theorem 3 (Typed Operators Consistency). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$, typing for unary and binary operators is consistent:*

$$\begin{array}{c}
\mathit{if}\ (\mu_\delta \vdash d : \tau) \wedge (\mu_\delta \vdash \oplus : \tau \rightarrow \tau') \\
\mathit{then}\ \exists d', (\oplus d \Downarrow d') \wedge (\mu_\delta \vdash d' : \tau')
\end{array}$$

$$\begin{array}{c}
\mathit{if}\ (\mu_\delta \vdash d_1 : \tau_1) \wedge (\mu_\delta \vdash d_2 : \tau_2) \wedge (\mu_\delta \vdash \otimes : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3) \\
\mathit{then}\ \exists d', (d_1 \otimes d_2 \Downarrow d') \wedge (\mu_\delta \vdash d' : \tau_3)
\end{array}$$

6 Business Rules Compiler

We now show how the proposed brand model and type system can be applied in the context of a business rules compiler. Due to space limitations, we focus on the changes from the version without brands from (Shinnar et al. 2015).

6.1 CAMP

The Calculus for Aggregating Matching Patterns was proposed in (Shinnar et al. 2015) to model the query fragment of production rules (Forgy 1981) with aggregation (Boyer and Mili 2011). That query fragment is a (nested) pattern matched against working memory. CAMP patterns scrutinize an implicit datum (**it**), in the context of an environment (**env**) mapping variables to data. Patterns may fail if they do not match the given data. Match failure, denoted **err**, is not fatal and can trigger alternative pattern matching attempts.

Definition 3 (CAMP Syntax).

$$\begin{aligned}\text{(patterns) } p ::= {} & d \mid \oplus p \mid p_1 \otimes p_2 \mid \textbf{map } p \mid \textbf{assert } p \mid p_1 \| p_2 \mid \textbf{it} \\ & \mid \textbf{let it} = p_1 \textbf{ in } p_2 \mid \textbf{env} \mid \textbf{let env} \mathrel{+}= p_1 \textbf{ in } p_2 \mid \textbf{cast } \mathsf{A}\end{aligned}$$

Definition 3 shows the syntax for CAMP with one expression added for casting. d returns a constant data. Unary ($\oplus$) and binary ($\otimes$) operators can be applied to the result of a pattern or patterns. **map** p maps a pattern p over the implicit data **it**. Assuming that **it** is a bag, the result is the bag of results obtained from matching p against each datum in **it**. Failing matches are skipped. The **assert** p construct allows a pattern p to conditionally cause match failure. If p evaluates to false, matching fails, otherwise, it returns the empty record $[\,]$. The $p_1 \| p_2$ construct allows for recovery from match failure: if p_1 matches successfully, p_2 is ignored; if p_1 fails to match, p_2 is evaluated. **it** returns the datum being matched. **let it** $= p_1$ **in** p_2 binds the implicit datum to the result of a pattern. **env** reifies the current environment as a record, which can then be manipulated via standard record operators. **let env** $+= p_1$ **in** p_2, adds new bindings to the environment. The result of matching p_1 must be a record, which is interpreted as a reified environment. If the current environment is compatible with the new one (all common attributes have equal values) they are merged and the pattern p_2 is evaluated with the merged environment. If they are incompatible, the pattern fails. Merge captures the standard semantics in rules languages where multiple bindings of the same variable must bind to the same value.

We add a new expression, **cast** A, which casts **it** to a given brand A and accounts for the specific failure semantics in CAMP. It returns the same value if it succeeds, and **err** if it fails. The semantics (resp. typing) for that extension of CAMP proceeds identically to the one described in (Shinnar et al. 2015), except that it takes a derivation hierarchy (resp. a model) as additional context. Instead of the evaluation judgment $\sigma \vdash p \mathbin{@} d \Downarrow d?$, we use $\delta; \sigma \vdash p \mathbin{@} d \Downarrow d?$, where δ is a derivation hierarchy, σ an environment binding variables to values, p a pattern, d an (implicit) input value, and $d?$ the successful match or a match failure. All the evaluation rules pass that additional context along, and the rules for casting are as follows.

$$\frac{\delta(\mathsf{A}', \mathsf{A})}{\delta; \sigma \vdash \textbf{cast}\ \mathsf{A} \mathbin{@} \mathsf{brand}\ \mathsf{A}'\ (d) \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d)}\ \text{(PCast)}$$

$$\frac{\neg\delta(\mathsf{A}', \mathsf{A})}{\delta; \sigma \vdash \textbf{cast}\ \mathsf{A} \mathbin{@} \mathsf{brand}\ \mathsf{A}'\ (d) \Downarrow \textbf{err}}\ \text{(PCastFail)}$$

Instead of the typing judgment $\Gamma \vdash p : \tau_0 \rightarrow \tau_1$, we use $\mu_\delta; \Gamma \vdash p : \tau_0 \rightarrow \tau_1$, where μ_δ is a well-formed model, Γ a type environment, p a pattern, τ_0 the type of the (implicit) input, and τ_1 the type of successful matches. All the typing rules pass that additional context along, and the rule for casting is as follows.

$$\frac{}{\mu_\delta; \Gamma \vdash \textbf{cast}\ \mathsf{A} : \mathit{Brand}\ \mathsf{A}' \rightarrow \mathit{Brand}\ \mathsf{A}}\ \text{(TPCast)}$$

Note that the dynamic semantics only requires δ which means evaluation can proceed entirely based on the brand name derivation with no need for structural checks. This suggests techniques for efficient representation and manipulation of objects should also apply in our context. The model μ_δ still needs to be passed at compile time as it is used in the typing rules for branding and unbranding operators (See Sect. 5). Type soundness holds for CAMP with brands.

Theorem 4 (Soundness of Type System for CAMP). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\begin{array}{c} \textit{if}\ (\mu_\delta; \Gamma \vdash p : \tau_0 \rightarrow \tau_1) \wedge (\mu_\delta \vdash \sigma : \Gamma) \wedge (\mu_\delta \vdash d_0 : \tau_0) \\ \textit{then}\ \exists d_1?, (\delta; \sigma \vdash p \mathbin{@} d_0 \Downarrow d_1?) \wedge (\mu_\delta \vdash d_1? : \tau_1) \end{array}$$

6.2 NRA

We use the NRA from (Shinnar et al. 2015) with one additional expression for casting. Other operators, e.g., ρ for unnesting, can be defined in terms of this core algebra (Cluet and Moerkotte 1993).

Definition 4 (NRA Syntax).

$$\begin{array}{rl} \text{(queries)}\ q ::= & d \mid \textsc{In} \mid \oplus q \mid q_1 \otimes q_2 \mid \chi_{\langle q_2 \rangle}(q_1) \mid \sigma_{\langle q_2 \rangle}(q_1) \\ \mid & q_1 \times q_2 \mid \bowtie^d{}_{\langle q_2 \rangle}(q_1) \mid q_1 \parallel q_2 \mid (\mathsf{A})\ q \end{array}$$

Here, d returns constant data, **In** returns the context value (usually a bag or a record), and $\oplus$ and $\otimes$ are the unary and binary operators from Sect. 5. χ is the map operation on bags, σ is selection, $\times$ is Cartesian product. The *dependent join*, $\bowtie^d$, evaluates q_2 with its context set to each value in the bag resulting from evaluating q_1, then concatenates records from q_1 and q_2 as in a Cartesian product. The $\parallel$ expression, which we call *default*, evaluates its first operand and returns its value, unless that value is $\emptyset$, in which case it returns the value of its second operand (as default).

$(\mathsf{A})\ q$, casts the result of q to a brand A. It returns a singleton bag containing that same value if the cast succeeds, and $\emptyset$ otherwise. The semantics (resp. typing) for the NRA proceeds identically to the one described in (Shinnar et al. 2015), except that it takes a derivation hierarchy (resp. a model) as additional context. Instead of the evaluation judgment $q\ @\ d \Downarrow d'$, we use $\delta \vdash q\ @\ d \Downarrow d'$, where δ is a derivation hierarchy, q an expression, d the (implicit) input value, and d' the output value. All the evaluation rules pass that additional context along, and the rules for casting are as follows.

$$\frac{\delta \vdash q\ @\ d_0 \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d) \quad \delta(\mathsf{A}', \mathsf{A})}{\delta \vdash (\mathsf{A})\ q\ @\ d_0 \Downarrow \{\mathsf{brand}\ \mathsf{A}'\ (d)\}}\ \text{(Cast)}$$

$$\frac{\delta \vdash q\ @\ d_0 \Downarrow \mathsf{brand}\ \mathsf{A}'\ (d) \quad \neg\delta(\mathsf{A}', \mathsf{A})}{\delta \vdash (\mathsf{A})\ q\ @\ d_0 \Downarrow \{\}}\ \text{(CastFail)}$$

Instead of the typing judgment $q : \tau_0 \to \tau_1$, we use $\mu_\delta \vdash q : \tau_0 \to \tau_1$, where μ_δ is a well-formed model, q an expression, τ_0 the type of the (implicit) input, and τ_1 the type of the output. All the typing rules pass that additional context along, and the rule for casting is as follows.

$$\frac{\mu_\delta \vdash q : \tau \to \mathit{Brand}\ \mathsf{A}'}{\mu_\delta \vdash (\mathsf{A})\ q : \tau \to \{\mathit{Brand}\ \mathsf{A}\}}\ \text{(TCast)}$$

Type soundness holds for NRA with brands.

Theorem 5 (Soundness of Type System for NRA). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\mathit{if}\ (\mu_\delta \vdash q : \tau_0 \to \tau_1) \wedge (\mu_\delta \vdash d_0 : \tau_0)\ \ \mathit{then}\ \exists d_1, (\delta \vdash q\ @\ d_0 \Downarrow d_1) \wedge (\mu_\delta \vdash d_1 : \tau_1)$$

6.3 From CAMP to NRA

Translation from CAMP to NRA proceeds identically as the one in (Shinnar et al. 2015), with one additional rule for the cast pattern:

$$[\![\mathbf{cast}\ \mathsf{A}]\!] = (\mathsf{A})\ (\textsc{In}.D)$$

We now restate the key correctness theorems of semantics and type preservation for that translation. A CAMP pattern that evaluates to d becomes an NRA query that evaluates to $\{d\}$ and a CAMP pattern that evaluates to **err** becomes an NRA query that evaluates to $\emptyset$.

Theorem 6 (Correctness of Translation from CAMP to NRA).

$$\begin{aligned} \delta; \sigma \vdash p\ @\ d_1 \Downarrow d_2 &\iff \delta \vdash [\![p]\!]\ @\ ([E : \sigma] * [D : d_1]) \Downarrow \{d_2\} \\ \delta; \sigma \vdash p\ @\ d_1 \Downarrow \mathbf{err} &\iff \delta \vdash [\![p]\!]\ @\ ([E : \sigma] * [D : d_1]) \Downarrow \emptyset \end{aligned}$$

Theorem 7 (Type Preservation). *Given a well-formed model, i.e., μ_δ such that $\models \mu_\delta$:*

$$\texttt{CAMP}{\leftrightarrow}\texttt{NRA}: \quad \mu_\delta; \Gamma \vdash p : \tau_0 \to \tau_1 \Leftrightarrow \mu_\delta \vdash [\![p]\!] : [E : \Gamma, D : \tau_0] \to \{\tau_1\}\,.$$

7 Related Work

Our work is related to several areas of databases and programming languages research which have all been influenced by Philip Wadler's work. Compiling rules to a database backend shares similarities with database-supported execution of programming languages with embedded queries, such as Links (Cooper et al. 2007) and others (Cheney et al. 2013). The Nested Relational Algebra is closely related to languages based on comprehensions such as (Trinder and Wadler 1988), and others (Tannen et al. 1992; Beeri and Kornatzky 1993; Cluet and Moerkotte 1993; Fegaras and Maier 2000). The idea to use of type annotations with a structural subtyping relation to marry nominal and structural typing is inspired by (Siméon and Wadler 2003) and others (Lee et al. 2015; Jones et al. 2015). Compared to (Siméon and Wadler 2003) we must account for the different data model which includes bags and records. This allows us to shed some of the complexity inherent to XML and XML Schema, such as derivation by restriction or substitution groups. However, the subtyping relation requires specific care to handle complex record operations. Compared to (Jones et al. 2015) and (Lee et al. 2015), we do not address typing for a full OO programming language but we integrate brands into a language suitable for bulk data processing.

Typing issues similar to ours are found in object-oriented databases, which support queries over nested objects (Berler et al. 2000). Implementations of OQL (Cluet and Moerkotte 1993) also compile queries into a nested algebra or calculus (Fegaras and Maier 2000; Trigoni and Bierman 2001; Cluet and Moerkotte 1993; Beeri and Kornatzky 1993; Claußen et al. 1997). At the surface language level, proposed type systems (Alagic 1999; Trigoni and Bierman 2001) rely on a purely nominal approach. In contrast, the type system for underlying algebras used for optimization (Cluet and Moerkotte 1993) involves structural typing and record operations but does not address inheritance. The approach presented here unifies both those type systems in a simple and natural way, allowing us to prove type-preservation for the resulting compiler.

8 Conclusion

In this paper, we have described a novel type system for languages that involve complex record operations in the context of a class hierarchy. Our compiler includes a translator from JRules to CAMP, a backend that generates Javascript code from NRA, and a query optimizer. Brands are integrated in the full pipeline and, with the exception of the initial front-end translation and Javascript generation, it has been fully mechanized and verified using the Coq proof assistant.

We are currently pursuing further development on both theoretical and practical aspects of our compiler. From a theoretical standpoint, we are exploring type inference and extensions with intersection types. From a practical standpoint, we are looking into the use types for optimization, and code-generation for various database runtimes. Beyond his work on XML types, we are indebted to Phil Wadler's long held interest in cross pollination between programming languages and databases without which this work would simply not exist.

Acknowledgments. We want to thank Joshua Auerbach, Martin Hirzel, and Louis Mandel, for their feedback on earlier versions of this draft.

A Operators Semantics

Figure 7 provides a denotational semantics for core unary operators using the judgment $\oplus d_1 \Downarrow d_2$ where $\oplus$ is the operator, d_1 the value it is applied to and d_2 the resulting value. Note that when writing e.g., $\{d\} \Downarrow \{d\}$, the left hand-side is really the application of the operator on d. I.e., it is really meant as $\lambda x.\{x\}(d) \Downarrow \{d\}$.

$$\frac{}{\mathit{ident}\ d \Downarrow d}\ (\text{Ident}) \qquad \frac{}{\{d\} \Downarrow \{d\}}\ (\text{Coll})$$

$$\frac{}{\mathit{flatten}(\{\}) \Downarrow \{\}}\ (\text{Flatten}_\emptyset) \qquad \frac{\mathit{flatten}\ (d_0) \Downarrow d_0'}{\mathit{flatten}\ (\{\{\overline{d_i}\}\} \cup d_0) \Downarrow \{\overline{d_i}\} \cup d_0'}\ (\text{Flatten}_\cup)$$

$$\frac{}{\neg\mathsf{true} \Downarrow \mathsf{false}}\ (\text{Not}_T) \qquad \frac{}{\neg\mathsf{false} \Downarrow \mathsf{true}}\ (\text{Not}_F)$$

$$\frac{}{[A\!:\!d] \Downarrow [A\!:\!d]}\ (\text{Rec}) \qquad \frac{d = [A : d_0; B_1 : d_1 ... B_n : d_n]}{d.A \Downarrow d_0}\ (\text{Field})$$

$$\frac{d = [A : d_0; B_1 : d_1; ... B_n : d_n]}{d - A \Downarrow [B_1 : d_1 ... B_n : d_n]}\ (\text{Remove})$$

$$\frac{d = [A_1 : d_1; ... A_n : d_n; B_1 : d_1'; ... B_n : d_n']}{\pi_{\overline{A_i}}(d) \Downarrow [A_1 : d_1 ... A_n : d_n]}\ (\text{Project})$$

$$\frac{}{\mathsf{brand\ A}\ (d) \Downarrow \mathsf{brand\ A}\ (d)}\ (\text{Brand}) \qquad \frac{d = \mathsf{brand\ A}\ (d')}{!d \Downarrow d'}\ (\text{Unbrand})$$

Fig. 7. Unary operators semantics. $\boxed{\oplus d \Downarrow d}$

Figure 8 provides a denotational semantics for core binary operators using the judgment $d_1 \otimes d_2 \Downarrow d_3$ where $\otimes$ is the operator, d_1 and d_2 the values it is applied to and d_3 the resulting value. Note that for concat and merge operators we rely on underlying merge and concat defined at the data model level (see Sect. A).

$$\frac{d_1 \doteq d_2}{d_1 = d_2 \Downarrow \mathsf{true}}\ (\text{Eq}) \qquad \frac{\neg d_1 \doteq d_2}{d_1 = d_2 \Downarrow \mathsf{false}}\ (\neg\ \text{Eq})$$

$$\frac{}{d \in \{\} \Downarrow \mathsf{false}}\ (\text{In}_\emptyset) \qquad \frac{d \doteq d_1}{d \in \{d_1\} \cup d_2 \Downarrow \mathsf{true}}\ (\text{In}_\cup)$$

$$\frac{\neg d \doteq d_1 \quad d \in d_2 \Downarrow d'}{d \in \{d_1\} \cup d_2 \Downarrow d'}\ (\neg\ \text{In}_\cup) \qquad \frac{}{\{\overline{d_i}\} \cup \{\overline{d'_i}\} \Downarrow \{\overline{d_i}; \overline{d'_i}\}}\ (\cup)$$

$$\frac{}{[\overline{A_i : d_i}] * [\overline{A'_i : d'_i}] \Downarrow [\overline{A_i : d_i} * \overline{A'_i : d'_i}]}\ (\text{Concat})$$

$$\frac{}{[\overline{A_i : d_i}] + [\overline{A'_i : d'_i}] \Downarrow [\overline{A_i : d_i} + \overline{A'_i : d'_i}]}\ (\text{Merge})$$

Fig. 8. Binary operators semantics. $\boxed{d \otimes d \Downarrow d}$

References

Alagic, S.: Type-checking OQL queries in the ODMG type systems. ACM Trans. Database Syst. **24**(3), 319–360 (1999)

Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark SQL: relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394 (2015)

Arnold, M., Grove, D., Herta, B., Hind, M., Hirzel, M., Iyengar, A., Mandel, L., Saraswat, V.A., Shinnar, A., Siméon, J., Takeuchi, M., Tardieu, O., Zhang, W.: META: middleware for events, transactions, and analytics. IBM J. Res. Devel. (IBMRD). (2016, to appear)

Bali, M.: Drools JBoss Rules 5.0 Developer's Guide. Packt Publishing, Birmingham (2009)

Beeri, C., Kornatzky, Y.: Algebraic optimization of object-oriented query languages. Theor. Comput. Sci. **116**(1&2), 59–94 (1993)

Berler, M., Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann, San Francisco (2000)

Boyer, J., Mili, H.: IBM websphere ILOG JRules. In: Boyer, J., Mili, H. (eds.) Agile Business Rule Development, pp. 215–242. Springer, Heidelberg (2011)

Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: International Conference on Functional Programming (ICFP), pp. 403–416 (2013)

Claußen, J., Kemper, A., Moerkotte, G., Peithner, K.: Optimizing queries with universal quantification in object-oriented and object-relational databases. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), pp. 286–295 (1997)

Cluet, S., Moerkotte, G.: Nested queries in object bases. In: Workshop on Database Programming Languages (DBPL), pp. 226–242 (1993)

Cooper, E., Lindley, S., Yallop, J.: Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)

Fegaras, L., Maier, D.: Optimizing object queries using an effective calculus. ACM Trans. Database Syst. (TODS) **25**(4), 457–516 (2000)

Forgy, C.L.: OPS5 user's manual. Technical report. 2397, CMU (1981)

Green, T.J., Aref, M., Karvounarakis, G.: LogicBlox, platform and language: a tutorial. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 1–8. Springer, Heidelberg (2012)

Grust, T., Mayr, M., Rittinger, J., Schreiber, T.: Ferry: Database-supported program execution. In: International Conference on Management of Data (SIGMOD), pp. 1063–1066 (2009)

Jones, T., Homer, M., Noble, J.: Brand objects for nominal typing. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic, pp. 198–221 (2015)

Lee, J., Aldrich, J., Shaw, T., Potanin, A.: A theory of tagged objects. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic (2015)

May, N., Helmer, S., Moerkotte, G.: Strategies for query unnesting in XML databases. Trans. Database Syst. (TODS) **31**(3), 968–1013 (2006)

Meijer, E., Beckman, B., Bierman, G.: Linq: reconciling object, relations and XML in the.NET framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 706–706 (2006)

Shinnar, A., Siméon, J., Hirzel, M.: A pattern calculus for rule languages: expressiveness, compilation, and mechanization. In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic, pp. 542–567 (2015)

Siméon, J., Wadler, P.: The essence of XML. In: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, 15–17 January 2003, pp. 1–13 (2003)

Tannen, V., Buneman, P., Wong, L.: Naturally embedded query languages. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 140–154. Springer, Heidelberg (1992)

Trigoni, A., Bierman, G.: Inferring the principal type and the schema requirements of an OQL query. In: Read, B. (ed.) BNCOD 2001. LNCS, vol. 2097, pp. 185–201. Springer, Heidelberg (2001)

Trinder, P., Wadler, P.: List comprehensions and the relational calculus. In: Proceedings of 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland, pp. 115–123, August 1988

The Recursive Union of Some Gradual Types

Jeremy G. Siek(✉) and Sam Tobin-Hochstadt

Indiana University, Bloomington, IN 47405, USA
jsiek@indiana.edu
http://homes.soic.indiana.edu/jsiek/

Abstract. We study union types and recursive types in the setting of a gradually typed lambda calculus. Our goal is to obtain a foundational account for languages that enable recursively defined data structures to be passed between static and dynamically typed regions of a program. We discuss how traditional sum types are not appropriate for this purpose and instead study a form of "true" union in the tradition of soft typing (Cartwright and Fagan, 1991) and occurrence typing (Tobin-Hochstadt and Felleisen, 2008). Regarding recursive types, our formulation is based on the axiomatization of subtyping by Brand and Henglein (1998).

This paper defines three artifacts. First, in the context of the simply typed lambda calculus, we define the semantics of our unions and integrate them with equi-recursive types. Second, we add a dynamic type $\star$ to obtain a gradually typed lambda calculus. Its semantics is defined by translation to the third artifact, a blame calculus (Wadler and Findler, 2009) extended with unions and equi-recursive types.

1 Introduction

The following program is gradually typed, with a dynamically typed region enclosed in $\lceil\cdot\rceil$. The dynamically typed code creates a list of two integers and passes the list to the statically-typed *sum* function.

$$
\begin{array}{l}
\texttt{letrec}\ \mathit{sum} = \lambda \mathit{ls} : \texttt{IntList}.\ \texttt{case}\ \times?(\mathit{ls})\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda x : \texttt{Int} \times \texttt{IntList}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{fst}\ x + \mathit{sum}(\texttt{snd}\ x) \\
\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \lambda y : \texttt{Unit}.\ 0 \\
\texttt{in}\ \mathit{sum}\ \lceil \mathit{cons}\ 1\ (\mathit{cons}\ 2\ \mathit{nil}) \rceil
\end{array}
$$

The `case` construct queries for the category of a value, e.g., whether it is a pair. The runtime semantics of this construct is intended to have the same power as value inspection primitives in languages like Scheme (Kelsey et al., 1998) and Racket (Flatt and PLT, 2014) (e.g. `procedure?`, `pair?`, etc.), and similar to those in other dynamic languages such as Python or JavaScript. We seek a foundational account of a language in which the above program can be written. The literature includes several candidate languages from which we could start.

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 388–410, 2016.
DOI: 10.1007/978-3-319-30936-1_21

The language ℂDuce (Frisch et al., 2008, Castagna et al., 2014) includes recursive types, union types, and more. However, the dynamic type dispatch feature of ℂDuce, of the form $x = L \in A \mathbin{?} M \mid N$, does not quite match our needs. (Here L, M, N are terms and A is a type). The type dispatch of ℂDuce requires more power (i.e., is more expensive) because it relies on type checking values in a deep manner:

$$(x = V \in A \mathbin{?} M \mid N) \longrightarrow \begin{cases} [x{:=}V]M & \text{if } \vdash V : A \\ [x{:=}V]N & \text{if } \vdash V : \neg A \end{cases}$$

The set-theoretic union studied by Pierce (1991) and others comes to mind, but the `case` construct from that line of research does not enable runtime dispatching but instead requires the body of the case to be well typed for each alternative type in the union:

$$\frac{\Gamma \vdash L : A_1 \cup A_2 \qquad \Gamma, x{:}A_i \vdash N : B \quad i \in 1,2}{\Gamma \vdash (\mathtt{case}\, L\, \mathtt{of}\; x \Rightarrow N) : B}$$

Alternatively, one might try to use sum types (Pierce, 2002) and define the above `IntList` type as follows.

$$\mathtt{IntList} \triangleq \mu X.\, \mathtt{Unit} + (\mathtt{Int} \times X)$$

However, sum types do not interoperate well with the dynamic type $\star$ of gradual typing. For example, what should the following sequence of casts reduce to?

$$4 : \mathtt{Int} \Longrightarrow \star \Longrightarrow (\mathtt{Int} + \mathtt{Int})$$

Should it be $\mathtt{inl}\, 4$ or $\mathtt{inr}\, 4$? There is not a good answer which suggests that one should not ask the question. To solve this, one could require the branches of the sums to be different types. In some sense this is what we propose to do in this paper, but with the following caveat. We require the branches of the sum to differ in their top-most type constructor, for example, disallowing the sum type

$$(\mathtt{Int} \to \mathtt{Int}) + (\mathtt{Int} \to \mathtt{Bool})$$

With this restriction, the value inspection primitives alluded to above are powerful enough to choose between the branches of the sum.

The `case` construct and union type in this paper can also be seen as a modern take on the `case` and union type of Cartwright and Fagan (1991). Our formalism differs from theirs in that we give a succinct typing rule for `case`, encode all of our restrictions on union and recursive types in the grammar of types, and provide syntax-directed subtyping rules that are more suggestive of a subtyping algorithm. Recently, Tobin-Hochstadt and Felleisen (2008, 2010) studied union types and proposed a type system that supports *occurrence typing* based on predicates and Boolean conditionals. The Typed Racket implementation supports recursive types but its combination of recursive types and union types has not been formalized. Here we study a `case` construct that is simpler than the one in Typed Racket, one that is more appropriate for a foundational calculus rather than a full-fledged programming language.

The contributions of this paper are threefold.

– Sect. 2 defines a simply typed lambda calculus with recursive types, union types, and a `case` construct for discriminating on the category of a value ($\lambda_{\mu\cup}$). Our subtyping rules for unions are inspired by Vouillon (2004) whereas our handling of recursive types is inspired by Brandt and Henglein (1998). However, the combined system is novel.
– Sect. 3 defines a gradually typed lambda calculus with recursive types and union types ($\lambda^{\star}_{\mu\cup}$). This section includes both a declarative type system and an algorithmic type system as well as a cast-inserting translation to a blame calculus.
– Sect. 4 presents a blame calculus with unions and recursive types ($\lambda\mathsf{B}_{\mu\cup}$).

2 STLC with Recursive Types and Unions, $\boldsymbol{\lambda_{\mu\cup}}$

Every gradual type system needs a static type system to be gradual with respect to. Thus, we begin with the Simply Typed Lambda Calculus (STLC) extended with union types and equi-recursive types, which we refer to has $\lambda_{\mu\cup}$. We discuss unions first and then the recursive types.

2.1 Union Types and Case

The `case` construct that we study in this paper has the form

$$\texttt{case}\,c?(L)\,\texttt{then}\,M\,\texttt{else}\,N$$

where c ranges over the type constructors (`Int`, $\to$, $\times$, etc.). The dynamic semantics of `case` is to test whether the value V of term L matches the type constructor c. If it does, apply M to V, and if not, apply N to V.

$$\texttt{case}\,c?(V)\,\texttt{then}\,M\,\texttt{else}\,N \longrightarrow \begin{cases} M\ V & \text{if } \mathit{tycon}(V) = c \\ N\ V & \text{otherwise} \end{cases}$$

The *tycon* function is defined as follows:

$$\mathit{tycon}(k) = \Delta(k) \qquad \mathit{tycon}((V_1, V_2)) = \times \qquad \mathit{tycon}(\lambda x{:}A.\ N) = \ \to$$

where Δ assigns base types to constants.

The above `case` construct is the elimination form of our union type. Given a union type, the `then` branch needs *the* element of the union type that matches the type constructor c. The `else` branch needs all of the elements of the union type that do not match c. Our union types therefore take the form

$$\cup\mathcal{C}$$

where $\mathcal{C}$ is a function from type constructors to pre-types. We require that the pre-type $\mathcal{C}(c)$ have c as its top-most type constructor. We use the notation $\mathcal{C} - c$

for the function whose domain has been restricted to omit c. The following is the typing rule for `case`.

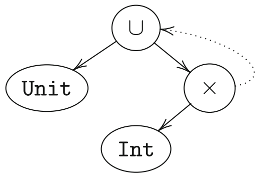
$$\frac{\Gamma \vdash L : \cup\mathcal{C} \qquad \Gamma \vdash M : \mathcal{C}(c) \to A \qquad \Gamma \vdash N : (\cup(\mathcal{C} - c)) \to A}{\Gamma \vdash (\texttt{case}\, c?(L)\ \texttt{then}\ M\ \texttt{else}\ N) : A}$$

The typing rule makes it clear why the two branches have function type: their parameter types take into account the knowledge learned about L, so our `case` can be viewed as supporting a limited form of occurrence typing.

The introduction rule for our union type is subsumption, that is, terms may be implicitly up-cast via subtyping to have a union type. The subtyping rules for unions, shown below, are inspired by Vouillon (2004). If the left-hand side is a union, then we require every type in the union to be a subtype of the right-hand side. If the left-hand side is not a union, but the right-hand side is, then the left-hand side needs to be a subtype of one of the types in the union.

$$\frac{\forall c \in \mathrm{dom}(\mathcal{C}).\ \Sigma \vdash \mathcal{C}(c) <: Q}{\Sigma \vdash \cup\mathcal{C} <: Q} \qquad \frac{\Sigma \vdash c(\bar{A}) <: \mathcal{C}(c) \qquad c \in \mathrm{dom}(\mathcal{C})}{\Sigma \vdash c(\bar{A}) <: \cup\mathcal{C}}$$

2.2 Equi-Recursive Types



Equi-Recursive types are best thought of as trees with back edges. For example, the tree on the right is the `IntList` type of the earlier example. To conveniently define subtyping rules, we define a syntax for types in Fig. 1. We choose a particular canonical form that streamlines the subtyping rules. Every node in the tree has two syntactic parts: (1) a μ binder so that it can be the target of a back edge and (2) a pre-type that describes its shape: either a union type or a type constructor with arguments. A back edge is represented using an occurrence of a type variable X.

The `IntList` type is therefore represented syntactically as follows:

$$\texttt{IntList} \equiv \mu X.\ \cup\{\texttt{Unit} \mapsto \texttt{Unit}, \times \mapsto (\mu Y.\ \texttt{Int}) \times X\}$$

As usual, we often write the application of a type constructor using infix notation (i.e., $A_1 \times A_2$ is $\times(A_1, A_2)$, $A_1 \to A_2$ is $\to(A_1, A_2)$, and ι is $\iota()$). The canonical form requires μ binders in many places in which they are not used, as in $\mu Y.$ `Int`, but this is harmless.

The definition of subtyping in the presence of equi-recursive types is delicate and is often expressed using coinduction (Pierce, 2002). Here we define subtyping using an inductive definition that mimics coinduction through the use of subtyping assumptions, an idea due to Brandt and Henglein (1998). The resulting subtyping definition is simpler than the classic one of Amadio and Cardelli (1993) in that it does not require a separate definition of type equality to allow for the unfolding of recursive types and it does not require the subtle "contract" rule. The basic idea is that when trying to deduce that $A <: B$, one can assume

Base types	ι	$::= \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid \cdots$
Constructors	$c \in \mathbb{C}$	$::= \iota \mid \times \mid \rightarrow$
Cnstr. to type	$\mathcal{C}$	$::= \{c \mapsto c(\bar{A}), \ldots\}$
Pre-types	P, Q	$::= c(\bar{A}) \mid \cup\mathcal{C}$
Types	A, B, C	$::= \mu X.\ P \mid X$

Fig. 1. Types for the STLC with recursive types and unions

$$\text{Subtype Env.} \qquad \Sigma ::= \emptyset \mid \Sigma, A <: B$$

Subtyping on pre-types $\boxed{\Sigma \vdash P <: Q}$

$$\frac{}{\Sigma \vdash \iota <: \iota} \qquad \frac{\Sigma \vdash A <: C \quad \Sigma \vdash B <: D}{\Sigma \vdash A{\times}B <: C{\times}D} \qquad \frac{\Sigma \vdash C <: A \quad \Sigma \vdash B <: D}{\Sigma \vdash A{\rightarrow}B <: C{\rightarrow}D}$$

$$\frac{\forall c \in \mathrm{dom}(\mathcal{C}).\ \Sigma \vdash \mathcal{C}(c) <: Q}{\Sigma \vdash \cup\mathcal{C} <: Q} \qquad \frac{\Sigma \vdash c(\bar{A}) <: \mathcal{C}(c) \quad c \in \mathrm{dom}(\mathcal{C})}{\Sigma \vdash c(\bar{A}) <: \cup\mathcal{C}}$$

Subtyping $\boxed{\Sigma \vdash A <: B}$

$$\frac{A <: B \in \Sigma}{\Sigma \vdash A <: B} \qquad \frac{\begin{array}{c} A = \mu X.\ P \qquad B = \mu Y.\ Q \\ \Sigma, A <: B \vdash [X{:=}A]P <: [Y{:=}B]Q \end{array}}{\Sigma \vdash A <: B}$$

Fig. 2. Subtyping for recursive types and unions.

$A <: B$. The trick to avoid circular reasoning is to make sure not to use the assumption right away, but only after progressing past a type constructor.

We accomplish this restriction in a slightly more compact way than Brandt and Henglein (1998). We separate the subtyping relation into two mutually recursive relations in Fig. 2, one on pre-types and the other on types. Assumptions may only be used for subtyping on types (not pre-types) and the body of a recursive type is a pre-type. The subtyping rules for pre-types are the usual ones for base types, products, and functions together with the above-described rules for unions. The rules for (recursive) types enable the use of assumptions or unfold the two recursive types, adding them to the set of assumptions. We abbreviate subtyping in an empty environment as $A <: B$. As usual, subtyping is reflexive and transitive.

The subtyping rules are syntax directed and therefore suggestive of an algorithm for subtyping. To obtain an efficient algorithm, one should thread the subtyping environment as both an input and an output and use structure sharing so that structural type equality checks can be implemented as pointer equality.

2.3 Type System

Figure 3 defines the type system for $\lambda_{\mu\cup}$. The standard rules for the STLC are in gray so as to emphasize the rules concerning recursive types and unions. Our syntax for types induces a slight inconvenience; we split the typing relation into two parts to handle whether the term's type is a pre-type or a recursive type. The introduction rules are generally in the typing relation for pre-types whereas the elimination rules typically produce types. There are two rules that convert back and forth between recursive types and pre-types. A recursive type can be turned into a pre-type by unfolding. A pre-type can be turned into a recursive type by folding it into the recursive type.

2.4 Operational Semantics

The operational semantics for $\lambda_{\mu\cup}$ is also given in Fig. 3. Except for the reduction rule for `case`, which we discussed in Sect. 2.1, the rules are standard. The type system of $\lambda_{\mu\cup}$ is sound with respect to the operational semantics. The proof is in Appendix A.1.

3 GTLC with Recursive Types and Unions, $\lambda^{\star}_{\mu\cup}$

This section defines a gradually typed lambda calculus equipped with equi-recursive types and unions, $\lambda^{\star}_{\mu\cup}$. We define the static semantics (that is, the type system) for $\lambda^{\star}_{\mu\cup}$ in Sect. 3.1 and then present a type checking algorithm in the form of a syntax-directed inductive definition (Sect. 3.2). The dynamic semantics, as usual for gradually typed languages, is defined by translation to a blame calculus. That translation is given in Sect. 3.3. (This blame calculus is defined in Sect. 4.)

3.1 Type System

We define the type system for $\lambda^{\star}_{\mu\cup}$ following the methodology for modifying a statically typed language, in this case $\lambda_{\mu\cup}$, to be gradually typed. Recent work by Cimini and Siek (2016) formalizes this methodology and provides a tool for automating the creation of gradual type systems, but does not yet handle types with binders, such as recursive or universal types. So here we proceed by hand.

We add the dynamic type $\star$ and change the elimination rules to use *matching* to handle the case when what is being eliminated has type $\star$. (The introduction rules remain unchanged.) We write $P \triangleright Q$ to say that pre-type P matches Q and define this relation as follows.

$$c(\bar{A}) \triangleright c(\bar{A}) \qquad \cup\mathcal{C} \triangleright \cup\mathcal{C} \qquad \star \triangleright \star \times \star \qquad \star \triangleright \star \to \star \qquad \star \triangleright \cup\{c \mapsto c(\bar{\star}) \mid c \in \mathbb{C}\} \qquad \boxed{P \triangleright Q}$$

We replace uses of type equality with either compatibility (e.g., the application rule) or by taking the meet with respect to naive subtyping (e.g., the `case` rule). We extend the usual definition of naive subtyping (Wadler and Findler, 2009) to handle unions and recursive types in a straightforward way:

$$P <:_n \star \qquad \frac{\forall i.\ A_i <:_n B_i}{c(\bar{A}) <:_n c(\bar{B})} \qquad \frac{\mathrm{dom}(C) = \mathrm{dom}(C') \quad \forall c \in \mathrm{dom}(C).\ \mathcal{C}(c) <:_n \mathcal{C}'(c)}{\cup\mathcal{C} <:_n \cup\mathcal{C}'} \qquad \boxed{P <:_n Q}$$

$$\frac{P <:_n Q}{\mu X.\ P <:_n \mu X.\ Q} \qquad X <:_n X \qquad \boxed{A <:_n B}$$

$$\boxed{L, M, N}$$

Terms	L, M, N	$::= k \mid op(\bar{M}) \mid \lambda x{:}A.\ N \mid L\ N \mid (M, N) \mid \texttt{fst}\, L \mid \texttt{snd}\, L \mid$ $\texttt{case}\, c?(L)\ \texttt{then}\ M\ \texttt{else}\ N$
Type Env.	Γ	$::= \emptyset \mid \Gamma, x{:}A$
Values	V, W	$::= k \mid \lambda x{:}A.\ N \mid (V, W)$
Frames	F	$::= \square\ N \mid V\ \square \mid (\square, N) \mid (V, \square) \mid \texttt{fst}\, \square \mid \texttt{snd}\, \square \mid$ $\texttt{case}\, c?(\square)\ \texttt{then}\ M\ \texttt{else}\ N$

Term typing $\boxed{\Gamma \vdash M : P}$

$$\frac{}{\Gamma \vdash k : \Delta(k)} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma, x{:}A \vdash N : B}{\Gamma \vdash (\lambda x{:}A.\ N) : A \to B}$$

$$\frac{\Gamma \vdash M : \mu X.\ P}{\Gamma \vdash M : [X{:=}\mu X.\ P]P}$$

$$\boxed{\Gamma \vdash M : A}$$

$$\frac{\Gamma \vdash M : A \quad \emptyset \vdash A <: B}{\Gamma \vdash M : B} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Delta(op) = (\bar{A}, B) \quad \Gamma \vdash \bar{M} : \bar{A}}{\Gamma \vdash op(\bar{M}) : B}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{fst}\, M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{snd}\, M : B} \qquad \frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : \cup\mathcal{C} \quad \Gamma \vdash M : \mathcal{C}(c) \to A \\ \Gamma \vdash N : (\cup(\mathcal{C} - c)) \to A\end{array}}{\Gamma \vdash (\texttt{case}\, c?(L)\ \texttt{then}\ M\ \texttt{else}\ N) : A} \qquad \frac{\Gamma \vdash M : [X{:=}\mu X.\ P]P}{\Gamma \vdash M : \mu X.\ P}$$

Reduction

$$\begin{aligned}
op(\bar{V}) &\longrightarrow \delta(op, \bar{V}) \\
(\lambda x{:}A.\ N)\ V &\longrightarrow [x{:=}V]N \\
\texttt{fst}\, (V, W) &\longrightarrow V \\
\texttt{snd}\, (V, W) &\longrightarrow W \\
\texttt{case}\, c?(V)\ \texttt{then}\ M\ \texttt{else}\ N &\longrightarrow \begin{cases} M\ V & \text{if } \mathit{tycon}(V) = c \\ N\ V & \text{otherwise} \end{cases} \\
F[M] &\longrightarrow F[N] \quad \text{if } M \longrightarrow N
\end{aligned}$$

Fig. 3. Semantics of STLC with recursive types and unions, $\lambda_{\mu\cup}$

Two types are *compatible*, written $A \sim B$, when there exists a lower bound of the two types with respect to naive subtyping. The *meet*, written $A \sqcap B$, is the greatest lower bound with respect to naive subtyping.

Finally, because $\lambda_{\mu\cup}$ supports subtyping, we use the approach of Siek and Taha (2007) which recommends adding a subsumption rule to the gradual type system and defining subtyping for $\star$ as follows.

$$\Sigma \vdash \star <: \star$$

With these ingredients in place, we present the static semantics of $\lambda^{\star}_{\mu\cup}$ in Fig. 4.

$$\begin{array}{lll}
\text{Pre-types} & P, Q & ::= c(\bar{A}) \mid \cup\mathcal{C} \mid \star \\
\text{Terms} & L, M, N & ::= k \mid op(\bar{M}) \mid \lambda x{:}A.\ N \mid L\ N \mid (M, N) \mid \texttt{fst}\ L \mid \texttt{snd}\ L \mid \\
& & \quad \texttt{case}\ c?(L)\ \texttt{then}\ M\ \texttt{else}\ N \mid M :: A
\end{array}$$

Term typing $\boxed{\Gamma \vdash M : P}$

identical to $\lambda_{\mu\cup}$

$\boxed{\Gamma \vdash M : A}$

$$\frac{\Gamma \vdash M : A \qquad \emptyset \vdash A <: B}{\Gamma \vdash M : B} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Delta(op) = \bar{A} \to B \qquad \Gamma \vdash \bar{M} : \bar{C} \qquad \forall i \in 1..n.\ C_i \sim A_i}{\Gamma \vdash op(\bar{M}) : B}$$

$$\frac{\Gamma \vdash M : P \qquad P \rhd (B \times C)}{\Gamma \vdash \texttt{fst}\ M : B} \qquad \frac{\Gamma \vdash M : P \qquad P \rhd (B \times C)}{\Gamma \vdash \texttt{snd}\ M : C} \qquad \frac{\Gamma \vdash L : P \qquad P \rhd (B \to C) \qquad \Gamma \vdash M : B' \qquad B' \sim B}{\Gamma \vdash L\ M : C} \qquad \frac{\Gamma \vdash M : A \qquad A \sim B}{\Gamma \vdash (M :: B) : B}$$

$$\frac{\Gamma \vdash L : P \qquad P \rhd \cup\mathcal{C} \qquad \Gamma \vdash M : \mathcal{C}(c) \to C \qquad \Gamma \vdash N : (\cup(\mathcal{C} - c)) \to C'}{\Gamma \vdash (\texttt{case}\ c?(L)\ \texttt{then}\ M\ \texttt{else}\ N) : (C \sqcap C')} \qquad \frac{\Gamma \vdash M : P \qquad X \notin \text{ftv}(P)}{\Gamma \vdash M : \mu X.\ P}$$

Fig. 4. Type system for the GTLC with recursive types and union, $\lambda^{\star}_{\mu\cup}$

3.2 Algorithmic Type System

As was the case for Siek and Taha (2007), more work is required to obtain a type checking algorithm. As usual, we need to remove the subsumption rule and incorporate subtyping into the elimination rules (Pierce, 2002). However, the presence of matching and compatibility presents a challenge. For example, a naive attempt to incorporate subtyping into the application rule gives us:

$$\frac{\Gamma \vdash L : A \qquad A <: A' \qquad A' \rhd (B \to C) \qquad \Gamma \vdash M : B' \qquad B' <: B'' \qquad B'' \sim B}{\Gamma \vdash L\ M : C}\ \text{(Naive)}$$

This rule is still not syntax directed because of the types A' and B''. We introduce two new relations, *compatible-subtyping* ($\lesssim$) and *subtype-matching* ($\trianglerighteq$), that combine subtyping with compatibility and matching, respectively, in a syntax directed manner (defined in Fig. 5).

Lemma 11.

1. *$A \lesssim C$ iff $A <: B$ and $B \sim C$ for some B.*
2. *$A \trianglerighteq Q$ iff $A <: \mu X.\ P$ and $[X{:=}\mu X.\ P]P \rhd Q$ for some $\mu X.\ P$.*

The interaction of subtyping and the meet operator in the typing rule for `case` raises a similar problem. Given a `case` whose branches have type C and C', the type of the entire `case` needs to be some type A such that $A = B \sqcap B'$, $C <: B$, and $C' <: B'$. So we need a type that is like an upper bound of C and

Compatible-subtyping on pre-types $\boxed{\Sigma \vdash P \lesssim Q}$

Same as subtyping on pre-types, but replace <: with $\lesssim$ and add:

$$\frac{}{\Sigma \vdash \star \lesssim Q} \qquad \frac{}{\Sigma \vdash P \lesssim \star}$$

Compatible-Subtyping $\boxed{\Sigma \vdash A \lesssim B}$

Same as subtyping, but replace <: with $\lesssim$.

Subtype-matching $\boxed{P \trianglerighteq Q}$ $\boxed{A \trianglerighteq Q}$

$$c(\bar{A}) \trianglerighteq c(\bar{A}) \qquad \star \trianglerighteq \star \times \star \qquad \star \trianglerighteq \star \!\rightarrow \star \qquad \star \trianglerighteq \cup\{c \mapsto c(\bar{\star}) \mid c \in \mathbb{C}\}$$

$$\cup\mathcal{C} \trianglerighteq \cup\mathcal{C} \qquad \frac{\mathrm{dom}(\mathcal{C}) = \{c\}}{\cup\mathcal{C} \trianglerighteq \mathcal{C}(c)} \qquad \frac{[X{:=}\mu X.\ P]P \trianglerighteq Q}{\mu X.\ P \trianglerighteq Q}$$

Fig. 5. Compatible-subtyping and subtype-matching.

C' with respect to <: but a lower bound with respect to $<:_n$. To accomplish this we define the *meet-join* relation ($\triangledown$) and the accompanying *meet-meet* ($\vartriangle$) relation (Fig. 10 of the Appendix).

Lemma 12.

1. *$A \triangledown B = C$ iff $A <: A'$, $B <: B'$, and $C = A' \sqcap B'$ for some A' and B'.*
2. *$A \vartriangle B = C$ iff $A' <: A$, $B' <: B$, and $C = A' \sqcap B'$ for some A' and B'.*

Using these auxiliary definitions, we define a type checking algorithm for $\lambda^{\star}_{\mu\cup}$ in Fig. 6. This algorithm coincides with the type system.

Theorem 11. *$\Gamma \vdash M : A$ iff $\Gamma \vdash M \hookrightarrow A$*

3.3 Translation to the Blame Calculus via Cast Insertion

The dynamic semantics for $\lambda^{\star}_{\mu\cup}$ is defined in terms of the blame calculus $\lambda\mathsf{B}_{\mu\cup}$ defined in Sect. 4. That blame calculus is an extension of $\lambda_{\mu\cup}$ with an explicit cast construct of the form

$$M : A \xRightarrow{p} B$$

where A is the static type of M, p is a blame label, and B is the target type of the cast, which must be compatible with A. To indicate whether the blame for a cast failure goes to the inside of a cast or its context, each blame label p has a complement $\overline{p}$. Complement is involutive, $\overline{\overline{p}} = p$. Unlike the gradually typed $\lambda^{\star}_{\mu\cup}$, casts to and from $\star$ are explicit. However, because this blame calculus supports unions and recursive types, it supports implicit up-casts with respect to subtyping.

Algorithmic term typing $\boxed{\Gamma \vdash M \hookrightarrow A}$

$$\frac{X \notin \mathrm{ftv}(\Delta(k))}{\Gamma \vdash k \hookrightarrow \mu X.\ \Delta(k)} \qquad \frac{\Gamma \vdash M \hookrightarrow A \quad \Gamma \vdash N \hookrightarrow B \quad X \notin \mathrm{ftv}(A \times B)}{\Gamma \vdash (M, N) \hookrightarrow \mu X.\ A{\times}B}$$

$$\frac{\Gamma, x{:}A \vdash N \hookrightarrow B \quad X \notin \mathrm{ftv}(A \to B)}{\Gamma \vdash (\lambda x{:}A.\ N) \hookrightarrow \mu X.\ A{\to}B} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x \hookrightarrow A} \qquad \frac{\Delta(op) = (\bar{A}, B) \quad \forall i.\ \Gamma \vdash M_i \hookrightarrow C_i \quad C_i \lesssim A_i}{\Gamma \vdash op(\bar{M}) \hookrightarrow B}$$

$$\frac{\Gamma \vdash M \hookrightarrow A \quad A \trianglerighteq (B \times C)}{\Gamma \vdash \mathtt{fst}\, M \hookrightarrow B} \qquad \frac{\Gamma \vdash M \hookrightarrow A \quad A \trianglerighteq (B \times C)}{\Gamma \vdash \mathtt{snd}\, M \hookrightarrow C} \qquad \frac{\Gamma \vdash L \hookrightarrow A \quad A \trianglerighteq (B \to C) \quad \Gamma \vdash M \hookrightarrow B' \quad B' \lesssim B}{\Gamma \vdash L\ M \hookrightarrow C}$$

$$\frac{\Gamma \vdash L \hookrightarrow A \quad A \trianglerighteq \cup\mathcal{C} \quad \Gamma \vdash M \hookrightarrow \mathcal{C}(c) \to C \quad \Gamma \vdash N \hookrightarrow (\cup(\mathcal{C} - c)) \to C'}{\Gamma \vdash (\mathtt{case}\, c?(L)\ \mathtt{then}\ M\ \mathtt{else}\ N) \hookrightarrow (C \triangledown C')} \qquad \frac{\Gamma \vdash M \hookrightarrow A \quad A \lesssim B}{\Gamma \vdash (M :: B) \hookrightarrow B}$$

Fig. 6. Algorithmic type system for $\lambda^{\star}_{\mu\cup}$

The translation to $\lambda\mathsf{B}_{\mu\cup}$, defined in Fig. 7, turns the implicit casts to and from $\star$ into explicit casts but ignores the implicit upcasts from subtyping. The translation is type directed, so its structure is similar to the algorithmic type system of Fig. 6. We define two judgments

$$\Gamma \vdash M : P \rightsquigarrow M' \quad \text{and} \quad \Gamma \vdash M : A \rightsquigarrow M'$$

where M' is the translated term in the blame calculus. An important property of the translation is that it be type preserving, which we prove in Appendix A.3.

Lemma 13 (Cast Insertion Preserves Types). *If $\Gamma \vdash M : A \rightsquigarrow M'$ then $\Gamma \vdash M' : A$.*

Towards explaining the cast insertion translation, we first consider the translation of a type ascripted term $M :: B$. In Fig. 6, M has type A and we require $A \lesssim B$. Naively, we might insert a cast from A to B as follows:

$$\frac{\Gamma \vdash M : A \rightsquigarrow M' \qquad A \lesssim B}{\Gamma \vdash (M :: B) : B \rightsquigarrow (M' : A \overset{p}{\Longrightarrow} B)}$$

but the inserted casts requires that $A \sim B$ and we only know that $A \lesssim B$. That is, there may be an implicit upcast from A to B with respect to subtyping. Thus, the explicit cast to B needs start at some type A' such that $A <: A'$ and $A' \sim B$. Again taking inspiration from Siek and Taha (2007), we define a *merge* operator ($\rightharpoonup$) (defined in Fig. 8) that computes A'. The correct translation rule for type ascription uses merge and is defined in Fig. 7.

Translation (cast insertion) $\boxed{\Gamma \vdash M : A \leadsto M'}$

$$\frac{X \notin \mathrm{ftv}(\Delta(k))}{\Gamma \vdash k : \mu X.\ \Delta(k) \leadsto k} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \leadsto M' \quad \Gamma \vdash N : B \leadsto N' \\ X \notin \mathrm{ftv}(A \times B)\end{array}}{\Gamma \vdash (M, N) : \mu X.\ A{\times}B \leadsto (M', N')}$$

$$\frac{\Gamma, x{:}A \vdash N : B \leadsto N' \quad X \notin \mathrm{ftv}(A \to B)}{\Gamma \vdash (\lambda x{:}A.\ N) : \mu X.\ A{\to}B \leadsto (\lambda x{:}A.\ N')}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A \leadsto x} \qquad \frac{\begin{array}{c}\Delta(op) = (\bar{A}, B) \qquad \Gamma \vdash \bar{M} : \bar{C} \leadsto \bar{M}' \\ \forall i \in 1..n.\ M''_i = (M'_i : (C_i \rightharpoonup A_i) \xRightarrow{p} A_i)\end{array}}{\Gamma \vdash op(\bar{M}) : B \leadsto op(\bar{M}'')}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : A \leadsto M' \quad A \trianglerighteq (B{\times}C) \\ A' = (A \rightharpoonup B{\times}C) \\ M'' = (M' : A' \xRightarrow{p} B{\times}C)\end{array}}{\Gamma \vdash \mathtt{fst}\, M : B \leadsto \mathtt{fst}\, M''} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \leadsto M' \quad A \trianglerighteq (B{\times}C) \\ A' = (A \rightharpoonup B{\times}C) \\ M'' = (M' : A' \xRightarrow{p} B{\times}C)\end{array}}{\Gamma \vdash \mathtt{snd}\, M : C \leadsto \mathtt{snd}\, M''}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : A \leadsto L' \qquad A \trianglerighteq (B \to C) \qquad \Gamma \vdash M : B' \leadsto M' \\ L'' = (L : (A \rightharpoonup B{\to}C) \xRightarrow{p} B{\to}C) \quad M'' = (M' : (B' \rightharpoonup B) \xRightarrow{p} B)\end{array}}{\Gamma \vdash L\ M : C \leadsto L''\ M''}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : A \leadsto L' \quad A \trianglerighteq \cup\mathcal{C} \quad L'' = L' : (A \rightharpoonup \cup\mathcal{C}) \xRightarrow{p} \cup\mathcal{C} \\ B_1 = \mathcal{C}(c){\to}C \quad B'_1 = (\cup(\mathcal{C} - c)) \to C' \quad \Gamma \vdash M : B_1 \leadsto M' \quad \Gamma \vdash N : B'_1 \leadsto N' \\ C_2 = C \triangledown C' \quad B_2 = \mathcal{C}(c){\to}C_2 \quad B'_2 = (\cup(\mathcal{C} - c)) \to C_2 \\ M'' = (M' : (B_1 \rightharpoonup B_2) \xRightarrow{p} B_2) \quad N'' = (N' : (B'_1 \rightharpoonup B'_2) \xRightarrow{p'} B'_2)\end{array}}{\Gamma \vdash (\mathtt{case}\, c?(L)\, \mathtt{then}\, M\, \mathtt{else}\, N) : C_2 \leadsto (\mathtt{case}\, c?(L'')\, \mathtt{then}\, M''\, \mathtt{else}\, N'')}$$

$$\frac{\Gamma \vdash M : A \leadsto M' \quad A \lesssim B}{\Gamma \vdash (M :: B) : B \leadsto M' : (A \rightharpoonup B) \xRightarrow{p} B}$$

Fig. 7. Translation from $\lambda^{\star}_{\mu\cup}$ to a blame calculus via cast insertion

Merge on pre-types $\boxed{\Sigma \vdash P \rightharpoonup Q = P'}$

$$\Sigma \vdash \star \rightharpoonup Q = \star \quad \Sigma \vdash P \rightharpoonup \star = P \qquad \frac{\forall i \in 1..n.\ \Sigma \vdash A_i \rightharpoonup B_i = C_i}{\Sigma \vdash c(\bar{A}) \rightharpoonup c(\bar{B}) = c(\bar{C})}$$

$$\frac{\begin{array}{c}\mathcal{C} = \{c \mapsto c(\bar{A})\} \quad \Sigma \vdash c(\bar{A}) \rightharpoonup P = P' \\ P \neq \star\end{array}}{\Sigma \vdash \cup\mathcal{C} \rightharpoonup P = P'} \qquad \frac{\begin{array}{c}\Sigma \vdash P \rightharpoonup \mathcal{C}(c) = Q \quad c \in \mathrm{dom}(\mathcal{C}) \\ P \neq \star\end{array}}{\Sigma \vdash P \rightharpoonup \cup\mathcal{C} = \cup([c := Q]\mathcal{C})}$$

Merge $\boxed{\Sigma \vdash A \rightharpoonup B = A'}$

$$\frac{Z \mapsto A \rightharpoonup B \in \Sigma}{\Sigma \vdash A \rightharpoonup B = Z} \qquad \frac{\begin{array}{c}A = \mu X.\ P \qquad\qquad B = \mu Y.\ Q \\ \Sigma, Z \mapsto A \rightharpoonup B \vdash [X{:=}A]P \rightharpoonup [Y{:=}B]Q = Q'\end{array}}{\Sigma \vdash \mu X.\ P \rightharpoonup \mu Y.\ Q = \mu Z.\ Q'}$$

Fig. 8. Merge operator

4 Blame Calculus with Recursive Rypes and Union, $\lambda\mathsf{B}_{\mu\cup}$

In this section we study a blame calculus, named $\lambda\mathsf{B}_{\mu\cup}$, that is the appropriate intermediate language for $\lambda^{\star}_{\mu\cup}$. As discussed in Sect. 3.3, this blame calculus extends $\lambda_{\mu\cup}$ with an explicit cast operation from one type to another *compatible* type. This blame calculus is related to the original one of Wadler and Findler (2009) in that it adds recursive types and unions but omits subset types. The syntax for $\lambda\mathsf{B}_{\mu\cup}$ is defined at the top of Fig. 9.

With the addition of the type $\star$, the first question is what values of type $\star$ should look like. Wandler and Findler (2009) choose them to be values injected from *ground types* to $\star$:

$$V : G \overset{p}{\Longrightarrow} \star$$

We make the same choice here but extend the notion of ground type to handle recursive types and unions (Fig. 9). The inclusion of μ binders and type variables in the ground types is somewhat gratuitous because a well-formed ground type does not contain any type variables. However, if we were to enforce this in the grammar, it would require considerable special-casing elsewhere in the formalism. The type system of $\lambda\mathsf{B}_{\mu\cup}$ defined in Fig. 9 is a straightforward extension of $\lambda_{\mu\cup}$ (Fig. 3) to include the typing rule for explicit casts and blame.

Regarding the reduction rules, many of them concern casts between pre-types and not between (recursive) types, so we expand the syntax of terms to include casts between pre-types, i.e., $M : P \overset{p}{\Longrightarrow} Q$. Also, as usual, we abbreviate a pair of casts in the following way to avoid the duplication of the middle type.

$$(M : A \overset{p}{\Longrightarrow} B) : B \overset{q}{\Longrightarrow} C \quad \equiv \quad M : A \overset{p}{\Longrightarrow} B \overset{q}{\Longrightarrow} C$$

Now, the reduction rules for casts must handle every combination of source A and target type B where $A \sim B$. Thus, we need to handle casting from a union type to a compatible union type and similarly for recursive types. When casting a value between union types, $V : \cup\mathcal{C} \overset{p}{\Longrightarrow} \cup\mathcal{C}'$, we can ask for V's category via *tycon* and use it to select out the appropriate alternatives from the two union types (Fig. 9). This reduction rule preserves types because $\mathcal{C}'(c) <: \cup\mathcal{C}'$.

Regarding casts between recursive types, we unfold the recursive types, yielding a cast between two pre-types (Fig. 9). This reduction rule preserves types because of the typing rule for implicit folding into a recursive type (Fig. 3).

The final reduction rules that deserve mention are the two rules that handle an injection followed by a projection, that is, $V : G \overset{p'}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H$. In recent accounts of the blame calculus (Siek et al., 2015), the projection succeeds if $G = H$. In the current setting, we also need to consider the case where $G \neq H$ but $G <: H$. For example, take $G = \mu X.\, \mathtt{Int}$ and $H = \mu X.\, \cup \{\mathtt{Int} \mapsto \mathtt{Int}, \times \mapsto \mu Y.\, \mathtt{Bool} \times \mu Y.\, \mathtt{Bool}\}$. We want to allow such casts so we alter the reduction rule to succeed when $G <: H$ and fail otherwise. This reduction rule preserves types because $\emptyset \vdash V : H$ by subsumption.

Pre-types	P, Q	$::= c(\bar{A}) \mid \cup\mathcal{C} \mid \star$
Blame	p, q	
Terms	L, M, N	$::= k \mid op(\bar{M}) \mid \lambda x{:}A.\ N \mid L\ N \mid (M, N) \mid \mathtt{fst}\ L \mid \mathtt{snd}\ L \mid$ $\mathtt{case}\ c?(L)\ \mathtt{then}\ M\ \mathtt{else}\ N \mid M : A \overset{p}{\Longrightarrow} B \mid \mathtt{blame}\ p$
Ground pre-types	γ	$::= c(\bar{\star}) \mid \cup\{c \mapsto c(\bar{\star}), \ldots\}$
Ground types	G, H	$::= \mu X.\ \gamma \mid X$
Values	V, W	$::= k \mid \lambda x{:}A.\ N \mid (V, W) \mid V : G \overset{p}{\Longrightarrow} \star$
Frames	F	$::= \square\ N \mid V\ \square \mid (\square, N) \mid (V, \square) \mid \mathtt{fst}\ \square \mid \mathtt{snd}\ \square \mid$ $\mathtt{case}\ c?(\square)\ \mathtt{then}\ M\ \mathtt{else}\ N$

Term typing

$$(\text{same as } \lambda_{\mu\cup}) \qquad \cdots \qquad \frac{\Gamma \vdash M : A \qquad A \sim B}{\Gamma \vdash M : A \overset{p}{\Longrightarrow} B} \qquad \frac{}{\Gamma \vdash \mathtt{blame}\ p : A}$$

Reduction

$$(\text{same as } \lambda_{\mu\cup})$$

$$\vdots$$

$$V : \iota \overset{p}{\Longrightarrow} \iota \longrightarrow V \tag{1}$$

$$V : A{\to}B \overset{p}{\Longrightarrow} C{\to}D \longrightarrow \lambda x{:}C.\ V\ (x : C \overset{\bar{p}}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} D \tag{2}$$

$$(V, W) : A{\times}B \overset{p}{\Longrightarrow} C{\times}D \longrightarrow (V : A \overset{p}{\Longrightarrow} C, W : B \overset{p}{\Longrightarrow} D) \tag{3}$$

$$V : \cup\mathcal{C} \overset{p}{\Longrightarrow} \cup\mathcal{C}' \longrightarrow V : \mathcal{C}(c) \overset{p}{\Longrightarrow} \mathcal{C}'(c) \quad \text{where } c = \mathit{tycon}(V) \tag{4}$$

$$V : \mu X.\ P \overset{p}{\Longrightarrow} \mu X.\ Q \longrightarrow V : [X{:=}\mu X.\ P]P \overset{p}{\Longrightarrow} [X{:=}\mu X.\ Q]Q \tag{5}$$

$$V : \star \overset{p}{\Longrightarrow} \star \longrightarrow V \tag{6}$$

$$V : A \overset{p}{\Longrightarrow} \star \longrightarrow V : A \overset{p}{\Longrightarrow} G \overset{p}{\Longrightarrow} \star \quad \text{if } A \neq \star, A \neq G, A \sim G \tag{7}$$

$$V : \star \overset{p}{\Longrightarrow} A \longrightarrow V : \star \overset{p}{\Longrightarrow} G \overset{p}{\Longrightarrow} A \quad \text{if } A \neq \star, A \neq G, A \sim G \tag{8}$$

$$V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow V \quad \text{if } G <: H \tag{9}$$

$$V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow \mathtt{blame}\ p \quad \text{if } G \not<: H \tag{10}$$

Fig. 9. Blame calculus with recursive types and unions, $\lambda\mathsf{B}_{\mu\cup}$.

4.1 Example Revisited

Now that we have defined the translation from $\lambda^{\star}_{\mu\cup}$ and $\lambda\mathsf{B}_{\mu\cup}$ (Sect. 3.3) and the operational semantics for $\lambda\mathsf{B}_{\mu\cup}$, we can give an account for the example in the Sect. 1. We start by showing the result of cast insertion. To keep things concise, we introduce some shorthand notation. We use the notation $\lceil e \rceil$ to stand for the result of performing cast insertion on a dynamically typed expression e. Also, to make types more concise, we write $\hat{P}$ to turn the pre-type P into a type by adding an unused recursive variable, as in $\mu X.\ P$. Also, for union types we introduce the binary notation

$$c_1(\bar{A}) \cup c_2(\bar{B}) \equiv \cup\{c_1 \mapsto c_1(\bar{A}), c_2 \mapsto c_2(\bar{B})\}$$

Recall that

$$\mathtt{IntList} \equiv \mu X.\, \mathtt{Unit} \cup (\widehat{\mathtt{Int}} \times X)$$

Finally, we define *cons* and *nil* as follows

$$\begin{aligned} \mathit{cons} &\equiv \lambda x{:}\hat{\star}.\ \lambda y{:}\hat{\star}.\ (x, y) : (\hat{\star} \mathbin{\hat{\times}} \hat{\star}) \xRightarrow{p_4} \hat{\star} \\ \mathit{nil} &\equiv (\mathtt{unit} : \widehat{\mathtt{Unit}} \xRightarrow{p_5} \hat{\star}) \end{aligned}$$

The statically typed *sum* function remains unchanged by cast insertion and has type $\mathtt{IntList} \to \mathtt{Int}$, but the dynamically typed code that creates the list is translated as follows

$$(\mathit{cons}\ \lceil 1 \rceil\ (\mathit{cons}\ \lceil 2 \rceil\ \mathit{nil})) : \hat{\star} \xRightarrow{p_3} \mathtt{IntList}$$

Starting with the β reductions for *cons*, we have the following reduction sequence.

$$\begin{array}{lll}
\longrightarrow^{*} & ((\lceil 1 \rceil, \lceil (2, \mathit{nil}) \rceil) : (\hat{\star} \mathbin{\hat{\times}} \hat{\star}) \xRightarrow{p_4} \hat{\star} \xRightarrow{p_3} \mathtt{IntList}) & \\
\longrightarrow & ((\lceil 1 \rceil, \lceil (2, \mathit{nil}) \rceil) : (\hat{\star} \mathbin{\hat{\times}} \hat{\star}) \xRightarrow{p_4} \hat{\star} \xRightarrow{p_3} (\mu X.\, \mathtt{Unit} \cup (\hat{\star} \times \hat{\star})) \xRightarrow{p_3} \mathtt{IntList}) & \text{rule (8)} \\
\longrightarrow & ((\lceil 1 \rceil, \lceil (2, \mathit{nil}) \rceil) : (\mu X.\, \mathtt{Unit} \cup (\hat{\star} \times \hat{\star})) \xRightarrow{p_3} \mathtt{IntList}) & \text{rule (9)} \\
\longrightarrow & ((\lceil 1 \rceil, \lceil (2, \mathit{nil}) \rceil) : (\mathtt{Unit} \cup (\hat{\star} \times \hat{\star})) \xRightarrow{p_3} (\mathtt{Unit} \cup (\mathtt{Int} \times \mathtt{IntList}))) & \text{rule (5)} \\
\longrightarrow & ((\lceil 1 \rceil, \lceil (2, \mathit{nil}) \rceil) : (\hat{\star} \times \hat{\star}) \xRightarrow{p_3} (\mathtt{Int} \times \mathtt{IntList})) & \text{rule (4)} \\
\longrightarrow & (\lceil 1 \rceil : \hat{\star} \xRightarrow{p_3} \mathtt{Int}, \lceil (2, \mathit{nil}) \rceil : \hat{\star} \xRightarrow{p_3} \mathtt{IntList}) & \text{rule (3)} \\
\longrightarrow & (1, \lceil (2, \mathit{nil}) \rceil : \hat{\star} \xRightarrow{p_3} \mathtt{IntList}) & \text{rule (1)} \\
\longrightarrow^{*} & (1, (2, \mathtt{unit})) &
\end{array}$$

As expected, the list normalizes to a value of type `IntList`. We skipped the final two-thirds of the reductions because they are repetitive, applying the same rules as in the above reductions.

4.2 Type Safety of $\lambda\mathsf{B}_{\mu\cup}$

We state the main lemmas for type safety here and include the proofs in Appendix A.4.

Lemma 14 (Canonical Forms).

- *Suppose* $\emptyset \vdash V : P$.
  1. *If* $P = \star$ *then* $V = (V' : c(\bar{A}) \xRightarrow{p} \star)$ *for some* V', c, $\bar{A}$, *and* p.
  2. *If* $P = \iota$ *then* $V = k$ *and* $\Delta(k) = \iota$ *for some constant* k.
  3. *If* $P = A \times B$ *then* $V = (V_1, V_2)$ *for some* V_1 *and* V_2.
  4. *If* $P = A \to B$ *then* $V = \lambda x{:}A'.\ M$ *for some* x, A', *and* M.
  5. *If* $P = \cup\mathcal{C}$ *then* $\emptyset \vdash V : \mathcal{C}(c)$ *for some* $c \in \mathrm{dom}(\mathcal{C})$.
- *Suppose* $\emptyset \vdash V : \mu X.\ P$. *Then* $\emptyset \vdash V : [X{:=}\mu X.\ P]P$ *and the first part of this proposition can be applied.*

Lemma 15 (Preservation). *If* $\emptyset \vdash M : A$ *and* $M \longrightarrow N$, *then* $\emptyset \vdash N : A$.

Lemma 16 (Progress). *If* $\emptyset \vdash M : A$ *then either* M *is a value or* $M \longrightarrow N$ *for some* N.

4.3 Blame Safety

Here we adapt the standard blame safety theorem (Wadler and Findler, 2009, Siek et al., 2015) to $\lambda\mathsf{B}_{\mu\cup}$. A cast from A to B decorated with p is *safe* for blame label q, if evaluation of the cast never allocates blame to q. The following three rules reflect that if $A <:^+ B$ the cast never allocates positive blame, if $A <:^- B$ the cast never allocates negative blame, and a cast with label p never allocates blame other than to p or $\overline{p}$. Positive and negative subtyping are defined in Fig. 11 in Appendix A.5.

$$\frac{A <:^+ B}{(A \overset{p}{\Longrightarrow} B)\ \mathsf{safe}\ p} \qquad \frac{A <:^- B}{(A \overset{p}{\Longrightarrow} B)\ \mathsf{safe}\ \overline{p}} \qquad \frac{p \neq q \qquad \overline{p} \neq q}{(A \overset{p}{\Longrightarrow} B)\ \mathsf{safe}\ q}$$

Safety extends to terms in the obvious way: M safe q if every cast in M is safe for q. Blame safety is established via a variant of preservation and progress. The proof is in Appendix A.5.

Theorem 11 (Blame Safety).

1. *If* $M\ \mathsf{safe}\ p$ *and* $M \longrightarrow N$ *then* $N\ \mathsf{safe}\ p$*.*
2. *If* $M\ \mathsf{safe}\ p$ *then* $M \not\longrightarrow \mathtt{blame}\ p$*.*

5 Conclusion

Disjoint unions, in the form of sum types as seen in languages such as ML and Haskell, have been widely successful in programming languages. However, non-disjoint unions, while mathematically natural, have seen much less adoption, even though they naturally model the reasoning programmers employ in dynamic languages. We contend that this failure is due to a lack of a natural elimination rule for union types. In this paper, we show that *dynamic type tests are the natural elimination rule for union types*. Furthermore, support for true unions fits naturally with equi-recursive types.

The need for type systems with true union types is particularly pressing in gradually-typed languages, and we show that existing approaches to gradual type systems scale naturally to handle both unions and equi-recursive types. Our model is both simple enough to serve as a core calculus, while also including the key features we need to model languages such as Typed Racket and TypeScript.

Our development scales from a simple, typed language through a gradual type system, to a calculus of casts which demonstrates the operational behavior of casts, including casts to union and recursive types. We show both declarative and algorithmic versions of our type system, as well as how to assign blame when casts fail, proving the blame theorem showing that our dynamic checks adequately enforce the corresponding static types.

A Appendix

A.1 Type Safety of $\lambda_{\mu\cup}$

Lemma 1 (Subtyping Substitution). *If $\mu X.\ P <: \mu X.\ Q$, then $[X{:=}\mu X.\ P]P <: [X{:=}\mu X.\ Q]Q$.*

Lemma 2 (Canonical Forms).

- *Suppose $\emptyset \vdash V : P$.*
  1. *If $P = \iota$ then $V = k$ and $\Delta(k) = \iota$ for some constant k.*
  2. *If $P = A \times B$ then $V = (V_1, V_2)$ for some V_1 and V_2.*
  3. *If $P = A \to B$ then $V = \lambda x{:}A'.\ M$ for some x, A', and M.*
  4. *If $P = \cup\mathcal{C}$ then $\emptyset \vdash V : \mathcal{C}(c)$ for some $c \in \mathrm{dom}(\mathcal{C})$.*
- *Suppose $\emptyset \vdash V : \mu X.\ P$. Then $\emptyset \vdash V : [X{:=}\mu X.\ P]P$ and the first part of this proposition can be applied.*

Proof. The proof is by mutual induction on the derivation of $\emptyset \vdash V : P$ and $\emptyset \vdash V : \mu X.\ P$. The cases for constants, pairs, and abstraction are immediate.

Case $\boxed{\dfrac{\emptyset \vdash V : \mu X.\ P}{\emptyset \vdash V : [X{:=}\mu X.\ P]P}}$

From $\emptyset \vdash V : \mu X.\ P$ we have $\emptyset \vdash V : [X{:=}\mu X.\ P]P$ by induction. Then again by induction we conclude.

Case $\boxed{\dfrac{\emptyset \vdash V : A \qquad \emptyset \vdash A <: B}{\emptyset \vdash V : B}}$

Let $A = \mu X.\ P$ and $B = \mu X.\ Q$. By induction we have $\emptyset \vdash V : [X{:=}A]P$. From $A <: B$ we have $[X{:=}A]P <: [X{:=}B]Q$ by Lemma 1. Thus we conclude that $\emptyset \vdash V : [X{:=}\mu X.\ Q]Q$

Case $\boxed{\dfrac{\Gamma \vdash V : [X{:=}\mu X.\ P]P}{\Gamma \vdash V : \mu X.\ P}}$

The rest of the typing rules don't apply to values. □

Lemma 3 (Preservation). *If $\emptyset \vdash M : A$ and $M \longrightarrow N$, then $\emptyset \vdash N : A$.*

Proof. The proof is by induction on the derivation of $M \longrightarrow N$. All of the reduction rules are standard except for the reduction rule for `case`.

Case $\boxed{\texttt{case}\, c?(V)\, \texttt{then}\, M'\, \texttt{else}\, N' \longrightarrow \begin{cases} M'\ V & \text{if } \mathit{tycon}(V) = c \\ N'\ V & \text{otherwise} \end{cases}}$ We have $\emptyset \vdash V : \cup\mathcal{C}$, $\emptyset \vdash M' : \mathcal{C}(c) \to A$, and $\emptyset \vdash N' : (\cup\mathcal{C} - c) \to A$. We have two subcases to consider.

Subcase $\mathit{tycon}(V) = c$. From $\emptyset \vdash V : \cup\mathcal{C}$, canonical forms (Lemma 2), and $\mathit{tycon}(V) = c$, we have $\emptyset \vdash V : \mathcal{C}(c)$. We conclude that $\emptyset \vdash M'\ V : A$.

Subcase $\mathit{tycon}(V) \neq c$. From $\emptyset \vdash V : \cup\mathcal{C}$, canonical forms (Lemma 2), and $\mathit{tycon}(V) \neq c$, we have $\emptyset \vdash V : \cup(\mathcal{C} - c)$. We conclude that $\emptyset \vdash N'\ V : A$.

Lemma 4 (Progress). *If $\emptyset \vdash M : A$ then either M is a value or $M \longrightarrow N$ for some N.*

Proof. The proof is by induction on the derivation of $\emptyset \vdash M : A$. The only novel case in the proof is for the typing rule of `case`.

$$\text{Case}\ \boxed{\dfrac{\emptyset \vdash L : \cup\mathcal{C} \qquad \emptyset \vdash M' : \mathcal{C}(c) \to A \\ \emptyset \vdash N' : (\cup(\mathcal{C} - c)) \to A}{\emptyset \vdash (\texttt{case}\, c?(L)\, \texttt{then}\, M'\, \texttt{else}\, N') : A}}$$

By induction, either L is a value or $L \longrightarrow L'$. In the later case we conclude immediately using the frame reduction rule. Suppose L is a value. From canonical forms (Lemma 2) we have $\emptyset \vdash L : \mathcal{C}(c')$ for some $c' \in \mathrm{dom}(\mathcal{C})$. We have two cases to consider.

Subcase $c = c'$. We have $\emptyset \vdash L : c(\bar{B})$ so by cases on the canonical forms, we have $\mathit{tycon}(L) = c$. Therefore the following reduction applies:

$$(\texttt{case}\, c?(L)\, \texttt{then}\, M'\, \texttt{else}\, N') \longrightarrow M'\ L$$

Subcase $c \neq c'$. We have $\emptyset \vdash L : c'(\bar{B})$ so by cases on the canonical forms, we have $\mathit{tycon}(L) \neq c$. Therefore the following reduction applies:

$$(\texttt{case}\, c?(L)\, \texttt{then}\, M'\, \texttt{else}\, N') \longrightarrow N'\ L$$

A.2 Correctness of Algorithmic Type System for $\lambda^{\star}_{\mu\cup}$

Figure 10 defines the meet-join and meet-meet operators.

A.3 Translation to $\lambda\mathsf{B}_{\mu\cup}$ Preserves Types

Lemma 5 (Merge and Subtyping). *If $A \lesssim B$ then $A <: (A \rightharpoonup B)$.*

Lemma 6 (Merge and Compatibility). $(A \rightharpoonup B) \sim B$

Lemma 7 (Meet-Join and Compatible-Subtype). $A \lesssim (A \triangledown B)$ *and* $B \lesssim (A \triangledown B)$.

Lemma 13 (Cast Insertion Preserves Types). *If $\Gamma \vdash M : A \rightsquigarrow M'$ then $\Gamma \vdash M' : A$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash M : A \rightsquigarrow M'$.

$$\text{Case}\ \boxed{\dfrac{\Gamma \vdash M : A \rightsquigarrow M' \qquad A \trianglerighteq (B\times C) \qquad A' = (A \rightharpoonup B\times C) \\ M'' = (M' : A' \overset{p}{\Longrightarrow} B\times C)}{\Gamma \vdash \texttt{fst}\, M : B \rightsquigarrow \texttt{fst}\, M''}}$$

Meet-join on pre-types $\boxed{\Sigma \vdash P \triangledown Q = P'}$

$$\frac{}{\Sigma \vdash \iota \triangledown \iota = \iota} \qquad \frac{\Sigma \vdash A_1 \triangledown B_1 = C_1 \quad \Sigma \vdash A_2 \triangledown B_2 = C_2}{\Sigma \vdash (A_1{\times}A_2) \triangledown (B_1{\times}B_2) = C_1{\times}C_2}$$

$$\frac{\Sigma \vdash A_1 \vartriangle B_1 = C_1 \quad \Sigma \vdash A_2 \triangledown B_2 = C_2}{\Sigma \vdash (A_1{\to}A_2) \triangledown (B_1{\to}B_2) = C_1{\to}C_2}$$

$$\frac{\begin{array}{c}\mathcal{C}_1 = \{P \mid c \in \mathrm{dom}(\mathcal{C}) \cap \mathrm{dom}(\mathcal{C}'), \Sigma \vdash \mathcal{C}(c) \triangledown \mathcal{C}'(c) = P\} \\ \mathcal{C}_2 = \{\mathcal{C}(c) \mid c \in \mathrm{dom}(\mathcal{C}) - \mathrm{dom}(\mathcal{C}')\} \quad \mathcal{C}_3 = \{\mathcal{C}'(c) \mid c \in \mathrm{dom}(\mathcal{C}') - \mathrm{dom}(\mathcal{C})\}\end{array}}{\Sigma \vdash \cup\mathcal{C} \triangledown \cup\mathcal{C}' = \cup(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3)}$$

$$\frac{\Sigma \vdash \cup\mathcal{C} \triangledown \cup\{c \mapsto c(\bar{B})\} = C}{\Sigma \vdash \cup\mathcal{C} \triangledown c(\bar{B}) = C} \qquad \frac{\Sigma \vdash \cup\{c \mapsto c(\bar{A})\} \triangledown \cup\mathcal{C} = C}{\Sigma \vdash c(\bar{A}) \triangledown \cup\mathcal{C} = C}$$

$$\frac{}{\Sigma \vdash \star \triangledown Q = Q} \qquad \frac{}{\Sigma \vdash P \triangledown \star = P}$$

Meet-join $\boxed{\Sigma \vdash A \triangledown B = C}$

$$\frac{Z \mapsto A \triangledown B \in \Sigma}{\Sigma \vdash A \triangledown B = Z} \qquad \frac{\begin{array}{cc} A = \mu X.\, P & B = \mu Y.\, Q \\ \Sigma, Z \mapsto A \triangledown B & \vdash [X{:=}A]P \triangledown [Y{:=}B]Q = Q'\end{array}}{\Sigma \vdash \mu X.\, P \triangledown \mu X.\, Q = \mu Z.\, Q'}$$

Meet-meet on pre-types $\boxed{\Sigma \vdash P \vartriangle Q = P'}$

$$\frac{}{\iota \vartriangle \iota = \iota} \qquad \frac{\Sigma \vdash A_1 \vartriangle B_1 = C_1 \quad \Sigma \vdash A_2 \vartriangle B_2 = C_2}{\Sigma \vdash (A_1{\times}A_2) \vartriangle (B_1{\times}B_2) = C_1{\times}C_2}$$

$$\frac{\Sigma \vdash A_1 \triangledown B_1 = C_1 \quad \Sigma \vdash A_2 \vartriangle B_2 = C_2}{\Sigma \vdash (A_1{\to}A_2) \vartriangle (B_1{\to}B_2) = C_1{\to}C_2}$$

$$\frac{\mathcal{C}_3 = \{\mathcal{C}_1(c) \vartriangle \mathcal{C}_2(c) \mid c \in \mathrm{dom}(\mathcal{C}_1) \cap \mathrm{dom}(\mathcal{C}_2)\}}{\cup\mathcal{C}_1 \vartriangle \cup\mathcal{C}_2 = \cup\mathcal{C}_3}$$

$$\frac{\Sigma \vdash \mathcal{C}(c) \vartriangle c(\bar{B}) = C}{\Sigma \vdash \cup\mathcal{C} \vartriangle c(\bar{B}) = C} \qquad \frac{\Sigma \vdash c(\bar{A}) \vartriangle \mathcal{C}(c) = C}{\Sigma \vdash c(\bar{A}) \vartriangle \cup\mathcal{C} = C}$$

$$\frac{}{\Sigma \vdash \star \vartriangle B = B} \qquad \frac{}{\Sigma \vdash A \vartriangle \star = A}$$

Meet-meet $\boxed{\Sigma \vdash A \vartriangle B = C}$

$$\frac{Z \mapsto A \vartriangle B \in \Sigma}{\Sigma \vdash A \vartriangle B = Z} \qquad \frac{\begin{array}{cc} A = \mu X.\, P & B = \mu Y.\, Q \\ \multicolumn{2}{c}{\Sigma, Z \mapsto A \vartriangle B \vdash [X{:=}A]P \vartriangle [Y{:=}B]Q = C}\end{array}}{\Sigma \vdash \mu X.\, P \vartriangle \mu X.\, Q = \mu Z.\, C}$$

Fig. 10. Combining meet and join with respect to $<:_n$ and $<:$.

By the induction hypothesis, we have $\Gamma \vdash M' : A$. From $A \trianglerighteq (B\times C)$ we have $A \lesssim (B\times C)$. Then by Lemma 5, we have $A <: A'$, so $\Gamma \vdash M' : A'$. By Lemma 6 we have $A' \sim B\times C$ and therefore $\Gamma \vdash M'' : B\times C$. We conclude that $\Gamma \vdash \mathtt{fst}\, M'' : B$.

Case $\boxed{\dfrac{\Gamma \vdash M : A \leadsto M'}{\Gamma \vdash (M :: B) : B \leadsto M' : (A \rightharpoonup B) \xRightarrow{p} B}}$

By the induction hypothesis, we have $\Gamma \vdash M' : A$. From $A \lesssim B$ we have $A <: (A \rightharpoonup B)$. So $\Gamma \vdash M' : (A \rightharpoonup B)$. Also, we have $(A \rightharpoonup B) \sim B$.

$$\Gamma \vdash (M' : (A \rightharpoonup B) \xRightarrow{p} B) : B$$

Case $\boxed{\dfrac{\begin{array}{c} \Gamma \vdash L : A \leadsto L' \quad A \trianglerighteq \cup\mathcal{C} \quad L'' = L' : (A \rightharpoonup \cup\mathcal{C}) \xRightarrow{p} \cup\mathcal{C} \\ B_1 = \mathcal{C}(c) \to C \quad B_1' = (\cup(\mathcal{C} - c)) \to C' \\ \Gamma \vdash M : B_1 \leadsto M' \qquad\qquad \Gamma \vdash N : B_1' \leadsto N' \\ C_2 = C \triangledown C' \quad B_2 = \mathcal{C}(c) \to C_2 \quad B_2' = (\cup(\mathcal{C} - c)) \to C_2 \\ M'' = (M' : (B_1 \rightharpoonup B_2) \xRightarrow{p} B_2) \quad N'' = (N' : (B_1' \rightharpoonup B_2') \xRightarrow{p'} B_2') \end{array}}{\Gamma \vdash (\mathtt{case}\, c?(L)\, \mathtt{then}\, M\, \mathtt{else}\, N) : C_2 \leadsto (\mathtt{case}\, c?(L'')\, \mathtt{then}\, M''\, \mathtt{else}\, N'')}}$

From $C_2 = C \vartriangle C'$ we have $C \lesssim C_2$ (Lemma 7) and so $C <: (C \rightharpoonup C_2)$. Similarly, we have $C' <: (C' \rightharpoonup C_2)$. Also, we have $(C \rightharpoonup C_2) \sim C_2$ and $(C' \rightharpoonup C_2) \sim C_2$ (Lemma 6). Thus, with the induction hypotheses for M and N, we have $\Gamma \vdash M'' : B_2$ and $\Gamma \vdash N'' : B_2'$. Similarly, we also have $\Gamma \vdash L'' : \cup\mathcal{C}$. Therefore, we have

$$\Gamma \vdash (\mathtt{case}\, c?(L'')\, \mathtt{then}\, M''\, \mathtt{else}\, N'') : C_2$$

The rest of the cases are similar. □

A.4 Type Safety of $\boldsymbol{\lambda}\mathbf{B}_{\mu\cup}$

Lemma 8. *If $\mu X.\, P \sim \mu Y.\, Q$ then $[X{:=}\mu X.\, P]P \sim [Y{:=}\mu Y.\, Q]Q$.*

Lemma 15 (Preservation). *If $\emptyset \vdash M : A$ and $M \longrightarrow N$, then $\emptyset \vdash N : A$.*

Proof. The proof is by induction on the derivation of $M \longrightarrow N$. We only consider the novel cases.

Case $\boxed{V : \cup\mathcal{C} \xRightarrow{p} \cup\mathcal{C}' \longrightarrow V : \mathcal{C}(c) \xRightarrow{p} \mathcal{C}'(c) \quad \text{where } c = \mathit{tycon}(V)}$

By canonical forms (Lemma 14) and because $\mathit{tycon}(V) = c$, we have $\emptyset \vdash V : \mathcal{C}(c)$. Also, because $\cup\mathcal{C} \sim \cup\mathcal{C}'$, we have $\mathcal{C}(c) \sim \mathcal{C}'(c)$. So $\emptyset \vdash (V : \mathcal{C}(c) \xRightarrow{p} \mathcal{C}'(c)) : \mathcal{C}'(c)$. Finally, because $\mathcal{C}'(c) <: \cup\mathcal{C}'$ we conclude that

$$\emptyset \vdash (V : \mathcal{C}(c) \xRightarrow{p} \mathcal{C}'(c)) : \cup\mathcal{C}'$$

Case $\boxed{V : \mu X.\, P \overset{p}{\Longrightarrow} \mu Y.\, Q \longrightarrow V : [X{:=}\mu X.\, P]P \overset{p}{\Longrightarrow} [Y{:=}\mu Y.\, Q]Q}$

By canonical forms (Lemma 14) we have $\emptyset \vdash V : [X{:=}\mu X.\, P]P$. Also, because $\mu X.\, P \sim \mu Y.\, Q$ we have $[X{:=}\mu X.\, P]P \sim [Y{:=}\mu Y.\, Q]Q$ (Lemma 8). Thus, we have

$$\emptyset \vdash (V : [X{:=}\mu X.\, P]P \overset{p}{\Longrightarrow} [Y{:=}\mu Y.\, Q]Q) : [Y{:=}\mu Y.\, Q]Q$$

Case $\boxed{V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow V}$

We have $\emptyset \vdash V : G$ and $G <: H$, so $\emptyset \vdash V : H$. □

Lemma 16 (Progress). *If $\emptyset \vdash M : A$ then either M is a value or $M \longrightarrow N$ for some N.*

Proof. The proof is by induction on the derivation of $\emptyset \vdash M : A$. All of the cases in the proof are standard except for the case for cast.

Case $\boxed{\dfrac{\Gamma \vdash M' : A' \qquad A' \sim A}{\Gamma \vdash M' : A' \overset{p}{\Longrightarrow} A}}$

By the induction hypothesis, either M' is a value or $M' \longrightarrow N'$. In the later case we can conclude immediately by applying the reduction rule for frames. Suppose M' is a value. We proceed by cases on $A' \sim A$.

Subcase $\boxed{\star \sim \star}$ We have

$$M' : \star \overset{p}{\Longrightarrow} \star \longrightarrow M'$$

Subcase $\boxed{\star \sim Q, Q \neq \star}$ By canonical forms (Lemma 14), $M' = V : G \overset{q}{\Longrightarrow} \star$. Now if Q is a ground type (let it be H), then we have either

$$V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow V \quad \text{or} \quad V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow \texttt{blame}\, p$$

depending on whether $G <: H$ or not.

Subcase $\boxed{P \sim \star, P \neq \star}$ If P is a ground type, then $M' : P \overset{p}{\Longrightarrow} \star$ is a value. Otherwise, we have

$$M' : P \overset{p}{\Longrightarrow} G \overset{p}{\Longrightarrow} \star$$

where G is the unique ground type compatible with P.

Subcase $\boxed{\iota \sim \iota}$ We have

$$M' : \iota \overset{p}{\Longrightarrow} \iota \longrightarrow M'$$

Subcase $\boxed{(A{\times}B) \sim (C{\times}D)}$ By canonical forms (Lemma 14), we have $M' = (V, W)$. So

$$(V, W) : A{\times}B \overset{p}{\Longrightarrow} C{\times}D \longrightarrow (V : A \overset{p}{\Longrightarrow} C, W : B \overset{p}{\Longrightarrow} D)$$

Subcase $\boxed{(A{\to}B) \sim (C{\to}D)}$ We have

$$M' : A{\to}B \overset{p}{\Longrightarrow} C{\to}D \longrightarrow \lambda x{:}C.\ M'\ (x : C \overset{\overline{p}}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} D$$

Subcase $\boxed{\cup\mathcal{C} \sim \cup\mathcal{C}'}$ Let $c = \mathit{tycon}(M')$. We have

$$M' : \cup\mathcal{C} \overset{p}{\Longrightarrow} \cup\mathcal{C}' \longrightarrow M' : \mathcal{C}(c) \overset{p}{\Longrightarrow} \mathcal{C}'(c)$$

Subcase $\boxed{\mu X.\ P \sim \mu X.\ Q}$ We have

$$M' : \mu X.\ P \overset{p}{\Longrightarrow} \mu X.\ Q \longrightarrow M' : [X{:=}\mu X.\ P]P \overset{p}{\Longrightarrow} [X{:=}\mu X.\ Q]Q$$

□

A.5 Blame Safety

Figure 11 defines the positive and negative subtype relations used to define blame safety in Sect. 4.3.

Lemma 9 (Unfolding Positive and Negative Subtyping).

1. *If* $\mu X.\ P <:^{+} \mu X.\ Q$ *then* $[X{:=}\mu X.\ P]P <:^{+} [X{:=}\mu X.\ Q]Q$
2. *If* $\mu X.\ P <:^{-} \mu X.\ Q$ *then* $[X{:=}\mu X.\ P]P <:^{-} [X{:=}\mu X.\ Q]Q$

Theorem 12 (Blame Safety).

1. *If* $M\ \mathsf{safe}\ p$ *and* $M \longrightarrow N$ *then* $N\ \mathsf{safe}\ p$.
2. *If* $M\ \mathsf{safe}\ p$ *then* $M \not\longrightarrow \mathtt{blame}\ p$.

Proof.

1. The proof of the first part is by induction on the derivation $M \longrightarrow N$. The interesting cases involve the reduction of casts.

   Case $\boxed{V : \cup\mathcal{C} \overset{q}{\Longrightarrow} \cup\mathcal{C}' \longrightarrow V : \mathcal{C}(c) \overset{q}{\Longrightarrow} \mathcal{C}'(c)}$ where $c = \mathit{tycon}(V)$

   We have V safe p and $(\cup\mathcal{C} \overset{p}{\Longrightarrow} \cup\mathcal{C}')$ safe p. There are three cases to consider:

   (a) Subcase $q = p$: So $\cup\mathcal{C} <:^{+} \cup\mathcal{C}'$ and therefore $\mathcal{C}(c) <:^{+} \mathcal{C}'(c)$. Thus

   $$(V : \mathcal{C}(c) \overset{p}{\Longrightarrow} \mathcal{C}'(c))\ \mathsf{safe}\ p$$

   (b) Subcase $q = \overline{p}$: So $\cup\mathcal{C} <:^{-} \cup\mathcal{C}'$ and therefore $\mathcal{C}(c) <:^{-} \mathcal{C}'(c)$. Thus

   $$(V : \mathcal{C}(c) \overset{\overline{p}}{\Longrightarrow} \mathcal{C}'(c))\ \mathsf{safe}\ p$$

   (c) Subcase $q \neq p, q \neq \overline{p}$: We immediately have

   $$(V : \mathcal{C}(c) \overset{q}{\Longrightarrow} \mathcal{C}'(c))\ \mathsf{safe}\ p$$

Positive subtype on pre-types $\boxed{\Sigma \vdash P <:^{+} Q}$

$$\frac{}{\Sigma \vdash \iota <:^{+} \iota} \qquad \frac{\Sigma \vdash A <:^{+} C \quad \Sigma \vdash B <:^{+} D}{\Sigma \vdash A{\times}B <:^{+} C{\times}D} \qquad \frac{\Sigma \vdash C <:^{-} A \quad \Sigma \vdash B <:^{+} D}{\Sigma \vdash A{\to}B <:^{+} C{\to}D}$$

$$\frac{\mathrm{dom}(C) = \mathrm{dom}(C') \quad \forall c \in \mathrm{dom}(C).\, \mathcal{C}(c) <:^{+} \mathcal{C}'(c)}{\cup\mathcal{C} <:^{+} \cup\mathcal{C}'} \qquad \frac{}{\Sigma \vdash P <:^{+} \star}$$

Positive subtyping $\boxed{\Sigma \vdash A <:^{+} B}$

$$\frac{A <:^{+} B \in \Sigma}{\Sigma \vdash A <:^{+} B} \qquad \frac{\begin{array}{c} A = \mu X.\, P \qquad B = \mu Y.\, Q \\ \Sigma, A <:^{+} B \vdash [X{:=}A]P <:^{+} [Y{:=}B]Q \end{array}}{\Sigma \vdash A <:^{+} B}$$

Negative subtype on pre-types $\boxed{\Sigma \vdash P <:^{-} Q}$

$$\frac{}{\Sigma \vdash \iota <:^{-} \iota} \qquad \frac{\Sigma \vdash A <:^{-} C \quad \Sigma \vdash B <:^{-} D}{\Sigma \vdash A{\times}B <:^{-} C{\times}D} \qquad \frac{\Sigma \vdash C <:^{+} A \quad \Sigma \vdash B <:^{-} D}{\Sigma \vdash A{\to}B <:^{-} C{\to}D}$$

$$\frac{\begin{array}{c} \mathrm{dom}(C) = \mathrm{dom}(C') \\ \forall c \in \mathrm{dom}(C).\, \mathcal{C}(c) <:^{-} \mathcal{C}'(c) \end{array}}{\cup\mathcal{C} <:^{-} \cup\mathcal{C}'} \qquad \frac{}{\Sigma \vdash \star <:^{-} Q} \qquad \frac{\Sigma \vdash P <:^{-} \gamma}{\Sigma \vdash P <:^{-} \star}$$

Negative subtyping $\boxed{\Sigma \vdash A <:^{-} B}$

$$\frac{A <:^{-} B \in \Sigma}{\Sigma \vdash A <:^{-} B} \qquad \frac{\begin{array}{c} A = \mu X.\, P \qquad B = \mu Y.\, Q \\ \Sigma, A <:^{-} B \vdash [X{:=}A]P <:^{-} [Y{:=}B]Q \end{array}}{\Sigma \vdash A <:^{-} B}$$

Fig. 11. Positive and negative subtyping

Case $\boxed{V : \mu X.\, P \overset{p}{\Longrightarrow} \mu X.\, Q \longrightarrow V : [X{:=}\mu X.\, P]P \overset{p}{\Longrightarrow} [X{:=}\mu X.\, Q]Q}$

We have V $\mathsf{safe}\ p$ and $(\mu X.\, P \overset{p}{\Longrightarrow} \mu X.\, Q)$ $\mathsf{safe}\ p$. There are three cases to consider:

(a) Subcase $q = p$: So $\mu X.\, P <:^{+} \mu X.\, Q$ and therefore $[X{:=}\mu X.\, P]P <:^{+} [X{:=}\mu X.\, Q]Q$ (Lemma 9). Thus

$$(V : [X{:=}\mu X.\, P]P \overset{p}{\Longrightarrow} [X{:=}\mu X.\, Q]Q)\ \mathsf{safe}\ p$$

(b) Subcase $q = \overline{p}$: So $\mu X.\, P <:^{-} \mu X.\, Q$ and therefore $[X{:=}\mu X.\, P]P <:^{-} [X{:=}\mu X.\, Q]Q$ (Lemma 9). Thus

$$(V : [X{:=}\mu X.\, P]P \overset{\overline{p}}{\Longrightarrow} [X{:=}\mu X.\, Q]Q)\ \mathsf{safe}\ p$$

(c) Subcase $q \neq p, q \neq \overline{p}$: We immediately have

$$(V : [X{:=}\mu X.\, P]P \overset{q}{\Longrightarrow} [X{:=}\mu X.\, Q]Q)\ \mathsf{safe}\ p$$

The rest of the cases are the same as for the original blame calculus.

2. The proof of the second part is by induction on $M \longrightarrow \texttt{blame}\, p$ and we seek to reach a contradiction. The only reduction rule that triggers blame is $\boxed{V : G \overset{q}{\Longrightarrow} \star \overset{p}{\Longrightarrow} H \longrightarrow \texttt{blame}\, p}$ where $G \not<: H$
   We have $(\star \overset{p}{\Longrightarrow} H)$ safe p so $\star <:^{+} H$. Thus $H = \star$ but $\star$ is not a ground type, hence a contradiction. □

References

Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. **15**(4), 575–631 (1993)

Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundam. Inf. **33**(4), 309–338 (1998)

Cartwright, R., Fagan, M.: Soft typing. In: Conference on Programming Language Design and Implementation, PLDI, pp. 278–292. ACM Press (1991)

Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In: Symposium on Principles of Programming Languages, POPL, pp. 5–17. ACM (2014)

Cimini, M., Siek, J.G.: The gradualizer: a methodology and algorithm for generating gradual type systems. In: Symposium on Principles of Programming Languages, POPL, January 2016

Flatt, M., and PLT.: The Racket reference 6.0. Technical report, PLT Inc. (2014). http://docs.racket-lang.org/reference/index.html

Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 19: 1–19: 64 (2008)

Kelsey, R., Clinger, W., Rees, J.: Revised^5 report on the algorithmic language scheme. High.-Order Symbolic Comput. **11**(1), 7–105 (1998)

Pierce, B.C.: Programming with intersection types, union types, and polymorphism. Technical report CMU-CS-91-106, Carnegie Mellon University (1991)

Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)

Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)

Siek, J.G., Thiemann, P., Wadler, P.: Blame and coercion: together again for the first time. In: Conference on Programming Language Design and Implementation, PLDI, June 2015

Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: Symposium on Principles of Programming Languages, January 2008

Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: International Conference on Functional Programming, ICFP, pp. 117–128. ACM (2010)

Vouillon, J.: Subtyping union types. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 415–429. Springer, Heidelberg (2004)

Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009)

A Delta for Hybrid Type Checking

Peter Thiemann(✉)

University of Freiburg, Freiburg, Germany
thiemann@acm.org

Abstract. A hybrid type checker defers parts of the type checking to run time. At compile time, the checker attempts to statically verify as many subtyping constraints as possible. Constraints that cannot be proved by the static checker, are reified as run-time casts in an intermediate language, which is a variant of the blame calculus.

The goal of this work is to simplify casts in the intermediate blame calculus by exploiting context information. To this end, we develop a coercion calculus that corresponds to the blame calculus via a pair of translations and we define the formal framework to simplify these coercions. We give a concrete instance of the calculus and demonstrate that simplification can be regarded as a synthesis problem.

1 Introduction

Flanagan and others [7] introduce hybrid type checking as a method to combine the best of two worlds, static and dynamic type checking. Their work provides a framework that employs static type checking as much as possible and reverts to dynamic checking when typing constraints cannot be resolved statically. The basis of their work is a highly expressive language with dependent types and refinements at base types.

Such a language supports refinement types like $x : \mathsf{int}\{x > 0\}$ for the set of positive integers and $x : \mathsf{int}\{x\%2 = 0\}$ for the set of even integers. On top of that, there are dependent function types like the type of strictly increasing functions $x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$.[1] These types come with the standard notion of subtyping for dependent types [1] (see also Fig. 3).

The extra ingredient of a language with hybrid type checking is a cast expression that we write as $M : S \Rightarrow T$ for casting the value of M from source type S to target type T. For instance, a cast may further restrict an increasing function F so that it never returns a value greater than twice its input.

$$\begin{aligned} G = (F :\quad & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}) \\ \Rightarrow\ & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x \wedge y < 2 * x\})) \end{aligned}$$

[1] This notation is borrowed from Swamy [11]. It stands for a dependent type with argument type $x : \mathsf{int}\{x > 0\}$ and result type $y : \mathsf{int}\{y > x\}$ where the scope of x extends to the result type. We sometimes omit the type annotation $: \mathsf{int}$ if the base type is clear from the context.

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 411–432, 2016.
DOI: 10.1007/978-3-319-30936-1_22

If we apply the resulting function G to a suitable argument, say 42, then after a few steps (and skipping over some details), we end up with a term that applies a cast that is derived from the ranges of the original function type cast to the result of the function application:

$$(F\,42) \ : \ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

The implementation of this cast checks the predicate $Q = y > 42 \wedge y < 2 * 42$ on the result $y = (F\,42)$ of the function call. But this check performs more work than strictly necessary because it ignores the static knowledge $P = y > 42$ from the precondition on y. Our goal is to develop a framework to find the *delta predicate*, which we check at run time, which is cheaper to test than Q, but which is equivalent to Q when assuming P. In this particular example, a suitable delta predicate is $y < 84$.

In a cast between dependent types, restricting the argument type can also lead to simplifying the run-time check on the result. As an example consider a function cast, where the source type is modeled after the intersection type $(Pos \to Pos) \cap (Even \to Even)$:

$$\begin{array}{l} x : \mathsf{int}\{x > 0 \vee x\%2 = 0\} \to y : \mathsf{int}\{(x > 0 \Rightarrow y > 0) \wedge (x\%2 = 0 \Rightarrow y\%2 = 0)\} \\ \Rightarrow \\ x : \mathsf{int}\{x > 0 \wedge x\%2 = 0\} \to y : \mathsf{int}\{y > 0 \wedge y\%2 = 0\} \end{array}$$

To evaluate this cast, we first have to cast the argument x according to

$$x : \mathsf{int}\{x > 0 \wedge x\%2 = 0\} \Rightarrow x : \mathsf{int}\{x > 0 \vee x\%2 = 0\}$$

This argument cast requires no run-time check because it casts to a supertype (since $x > 0 \wedge x\%2 = 0$ implies $x > 0 \vee x\%2 = 0$ for all $x : \mathsf{int}$). To simplify the result cast, we can exploit the knowledge from **both** domain predicates. It turns out that $x > 0 \wedge x\%2 = 0$ and $(x > 0 \Rightarrow y > 0) \wedge (x\%2 = 0 \Rightarrow y\%2 = 0)$ implies $y > 0 \wedge y\%2 = 0$, so that no run-time check is required for the result cast, either. This freedom of run-time checks should not come as a surprise, because the cast's target is a (dependent) subtype of the cast's source. In our framework, we want to eliminate all run-time checks from this cast.

In the rest of the paper, we formalize a dependently typed blame calculus based on the ideas of hybrid typing. This calculus can be seen as an intermediate language in compiling a dependently typed language with refinement types: Subtyping constraints that can be discharged at compile time are eliminated and the remaining ones are reified as run-time type casts. The calculus can also be considered as a source language where programmers place explicit casts. This situation is similar as in the blame calculus considered by Wadler and Findler [14]. However, their base calculus is simply typed and their predicates are boolean-typed terms in the calculus. In contrast, our calculus is dependently typed and we employ a separate language of predicates (first-order predicate logic).

Following Wadler and others [9], we develop the corresponding coercion calculus and define a translation between the calculi that embodies the simplification motivated by the above examples. While our initial development happens in a setting with first order logic with abstract predicates in types, we subsequently instantiate abstract predicates to linear integer constraints and demonstrate that finding the simplified run-time checks amounts to a synthesis problem. We implemented this synthesis procedure using the Rosette system [12,13] and performed some experiments.

2 The Hybrid Blame Calculus

The main difference between our hybrid blame calculus λ_{hb} and previous work on hybrid type checking and blame calculi is that refinements are formed *using a separate language of predicates* that only shares (first-order) variables, constants, and primitive operations with the usual term language. In particular, predicates are not subject to evaluation, but a predicate P is checked by appealing to an external solver written as $\models P$.

2.1 Syntax and Dynamics

Figure 1 defines the syntax of λ_{hb}, the hybrid blame calculus. The term language of λ_{hb} comprises variables, abstractions, applications, constants, and casts $M : S \Rightarrow T$ from source S to target T. First-order constants are (at least) booleans and integers; we write $\lceil m \rceil$ for the syntactic encoding of the integer m. Higher-order constants include primitive operations; in examples, we freely use infix notation for familiar arithmetic operations.

A type is either a refinement of a base type $x{:}B\{P\}$, where B is a base type (e.g., booleans and integers) and P is a first-order logic predicate where all variables range over base types. Predicates are constructed from an open set of atomic predicates that includes equality between primitive terms (consisting of variables, constants, and primitive operations), the standard logical connectives, and existential quantification over a base type variable.

We define a small-step dynamics of the language using as values abstractions, constants, and function casts applied to a value. Evaluation contexts are defined in the usual way to specify left-to-right call-by-value evaluation.

The dynamics are defined using three judgments on closed terms, $M \longrightarrow N$ for an evaluation step, $M \longrightarrow \mathsf{blame}$ to indicate a failed cast, and $M\ \mathsf{value}$ to test for a syntactic value. Evaluation steps are beta-value reduction and delta reduction, the rules for dealing with casts, and the obvious context rules. We do not give a full definition for delta reduction but provide two examples for addition and remainder.

The rules for casts at base type are standard. The rule Cast-Refine deals with a successful cast between two refinements: it checks that the target predicate Q holds on the cast's subject m. The companion rule Cast-Refine-Fail applies if the cast does not succeed (i.e., if Q does not hold).

For function casts, the astute reader may expect a rule reminiscent of the standard rule introduced by Findler and Felleisen [3], that is

$$\textsc{Cast-Fun-FF}$$
$$(V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \longrightarrow (V\, (W : S' \Rightarrow S)) : T \Rightarrow T'$$

However, this rule disregards the variable x that may occur in T and T'. Knowles and Flanagan [7] apply the cast eagerly to a value by wrapping it in a suitable conversion function. A transliteration of their rule to our setting yields the following rule, which amounts to the picky semantics of dependent casts:

$$\textsc{Cast-Fun-KF}$$
$$\begin{array}{l} \quad (V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \\ \longrightarrow (\lambda x.(V\, (x : S' \Rightarrow S)) : (T[x : S' \Rightarrow S/x]) \Rightarrow T')\, W \end{array}$$

Unfortunately, the rule Cast-Fun-KF breaks type preservation in our system because it substitutes a cast into a type: $T[x : S' \Rightarrow S/x]$. This substitution is syntactically forbidden in our calculus to avoid evaluation inside of types: any free occurrence of x in T is in a predicate and the syntax of predicates excludes casts. This restriction is necessary because a failing cast in a type would be nonsensical. Knowles and Flanagan seem to get around this problem by allowing arbitrary terms to be substituted into types, but omitting a conversion rule that would evaluate inside types.

For these reasons, we split the function cast rule in three rules that perform the argument cast in the assumption. The rule Cast-Arg-Fail applies if the first-order cast $S' \Rightarrow S$ fails on the function's argument. The assumption of this rule corresponds to a single application of rule Cast-Refine-Fail.

The rule Cast-Arg-Base applies if $S' \Rightarrow S$ is a successful first-order refinement check on the argument W, that is, its assumption corresponds to a single application of rule Cast-Refine. After the successful cast, we know that $W : S'$ and $W : S$ both hold, so that we can safely substitute the value W in T and T'.

The rule Cast-Arg-Fun applies if the argument cast $S' \Rightarrow S$ is a function cast, which is tested using the $\cdot\ \mathsf{value}$ judgment. In this case x cannot occur free in either T or T' because variables that occur in refinements are restricted to first-order base types. Thus, the functions involved are not truly dependent and the traditional reduction rule Cast-Fun-FF for non-dependent function casts is applicable.

2.2 Statics

Figure 2 contains the definitions of type environments Γ, ground types G (which strip refinements and dependencies from types and thus amount to simple types), and ground type environments Δ. The figure also defines two auxiliary functions. The function $\lfloor\cdot\rfloor$ maps a type to its underlying ground type (i.e., it maps a refinement to its underlying base type and a dependent function type to the

$M, N ::= x \mid \lambda x.M \mid M\,N \mid C \mid M : S \Rightarrow T$	terms
$C ::= c \mid + \mid \cdot \mid \ldots$	constants
$c ::= \mathsf{true} \mid \mathsf{false} \mid \lceil m \rceil \mid \ldots$	first-order constants
$S, T ::= x : B\{P\} \mid (x : S) \to T$	types
$B ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots$	base types
$P, Q, R ::= A \mid \neg P \mid P \supset Q \mid \exists x : B.P$	predicates
$A ::= \mathsf{true} \mid \mathsf{false} \mid u = v \mid \ldots$	atoms
$u, v ::= c \mid x \mid u + v \mid c \cdot u$	primitive terms
$V, W ::= \lambda x.M \mid C \mid V : (x : S) \to T \Rightarrow (x : S') \to T'$	values
$E ::= \Box\,N \mid V\,\Box \mid \Box : S \Rightarrow T$	evaluation contexts

Dynamics

Beta-Value
$$(\lambda x.M)\,W \longrightarrow M[W/x]$$

Delta
$$\frac{c \in dom(\delta_C)}{C\,c \longrightarrow \delta_C(c)}$$

Value
$$W \ \mathsf{value}$$

Cast-Refine
$$\frac{\models Q[c/x]}{c : (x : B\{P\}) \Rightarrow (x : B\{Q\}) \longrightarrow c}$$

$$\frac{M \longrightarrow N}{E[M] \longrightarrow E[N]}$$

Cast-Refine-Fail
$$\frac{\models \neg Q[c/x]}{c : (x : B\{P\}) \Rightarrow (x : B\{Q\}) \longrightarrow \mathsf{blame}}$$

$$\frac{M \longrightarrow \mathsf{blame}}{E[M] \longrightarrow \mathsf{blame}}$$

Cast-Arg-Fail
$$\frac{(W : S' \Rightarrow S) \longrightarrow \mathsf{blame}}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\,W \longrightarrow \mathsf{blame}}$$

Cast-Arg-Base
$$\frac{(W : S' \Rightarrow S) \longrightarrow W}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\,W \longrightarrow (V\,W) : (T \Rightarrow T')[W/x]}$$

Cast-Arg-Fun
$$\frac{(W : S' \Rightarrow S)\ \mathsf{value}}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\,W \longrightarrow (V\,(W : S' \Rightarrow S)) : (T \Rightarrow T')}$$

Examples for primitive operations

$$\delta_{+}(\lceil m \rceil) = +_m \qquad \delta_{\%}(\lceil m \rceil) = \%_m$$
$$\delta_{+_m}(\lceil n \rceil) = \lceil m + n \rceil \qquad \delta_{\%_m}(\lceil n \rceil) = \lceil m \% n \rceil \text{ if } n \neq 0$$

Fig. 1. Syntax and dynamics of hybrid blame calculus

$$\begin{array}{lll} \Gamma ::= \cdot \mid x : S, \Gamma & & \text{type environments} \\ G ::= B \mid G \to G & & \text{ground types} \\ \Delta ::= \cdot \mid x : G, \Delta & & \text{ground environments} \end{array}$$

Shape of a dependent type; extracting a ground environment

$$\begin{array}{ll} \lfloor x : B\{P\} \rfloor = B & \lfloor \cdot \rfloor = \cdot \\ \lfloor (x : S) \to T \rfloor = \lfloor S \rfloor \to \lfloor T \rfloor & \lfloor x : S, \Gamma \rfloor = x : \lfloor S \rfloor, \lfloor \Gamma \rfloor \end{array}$$

Extracting first-order quantification from type environment

$$\begin{aligned} \forall [\, \cdot \,].P &= P \\ \forall [x : B\{Q\}, \Gamma].P &= \forall x : B.Q \supset \forall [\Gamma].P \\ \forall [x : (y : S) \to T, \Gamma].P &= \forall [\Gamma].P \end{aligned}$$

Fig. 2. Environments and environment manipulation

corresponding plain function type) and which is lifted pointwise to type environments. The last part of the figure defines a function $\forall[\Gamma].P$ that extracts all first-order quantifications from typing environment Γ into a sequence of universal quantifications which is prepended to predicate P.

Figure 3 contains the definitions of the judgments for the statics. The first group of formation rules deals with the judgments

- $\vdash \Gamma$ ctx for valid contexts,
- $\Gamma \vdash S$ type for well-formed types,
- $\Delta \vdash P$ pred for well-formed predicates,
- $\Delta \vdash A$ atom for atomic predicates,
- $\Delta \vdash u$ term for first-order terms (see Fig. 8).

The definition of atom for atomic predicates is intentionally open-ended. This definition only includes the essential constructs to define the calculus. A concrete instance is expected to provide further predicates. The rules are unsurprising, except that the predicate in a refinement type may refer to all (first-order) variables in scope.

The typing rules for variables, abstractions, and application are standard for dependently typed lambda calculi. The application rule has an additional premise (taken from Sjöberg and others [10]) to adapt to a call-by-value calculus with effects—our sole effect is a cast failure. The additional premise requires the result type to be well-formed after substituting the argument term for the argument variable. This requirement forces terms in predicates to remain primitive and thus free of effects.

The rule for constants, which include primitive operations, is inspired by the corresponding definitions for hybrid type checking [7]. It relies on a function

Formation

$$\Delta \vdash \mathsf{true\ atom} \qquad \Delta \vdash \mathsf{false\ atom} \qquad \frac{\Delta \vdash u\ \mathsf{term} \qquad \Delta \vdash v\ \mathsf{term}}{\Delta \vdash u = v\ \mathsf{atom}}$$

$$\frac{\Delta \vdash A\ \mathsf{atom}}{\Delta \vdash A\ \mathsf{pred}} \qquad \frac{\Delta \vdash P\ \mathsf{pred}}{\Delta \vdash \neg P\ \mathsf{pred}} \qquad \frac{\Delta \vdash P\ \mathsf{pred} \qquad \Delta \vdash Q\ \mathsf{pred}}{\Delta \vdash P \supset Q\ \mathsf{pred}}$$

$$\frac{\Delta, x : B \vdash P\ \mathsf{pred}}{\Delta \vdash \exists x : B.P\ \mathsf{pred}}$$

$$\frac{\vdash \Gamma\ \mathsf{ctx} \qquad \lfloor \Gamma \rfloor, x : B \vdash P\ \mathsf{pred}}{\Gamma \vdash x{:}B\{P\}\ \mathsf{type}} \qquad \frac{\Gamma \vdash S\ \mathsf{type} \qquad \Gamma, x : S \vdash T\ \mathsf{type}}{\Gamma \vdash (x : S) \to T\ \mathsf{type}}$$

$$\vdash \cdot\ \mathsf{ctx} \qquad \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma \vdash S\ \mathsf{type}}{\vdash \Gamma, x : S\ \mathsf{ctx}}$$

Typing

$$\frac{\vdash \Gamma, x : T, \Gamma'\ \mathsf{ctx}}{\Gamma, x : T, \Gamma' \vdash x : T} \qquad \frac{\Gamma \vdash S\ \mathsf{type} \qquad \Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.M : (x : S) \to T}$$

$$\frac{\Gamma \vdash M : (x : S) \to T \qquad \Gamma \vdash N : S \qquad \Gamma \vdash T[N/x]\ \mathsf{type}}{\Gamma \vdash M\,N : T[N/x]} \qquad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash C : \mathit{Ty}(C)}$$

$$\frac{\Gamma \vdash M : S \qquad \Gamma \vdash T\ \mathsf{type} \qquad \lfloor S \rfloor = \lfloor T \rfloor}{\Gamma \vdash (M : S \Rightarrow T) : T} \qquad \frac{\Gamma \vdash M : S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash M : T}$$

Subtyping

$$\frac{\vdash \Gamma\ \mathsf{ctx} \qquad \lfloor \Gamma \rfloor, x : B \vdash P \supset Q\ \mathsf{pred} \qquad \models \forall \bigl[\Gamma, x : x{:}B\{P\}\bigr].Q}{\Gamma \vdash x{:}B\{P\} \leq x{:}B\{Q\}}$$

$$\frac{\Gamma \vdash S'\ \mathsf{type} \qquad \Gamma, x : S' \vdash T'\ \mathsf{type} \qquad \Gamma \vdash S' \leq S \qquad \Gamma, x : S' \vdash T \leq T'}{\Gamma \vdash (x : S) \to T \leq (x : S') \to T'}$$

Typing of constants (excerpt cf. [7])

$$\begin{aligned}
\mathit{Ty}(\lceil m \rceil) &= x{:}\mathsf{int}\{x = m\} \\
\mathit{Ty}(\mathsf{true}) &= x{:}\mathsf{bool}\{x = \mathsf{true}\} \\
\mathit{Ty}(+) &= x{:}\mathsf{int}\{P\} \to (y{:}\mathsf{int}\{Q\} \to z{:}\mathsf{int}\{\exists xy.P \wedge Q \wedge z = x + y\}) \\
\mathit{Ty}(+_m) &= y{:}\mathsf{int}\{Q\} \to z{:}\mathsf{int}\{\exists y.Q \wedge z = m + y\} \\
\mathit{Ty}(\%) &= x{:}\mathsf{int}\{P\} \to (y{:}\mathsf{int}\{Q \wedge y \neq 0\} \to z{:}\mathsf{int}\{\exists xy.P \wedge Q \wedge z = x\%y\}) \\
\mathit{Ty}(\%_m) &= y{:}\mathsf{int}\{Q \wedge y \neq 0\} \to z{:}\mathsf{int}\{\exists y.Q \wedge z = x\%y\}
\end{aligned}$$

Fig. 3. Typing for the hybrid blame calculus

$Ty(C)$ that maps a constant to a type. The bottom part of Fig. 3 gives some examples. The rule for cast requires that the source and target type, S and T, of a cast share the same underlying shape $\lfloor S \rfloor = \lfloor T \rfloor$. The subsumption rule is standard.

Subtyping between refinements is generated by implication of the predicates, which is proved using the external solver $\models P \supset Q$. As further preconditions, we take the first-order knowledge accumulated in the environment. The function $\forall[\,\cdot\,]$ (see Fig. 2) performs this task. It skips over variables of function type and poses first-order knowledge as a precondition. The subtyping rule for dependent function types is taken straight from Aspinall and Compagnoni's work [1].

3 The Hybrid Coercion Calculus

We define the hybrid coercion calculus λ_{hc} as an extension of the hybrid blame calculus λ_{hb} by coercions. Usually, a program is written entirely using casts or using coercions, but our calculus admits an arbitrary mix. This view is supported by the existence of translations in both directions elaborated in Sect. 4.

3.1 Syntax and Dynamics

Figure 4 contains the new syntactic elements and evaluation rules. There is a new term for coercion application, where coercions are either primitive coercions between refinements, written as $x.R$, or dependent function coercions, written as $(x : k) \to d$. The set of values is extended by the application of a function coercion to a value.

The primitive coercion $x.R$ just states the predicate R to check on the value using the same solver as before. See rules CoerceRefine and Coerce-Refine-Fail.

The picky semantics of the function coercion has the same subtleties as the function cast, so it is also implemented by three reduction rules: rule Coerce-Arg-Fail raises an exception if the first-order argument cast fails; rule Coerce-Arg-Base continues with the function call if the argument cast succeeds; and rule Coerce-Arg-Fun applies if the coercion application is a value, in which case no substitution for x is needed.

3.2 Statics

Figure 5 contains the typing rules for coercions and the obvious rule for coercion application. The rules for coercions define a judgment $\Gamma \vdash k :: S \Longrightarrow T$ that affirms that coercion k maps from type S to type T under assumptions Γ. The main workhorse of this judgment is the typing rule for the primitive coercion between base type refinements. The obvious rule would state that $x.R$ coerces from $x\{P\}$ to $x\{R\}$. Such a rule would be correct, but it totally ignores the knowledge that predicate P already holds for x. Clearly, knowing that P holds and checking R successfully establishes any predicate Q such that $P \wedge R \supset Q$.

Syntax

$$\begin{array}{rll} M, N ::= \cdots \mid M\langle k\rangle & & \text{Coercion application} \\ k, d ::= x.R \mid (x : k) \to d & & \text{Coercions} \\ V, W ::= \cdots \mid V\langle (x : k) \to d\rangle & & \text{Values} \end{array}$$

Dynamics

$$\textsc{Coerce-Refine}\ \frac{\models R[c/x]}{c\langle x.R\rangle \longrightarrow c} \qquad \textsc{Coerce-Refine-Fail}\ \frac{\models \neg R[c/x]}{c\langle x.R\rangle \longrightarrow \ \mathsf{blame}}$$

$$\textsc{Coerce-Arg-Fail}\ \frac{W\langle k\rangle \longrightarrow \ \mathsf{blame}}{(V\langle (x : (x.R)) \to d\rangle)\, W \longrightarrow \ \mathsf{blame}}$$

$$\textsc{Coerce-Arg-Base}\ \frac{W\langle k\rangle \longrightarrow W}{(V\langle (x : k) \to d\rangle)\, W \longrightarrow (V\, W)\langle d[W/x]\rangle}$$

$$\textsc{Coerce-Arg-Fun}\ \frac{W\langle k\rangle \ \mathsf{value}}{(V\langle (x : k) \to d\rangle)\, W \longrightarrow (V\, (W\langle k\rangle))\langle d\rangle}$$

Fig. 4. Coercion calculus

$$\frac{\forall[\Gamma]\ \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor\Gamma\rfloor, x : B \vdash P, Q, R\ \mathsf{pred}}{\Gamma \vdash x.R :: x : B\{P\} \Longrightarrow x : B\{Q\}}$$

$$\frac{\Gamma \vdash k :: S' \Longrightarrow S \qquad \Gamma, x : S \curlywedge S' \vdash d :: T \Longrightarrow T'}{\Gamma \vdash (x : k) \to d :: (x : S) \to T \Longrightarrow (x : S') \to T'}$$

$$\frac{\Gamma \vdash M : S \qquad \Gamma \vdash k :: S \Longrightarrow T}{\Gamma \vdash M\langle k\rangle : T}$$

Fig. 5. Typing for coercion calculus

To establish a given predicate Q, the task is now to find a suitable R to check. Clearly, such a predicate exists. For example, $R = Q$ or $R = \mathsf{false}$ would work. However, it would be useless if R was overly restrictive; for example, checking the extreme case $R = \mathsf{false}$ would make the cast $x.R$ fail always, but establish any Q nevertheless. Hence, the idea is to choose some test predicate R that is equivalent to Q when P is assumed: find a predicate R such that $P \supset (Q \Leftrightarrow R)$. Such an R also exists (e.g., $R = Q$), but there may be less restrictive test predicates that are cheaper to check than Q. For an extreme example, if $P \supset Q$ already, then $x\{P\}$ is a subtype of $x\{Q\}$ and we can choose $R = \mathsf{true}$ as test predicate.

For the typing of the dependent function coercion $(x : k) \to d$, we need to extract an assumption on the type of x from the coercion $k :: S' \Longrightarrow S$ to derive the type of d. While the shapes of S and S' are the same, it is not clear which of those types should be assumed for x. Unlike the case for subtyping, S and S' need not be in a subtyping relationship. Furthermore, S may only make sense as an argument type with T, but not with T', and vice versa. For that reason, we use $S \curlywedge S'$, a conjunction of S and S that is restricted to first-order refinements. It is defined only for types of the same shape:

$$S \curlywedge S' = \begin{cases} x : B\{P \wedge P\}' & S = x : B\{P\}, S' = x : B\{P\}' \\ S & \lfloor S \rfloor = \lfloor S' \rfloor = G \to G' \end{cases}$$

For function types, the choice does not matter because the type of x does not influence the typing.

4 Translations

Casts can be translated to coercions and vice versa. The translations are nondeterministic to encompass the design space for an actual implementation. However, the nondeterminism is not essential to obtain a correct solution. Any local nondeterministic choice can be completed to a full translation.

Typically, a compiler would translate from casts to coercions and exploit the nondeterminism to choose "minimal" test predicates in the primitive coercions. We use the word "minimal" informally to mean that the test should be executable in as few cycles as possible. In our experiments (Sect. 6), we minimize the size of a syntactic representation for the predicate.

Figure 6 contains the nondeterministic specification of the translation from casts to coercions, that is, from λ_{hb} to λ_{hc}. It is defined as a judgment $\Gamma \vdash S \Rightarrow T \rightsquigarrow k$ where coercion k is the result of translating the cast from S to T under assumptions Γ. For first-order refinements, the translation facilitates simplification: the cast from $x\{P\}$ to $x\{Q\}$ is translated to a coercion $x.R$,

$$\frac{\forall[\Gamma]\ \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor\Gamma\rfloor, x : B \vdash P, Q, R\ \mathsf{pred}}{\Gamma \vdash x : B\{P\} \Rightarrow x : B\{Q\} \rightsquigarrow x.R}$$

$$\frac{\Gamma \vdash S' \Rightarrow S \rightsquigarrow k \qquad \Gamma, x : S \curlywedge S' \vdash T \Rightarrow T' \rightsquigarrow d}{\Gamma \vdash (x : S) \to T \Rightarrow (x : S') \to T' \rightsquigarrow (x : k) \to d}$$

Fig. 6. Translation from blame to coercions

$$\frac{\forall\lceil\Gamma\rceil\,\forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor\Gamma\rfloor, x : B \vdash P, Q, R \ \textsf{pred}}{\Gamma \vdash x.R \rightsquigarrow x{:}B\{P\} \Rightarrow x{:}B\{Q\}}$$

$$\frac{\Gamma \vdash k \rightsquigarrow S' \Rightarrow S \qquad \Gamma, x : S \curlywedge S' \vdash d \rightsquigarrow T \Rightarrow T'}{\Gamma \vdash (x : k) \to d \rightsquigarrow (x : S) \to T \Rightarrow (x : S') \to T'}$$

Fig. 7. Translation from coercions to blame

where R and Q are equivalent when assuming Γ and P. This choice is not empty, because choosing $R = Q$ is always possible.

The rule for translating a function cast to a function coercion is unsurprising up to one point: When it comes to picking the assumption for the type of x in the translation the result part of the function coercion, we can choose between S and S' because when evaluation reaches the function return and S and S' are refinement types, then we certainly know that $x : S$ and $x : S'$. If conjunction was part of the type language, we could even have chosen $x : S \wedge S'$. On the other hand, if the types S and S' are function types, then the choice does not matter because the type of x cannot influence the translation of $T \Rightarrow T'$. As in the coercion typing, we use the first-order conjunction $S \curlywedge S'$ in the assumption.

Figure 7 contains the nondeterministic translation from coercions to casts. It is defined by the judgment $\Gamma \vdash k \rightsquigarrow S \Rightarrow T$ which relates a coercion k to the cast from S to T under assumptions Γ. The base type coercion $x.R$ is related to any cast from $x\{P\}$ to $x\{Q\}$ where Q and R are equivalent assuming Γ and P (as seen before).

For the translation of function coercions, the same remarks apply as for the reverse direction. To ensure correctness, the handling of the argument type must match its handling in the reverse translation.

5 Results

This section collects some results of the metatheory of λ_{hb} and λ_{hc} that are important for the purposes of the present paper. The results are mostly standard.

Lemma 1 (Substitution)**.** *Let J range over formation, typing, and subtyping judgments.*

If $\Gamma, x : S, \Gamma' \vdash J$ and $\Gamma \vdash W : S$, then $\Gamma \vdash J[W/x]$.

Proposition 1 (Narrowing)**.** *Let J range over formation, typing, and subtyping judgments.*

Suppose that $\Gamma, x : T, \Gamma' \vdash J$ and $\Gamma \vdash S \leq T$. Then $\Gamma, x : S, \Gamma' \vdash J$.

Proof. By simultaneous induction on all judgments. There are two key steps.

1. In the variable case for x, we have $\Gamma, x : S, \Gamma' \vdash x : S$ and obtain $x : T$ by subsumption.

2. In the subset-type case for subtyping, S and T may be assumed to be subset types with predicates P_x and Q_x (otherwise, x is removed from the assumptions by the $\forall[\_]$ operation and the subtyping is independent of x), so that $\models \forall[\Gamma].P_x \supset Q_x$. From here, it is easy to conclude that subtyping is preserved. □

Subtyping is reflexive and transitive.

Lemma 2.

1. *For all S, $\Gamma \vdash S \leq S$.*
2. *For all S, S', S'', if $\Gamma \vdash S \leq S'$ and $\Gamma \vdash S \leq S''$, then $\Gamma \vdash S \leq S''$.*

Here are some results, which are specific for λ_{hb} and λ_{hc}. Subtyping is only possible between types of the same shape.

Lemma 3. *If $\Gamma \vdash S \leq T$, then $\lfloor S \rfloor = \lfloor T \rfloor$.*

Subtyping judgments are independent of type assumptions that concern function types. This result holds because refinements are restricted to first-order predicates, so that only first-order typed variables can influence subtyping.

Lemma 4. *Let $\Gamma \vdash S$ type and $\Gamma \vdash S'$ type with $\lfloor S \rfloor = \lfloor S' \rfloor = G \rightarrow G'$. If $\Gamma, x : S, \Gamma' \vdash T \leq T'$, then $\Gamma, x : S', \Gamma' \vdash T \leq T'$.*

Taken together with narrowing, we obtain the following result about first-order conjunction.

Lemma 5. *Let $\Gamma \vdash S$ type and $\Gamma \vdash S'$ type with $\lfloor S \rfloor = \lfloor S' \rfloor$. $\Gamma, x : S, \Gamma' \vdash T \leq T'$ implies $\Gamma, x : S \curlywedge S', \Gamma' \vdash T \leq T'$.*

Proof. If $\lfloor S \rfloor = B$, then $\Gamma \vdash S \curlywedge S' \leq S$, and we conclude by narrowing. Otherwise, apply Lemma 4. □

The combined calculus λ_{hb} and λ_{hc} enjoys type soundness. This result is proven in Wright-Felleisen style [15] by establishing type preservation and progress. Details may be found in the appendix.

Proposition 2 (Type Preservation for λ_{hb} and λ_{hc}). *If $\cdot \vdash M : T$ and $M \longrightarrow N$, then $\cdot \vdash N : T$.*

Proposition 3 (Progress). *Suppose that $\cdot \vdash M : T$. Then exactly one of the following holds:*

1. *$\exists N$ such that $M \longrightarrow N$;*
2. *M value;*
3. *$M \longrightarrow$ blame.*

The next result is specific to our calculus and establishes a tight connection between the translations from casts to coercions and back and typing of a coercion. The final piece would relate to the typing of a cast $S \Rightarrow T$, which is trival given the typing rule for casts.

Proposition 4. *The following three statements are equivalent.*

1. $\Gamma \vdash S \Rightarrow T \rightsquigarrow k$.
2. $\Gamma \vdash k \rightsquigarrow S \Rightarrow T$.
3. $\Gamma \vdash k :: S \Longrightarrow T$.

Proof. By rule induction on the translation. □

$$\frac{}{\Delta \vdash c \ \mathsf{term}} \qquad \frac{}{\Delta, x : B \vdash x \ \mathsf{term}} \qquad \frac{\Delta \vdash u \ \mathsf{term}}{\Delta \vdash c \cdot u \ \mathsf{term}}$$

$$\frac{\Delta \vdash u \ \mathsf{term} \qquad \Delta \vdash v \ \mathsf{term}}{\Delta \vdash u + v \ \mathsf{term}}$$

$$\frac{\Delta \vdash u \ \mathsf{term} \qquad \Delta \vdash v \ \mathsf{term}}{\Delta \vdash u \leq v \ \mathsf{atom}} \qquad \frac{\Delta \vdash u \ \mathsf{term} \qquad \Delta \vdash v \ \mathsf{term}}{\Delta \vdash u = v \ \mathsf{atom}}$$

$$\Delta, x : \mathsf{bool} \vdash x \ \mathsf{atom}$$

Fig. 8. Formation rules for atomic predicates

6 A Concrete Instance

To consider a concrete instance of the hybrid system, we choose boolean values and linear inequations as atomic predicates and we make the translation to coercions deterministic.

$$\begin{array}{rll} A ::= & x \mid u = v \mid u \leq v & \text{atomic predicates} \\ u, v ::= & c \cdot u \mid c \mid x \mid u + v & \text{first-order linear terms} \end{array}$$

Figure 8 contains the formation rules for atoms.

Recall that the refinement rule is the source of nondeterminism in the translation from casts to coercions (Fig. 6):

$$\frac{\forall[\Gamma]\, \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R \ \mathsf{pred}}{\Gamma \vdash x{:}B\{P\} \Rightarrow x{:}B\{Q\} \rightsquigarrow x.R}$$

The key to making this rule deterministic is to realize that finding a suitable predicate R for the coercion can be considered as a synthesis problem that can be solved with a system like Rosette [12,13]. To this end, we first exhibit a grammar that restricts the search space for R when Γ, x, P, and Q are given.

$$\begin{array}{rl} r ::= & \mathsf{true} \mid \mathsf{false} \mid p \\ p, q ::= & z \leq c \mid c \leq z \mid z \leq y \mid \neg p \mid p \wedge q \mid p \vee q \end{array}$$

```
(define r (grammar vars depth))
(evaluate
 r
 (synthesize
  #:forall vars
  #:guarantee
  (assert
   (=> (interpret p) (<=> (interpret q) (interpret r))))))
```

Listing 1. Synthesis code for Rosette

We obtain a finite set of candidates by restricting the variables y and z to range over the variables quantified in $\forall\lceil\Gamma\rceil$ and x (but with $y \neq z$), having c range over integer constants representable in a small number of bits, and by imposing a depth bound on terms. Then we define an interpretation function that maps a predicate r to true or false. This function can be used to drive the synthesis as demonstrated in the code fragment in Listing 1.

The function `grammar` is parameterized over the variables as just explained and a depth bound. It produces a representation of r as a symbolic value suitable for synthesis. The function `synthesize` finds a model of the assertion (if one exists) and function `evaluate` takes a symbolic value `r` and converts it to a concrete value using that model. In our example, the model determines all choices that need to be taken when deriving a predicate from the grammar for r and `evaluate` fixes these choices and returns a syntactic representation of the resulting predicate.

The code fragment in Listing 1 is good enough for obtaining a solution, but we are interested in a *minimal solution*. To this end, we define a cost function on predicates:

$$\begin{aligned} \mathit{cost}(\mathsf{true}) = \mathit{cost}(\mathsf{false}) &= 0 \\ \mathit{cost}(z \leq c) = \mathit{cost}(c \leq z) = \mathit{cost}(y \leq z) &= 1 \\ \mathit{cost}(\neg p) &= 1 + \mathit{cost}(p) \\ \mathit{cost}(p \wedge q) = \mathit{cost}(p \vee q) &= 1 + \mathit{cost}(p) + \mathit{cost}(q) \end{aligned}$$

Then we instruct the synthesizer to find a solution with minimal cost. Doing so is slightly more involved, but recent versions of the system support synthesis queries including minimzation out of the box [2].

We applied our system to a range of examples, which are summarized in Table 1. For readability, we sometimes contracted conjunctions like $a \leq x \wedge x \leq b$ to $a \leq x \leq b$ and reordered literals in the output. The first two lines are concerned with overlapping intervals. In each case, the tested predicate only includes the tighened bound. The third line deals with overlapping rectangles where the y dimension is properly nested so that both bounds need to be tested. The required predicate in the fourth line carves a triangle out of a rectangle such that only the test $x \leq y$ is required. Line 5 assumes a disjunction of intervals and

Table 1. Delta predicates R synthesized by Rosette

P given	Q required	R tested
$1 \leq x \wedge x \leq 10$	$5 \leq x \wedge x \leq 15$	$5 \leq x$
$5 \leq x \wedge x \leq 15$	$1 \leq x \wedge x \leq 10$	$x \leq 10$
$1 \leq x \leq 10 \wedge 0 \leq y \leq 9$	$5 \leq x \leq 15 \wedge 3 \leq y \leq 6$	$3 \leq y \leq 6 \wedge 5 \leq x$
$1 \leq x \leq 9 \wedge 1 \leq y \leq 5$	$1 \leq x \leq 5 \wedge x \leq y$	$x \leq y$
$0 \leq x \leq 5 \vee 10 \leq x \leq 15$	$6 \leq x \leq 9$	false
$6 \leq x \leq 9$	$5 \leq x \leq 10$	true

requires a disjoint interval. As the intersection of P and Q is empty, the minimal test is false. In the last line, the requirement is weaker than the assumption, so no test is needed.

7 More Manifestation

The results on the translation from λ_{hb} into the coercion calculus λ_{hc} show that it is possible to simplify the predicate that is tested at run time by taking into account what is already known at this point. However, this additional knowledge is not manifested in the type system and one may wonder about the ramifications of such a manifestation.

As an example, consider the cast

$$x : \mathsf{int}\{x > 0\} \Rightarrow x : \mathsf{int}\{x < 10\} \tag{1}$$

While the run-time test for checking this cast cannot be simplified, we could imagine a typing rule that manifests the entire accumulated knowledge: Clearly, a term whose value passes the cast (1) fulfills $x > 0$ and $x < 10$ so that it could be given type $x : \mathsf{int}\{x > 0 \wedge x < 10\}$.

Obtaining such a refined typing requires a separate typing judgment $\Delta \vdash S \Rightarrow T :: S' \Longrightarrow T'$ for casts because a cast $S \Rightarrow T$ will now transform values between types S' and T', as determined by the judgment, which is defined in Fig. 9. The only interesting rule is the one between refinements $x\{P\}$ and $x\{Q\}$ which concludes $x\{P \wedge Q\}$. The rule for functions is the obvious one, but it forces using a ground type environment for the judgment. Otherwise, it would not be clear whether $x : S_1$ or $x : S_2$ should be added to the type environment, but their ground types are equal.

The typing rules for the coercion calculus from Fig. 5 can be refined in a similar way. The rule for base type coercions is very similar to the one for casts while the rule for function coercions remains as before.

$$\frac{\lfloor \Gamma \rfloor, x : B \vdash P, R \ \mathsf{pred}}{\Gamma \vdash x.R :: x : B\{P\} \Longrightarrow x : B\{P \wedge R\}}$$

The first question to ask about the typing rules for casts is about the relation between the advertised type of a cast and its inferred type.

$$\frac{\Delta \vdash Q \textsf{ pred}}{\Delta \vdash x\!:\!B\{P\} \Rightarrow x\!:\!B\{Q\} :: x\!:\!B\{P\} \Longrightarrow x\!:\!B\{P \wedge Q\}}$$

$$\frac{\Delta \vdash S_2 \Rightarrow S_1 :: S_2' \Longrightarrow S_1' \qquad \Delta, x : \lfloor S_1 \rfloor \vdash T_1 \Rightarrow T_2 :: T_1' \Longrightarrow T_2'}{\Delta \vdash (x : S_1) \to T_1 \Rightarrow (x : S_2) \to T_2 :: (x : S_1') \to T_1' \Longrightarrow (x : S_2') \to T_2'}$$

$$\frac{\Gamma \vdash M : S' \qquad \lfloor \Gamma \rfloor \vdash S \Rightarrow T :: S' \Longrightarrow T'}{\Gamma \vdash (M : S \Rightarrow T) : T'}$$

Fig. 9. Precise typing for casts

Proposition 5. *If* $\lfloor \Gamma \rfloor \vdash S \Rightarrow T :: S' \Longrightarrow T'$*, then* $\Gamma \vdash S \leq S'$ *and* $\Gamma \vdash T' \leq T$*.*

Proof. By rule induction on the typing of casts.

Case $\dfrac{\lfloor \Gamma \rfloor \vdash Q \textsf{ pred}}{\lfloor \Gamma \rfloor \vdash x\!:\!B\{P\} \Rightarrow x\!:\!B\{Q\} :: x\!:\!B\{P\} \Longrightarrow x\!:\!B\{P \wedge Q\}}.$

1. By reflexivity, $\Gamma \vdash x\!:\!B\{P\} \leq x\!:\!B\{P\}$.
2. From $\forall \lceil \Gamma \rceil \forall x : B.P \wedge Q \supset Q$, we have $\Gamma \vdash x\!:\!B\{P \wedge Q\} \leq x\!:\!B\{Q\}$.

Case $\dfrac{\lfloor \Gamma \rfloor \vdash S_2 \Rightarrow S_1 :: S_2' \Longrightarrow S_1' \qquad \lfloor \Gamma \rfloor, x : \lfloor S_1 \rfloor \vdash T_1 \Rightarrow T_2 :: T_1' \Longrightarrow T_2'}{\lfloor \Gamma \rfloor \vdash (x : S_1) \to T_1 \Rightarrow (x : S_2) \to T_2 :: (x : S_1') \to T_1' \Longrightarrow (x : S_2') \to T_2'}.$

1. Induction on the premise $\lfloor \Gamma \rfloor \vdash S_2 \Rightarrow S_1 :: S_2' \Longrightarrow S_1'$ yields
   (a) $\Gamma \vdash S_2 \leq S_2'$ and
   (b) $\Gamma \vdash S_1' \leq S_1$
2. Induction on the premise $\lfloor \Gamma \rfloor, x : \lfloor S_1 \rfloor \vdash T_1 \Rightarrow T_2 :: T_1' \Longrightarrow T_2'$ yields
   (a) $\Gamma, x : S_1' \vdash T_1 \leq T_1'$ and
   (b) $\Gamma, x : S_2 \vdash T_2' \leq T_2$
   It is key that the inductive hypothesis permits us to freely choose types for x subject to $\lfloor S_1 \rfloor = \lfloor S_1' \rfloor = \lfloor S_2 \rfloor$.
3. It remains to establish the conclusion
   (a) $\Gamma \vdash (x : S_1) \to T_1 \leq (x : S_1') \to T_1'$ by Item 1b and Item 2a
   (b) $\Gamma \vdash (x : S_2') \to T_2' \leq (x : S_2) \to T_2$ by Item 1a and Item 2b. □

The proof reflects that all that matters in establishing the subtyping relations is the base case for the refinements. The choice of the "preconditions" in Items 2a and 2b does not affect the subtyping relation which is "caused" by the tautology $P \wedge Q \supset Q$.

Examination of the proof shows that, for refinements, the source type of a cast coincides with its inferred source type. This coincidence enables us to find a subtyping relation between the first-order conjunctions as follows.

Lemma 6. *If* $\Gamma \vdash S \Rightarrow T :: S' \Longrightarrow T'$ *and* $\lfloor S \rfloor = B$*, then*

1. $S = S'$ *and* $\Gamma \vdash S' \curlywedge T' \leq S \curlywedge T$*;*
2. $\Gamma \vdash T' \leq S$ *and* $T' = S \curlywedge T$*.*

Proof. If $\lfloor S \rfloor = B$, then $S = x : B\{P\} = S'$ (see first case of proof of Proposition 5). Furthermore, $T = x : B\{Q\}$ and $T' = x : B\{Q'\}$ and $\Gamma \vdash T' \leq T$ by the proposition. By the definition of subtyping, it must be that $\forall\lceil\Gamma\rceil.Q' \supset Q$. Hence, $\forall\lceil\Gamma\rceil.(P \wedge Q') \supset (P \wedge Q)$, which proves that $\Gamma \vdash S' \curlywedge T' \leq S \curlywedge T$.

To see the second part, recall from the proof that $Q' = P \wedge Q$ which means $\forall\lceil\Gamma\rceil.Q' \supset P$ and thus $\Gamma \vdash T' \leq S$. Finally, $S \curlywedge T = x : B\{P \wedge Q\} = T'$. $\square$

With the new typing rules in place, we can ask again whether the translation from casts to coercions preserves types. We write $\Gamma \vdash S \equiv T$ for type equivalence, indicating that S and T are mutual subtypes $\Gamma \vdash S \leq T$ and $\Gamma \vdash T \leq S$.

Conjecture 1. Suppose that $\Gamma \vdash S \Rightarrow T \rightsquigarrow k$ and $\lfloor\Gamma\rfloor \vdash S \Rightarrow T :: S' \Longrightarrow T'$. Then $\Gamma \vdash k :: S'' \Longrightarrow T''$ and $\Gamma \vdash S' \equiv S''$ and $\Gamma \vdash T' \equiv T''$.

Unfortunately, the proof of the conjecture does not go through with the present statement of the translation rules.

8 Related Work

Knowles and Flanagan's work on Hybrid Type Checking [7] (HTC) is most closely related. As already remarked in the text, the metatheory of HTC is slightly different. First, HTC does not commit to a particular evaluation order, whereas our calculi λ_{hb} and λ_{hc} are call-by-value. Second, while we restrict predicates to first-order predicates suitable for synthesis, HTC permits arbitrary terms of type boolean as predicates. Third, neither HTC nor our calculi have a conversion rule for evaluating in types, but our system imposes an additional restriction on the application rule to guarantee well-formed cast- and coercion-free types.

Greenberg and others [4] consider calculi that are closely related to our setting. Their calculus λ_H includes dependent casts, refinement types, and dependent function types. Compared to their work, we restrict to a specific predicate language and we omit blame labels, which would be easy to add.

The idea to relate cast and coercion calculi (introduced in the dynamic typing works of Henglein [5]) and to translate between them goes back to Herman, Tomb, and Flanagan [6]. It was further refined by Siek, Thiemann, and Wadler [9] who give fully abstract translations between calculi with casts, coercions, and space-efficient coercions. However, the latter work is concerned with dynamic typing and considers casts/coercions between types of different precision, whereas the present work considers refinements of a simply-typed language.

Dependent types in connection with casts have been considered by Ou and others [8]. Their language has similar features than ours, but they define a translation into an internal language that implements the coercions inline rather than coming up with a specific coercion calculus. As their language is (also) a call-by-value language that includes computational effects, they restrict dependent function application to pure arguments. Furthermore, our treatment of casts at dependent function types seems related to theirs.

Acknowledgment. Emina Torlak graciously helped getting up to speed with her Rosette system. Jeremy Siek offered some helpful suggestions. Michael Greenberg answered subtle questions about manifest contract systems.

A Proofs

A.1 Preservation for λ_{hb}

Proof of Proposition 2. Induction on the derivation of $M \longrightarrow N$.

Case $(\lambda x.M)\, W \longrightarrow M[W/x]$.

1. Inversion of $\cdot \vdash (\lambda x.M)\, W : T$ yields the next two items
2. $\dfrac{\cdot \vdash (\lambda x.M) : (x : S) \to T' \qquad \cdot \vdash W : S \qquad \cdot \vdash T'[W/x]\ \mathsf{type}}{\cdot \vdash (\lambda x.M)\, W : T'[W/x]}$
3. $\cdot \vdash T'[W/x] \leq T$
4. Inversion on the first premise of Item 2 yields the next two items
5. $x : S' \vdash M : T''$
6. $\cdot \vdash (x : S') \to T'' \leq (x : S) \to T'$ which implies $\cdot \vdash S \leq S'$ and $x : S \vdash T'' \leq T'$
7. From Items 5 and 6 and the premise $\cdot \vdash W : S$ in Item 2, the substitution lemma yields $\cdot \vdash M[W/x] : T''[W/x]$
8. From Item 6 and substitution, we obtain $\cdot \vdash T''[W/x] \leq T'[W/x]$
9. Item 3, transitivity of subtyping, and subsumption yield $\cdot \vdash M[W/x] : T$

Case $\lceil m \rceil + \lceil n \rceil \longrightarrow \lceil m + n \rceil$.

1. Inversion of $\cdot \vdash \lceil m \rceil + \lceil n \rceil : T$ yields the following three items
2. $\cdot \vdash z : \mathsf{int}\{\exists xy.z = x + y \wedge P \wedge Q\} \leq T$
3. $\cdot \vdash \lceil m \rceil : x : \mathsf{int}\{P\}$ where $\cdot \vdash x : \mathsf{int}\{x = m\} \leq x : \mathsf{int}\{P\}$
4. $\cdot \vdash \lceil n \rceil : y : \mathsf{int}\{Q\}$ where $\cdot \vdash y : \mathsf{int}\{y = n\} \leq y : \mathsf{int}\{Q\}$
5. Hence, $x = m \supset P$ and $y = n \supset Q$
6. Now $\cdot \vdash \lceil m + n \rceil : z : \mathsf{int}\{z = m + n\}$
7. The predicate is equivalent to $\exists xy.z = x + y \wedge x = m \wedge y = n$.
8. By Item 5, the predicate in Item 7 implies $\exists xy.z = x + y \wedge P \wedge Q$
9. Taking Items 6, 7, 8, and 2 and applying subsumption yields $\cdot \vdash \lceil m + n \rceil : T$

Case $\dfrac{\models Q[m/x]}{\lceil m \rceil : x : B\{P\} \Rightarrow x : B\{Q\} \longrightarrow \lceil m \rceil}$ (CAST-REFINE).

1. Inversion applied to $\cdot \vdash \lceil m \rceil : x : B\{P\} \Rightarrow x : B\{Q\} : T$ yields
2. $\cdot \vdash x : B\{Q\} \leq T$ and
3. $\cdot \vdash \lceil m \rceil : x : B\{P\}$ and
4. $\models Q[m/x]$
5. For the reduct, $\cdot \vdash \lceil m \rceil : x : B\{x = m\}$
6. Since $\models Q[m/x]$, we have $\models \forall x.x = m \wedge Q$, which establishes the subtyping $\cdot \vdash x : B\{x = m\} \leq x : B\{Q\}$
7. The claim follows by subsumption with Item 2.

Case $$\frac{(W : S' \Rightarrow S) \longrightarrow W}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \longrightarrow (V\, W) : (T \Rightarrow T')[W/x]}\ \textsc{Cast-Arg-Base}.$$

1. Applying inversion to $\cdot \vdash (V : (x : S) \to T \Rightarrow (x : S') \to T')\, W : T'[W/x]$ yields the following items
2. $\cdot \vdash (V : (x : S) \to T \Rightarrow (x : S') \to T') : (x : S') \to T'$
3. $\cdot \vdash W : S'$
4. $\cdot \vdash T'[W/x]$
5. Inverting the premise of Cast-Arg-Base, we further obtain
6. $S = x\!:\!B\{P\}$
7. $S' = x\!:\!B\{P\}'$
8. $\models P'[W/x]$
9. From Item 3 and 6, we obtain $\models P[W/x]$
10. Applying inversion to Item 2, we obtain
11. $\cdot \vdash V : (x : S) \to T$ and
12. $\lfloor (x : S) \to T \rfloor = \lfloor (x : S') \to T' \rfloor$
13. Using Item 9, we obtain $\cdot \vdash W : S$ (with an intermediate subsumption step)
14. Hence $\cdot \vdash T[W/x]$ and $\cdot \vdash V\, W : T[W/x]$
15. We conclude that $\cdot \vdash (V\, W : (T \Rightarrow T')[W/x]) : T'[W/x]$

Case $$\frac{(W : S' \Rightarrow S)\ \mathsf{value}}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \longrightarrow (V\, (W : S' \Rightarrow S)) : (T \Rightarrow T')}\ \textsc{Cast-Arg-Fun}.$$

1. From the premise, we know that $\lfloor S' \rfloor = \lfloor S \rfloor = G \to G'$
2. Applying inversion to $\cdot \vdash (V : (x : S) \to T \Rightarrow (x : S') \to T')\, W : T_0$ yields
3. $T_0 = T'[W/x]$
4. $\cdot \vdash W : S'$
5. $\cdot \vdash (V : (x : S) \to T \Rightarrow (x : S') \to T') : (x : S') \to T'$
6. From Item 1 and 3, we obtain $T_0 = T'[W/x] = T'$ and, in fact, $T'[N/x] = T'$ and $T[N/x] = T$, for any N, because all free variables in types are restricted to base types.
7. Inversion of Item 5 yields the following
8. $\cdot \vdash V : (x : S) \to T$ and
9. $\lfloor (x : S) \to T \rfloor = \lfloor (x : S') \to T' \rfloor$
10. From Item 5, we conclude $\cdot \vdash (W : S' \Rightarrow S) : S$
11. From Item 10 and 8, we obtain $\cdot \vdash (V\, (W : S' \Rightarrow S)) : T[(W : S' \Rightarrow S)/x]$, where Item 6 assures us that $T[(W : S' \Rightarrow S)/x] = T$ is well-formed.
12. We conclude that $\cdot \vdash (V\, (W : S' \Rightarrow S) : (T \Rightarrow T')) : T'$ where $T' = T'[W/x] = T_0$ by Item 6.

Case reduction in context: immediate by application of the inductive hypothesis. $\square$

A.2 Preservation for λ_{hc}

Proof of Proposition 2 ; cases for λ_{hc}. We just state the additional inductive cases for the reductions involving coercions.

Case $\dfrac{\models R[m/x]}{\lceil m \rceil \langle x.R \rangle \longrightarrow \lceil m \rceil}$.

1. From $\cdot \vdash \lceil m \rceil \langle x.R \rangle : T$ (wlog, we omit inversion of a potential outermost application of the subsumption rule) we obtain the following two items
2. $\cdot \vdash \lceil m \rceil : S$
3. $\cdot \vdash x.R :: S \Longrightarrow T$
4. Inversion of Item 2 yields $\cdot \vdash x : B\{x = m\} \leq S$
5. Hence, $S = x : B\{P\}$ for some P such that $\forall x.x = m \supset P$
6. Inversion of Item 3 yields $T = x : B\{Q\}$ such that $\forall x : B.P \supset (Q \Leftrightarrow R)$.
7. Since $\models R[m/x]$, we obtain that $\forall x.x = m \supset P \wedge R$ and thus $\forall x.x = m \supset Q$ by Item 6.
8. Hence, $\cdot \vdash \lceil m \rceil : T = x : B\{Q\}$ by subsumption.

Case $\dfrac{W\langle k \rangle \longrightarrow W}{(V\langle (x : k) \to d \rangle)\, W \longrightarrow (V\, W)\langle d[W/x] \rangle}$ Coerce-Arg-Base.

1. From the premise, we obtain
2. $k = x.R$ and
3. $\models R[W/x]$.
4. Applying inversion to $\cdot \vdash (V\langle (x : k) \to d \rangle)\, W : T_0$ yields
5. $\cdot \vdash (V\langle (x : k) \to d \rangle) : (x : S') \to T'$
6. $\cdot \vdash W : S'$
7. $\cdot \vdash T_0$ where $T_0 = T'[W/x]$
8. Inversion of Item 5 yields
9. $\cdot \vdash (x : k) \to d :: (x : S) \to T \Longrightarrow (x : S') \to T'$
10. $\cdot \vdash V : (x : S) \to T$
11. From Item 9, we obtain by inversion
12. $\cdot \vdash k :: S' \Longrightarrow S$ and
13. $x : S \vdash d :: T \Longrightarrow T'$.
14. Hence, $S = x : B\{P\}$ and $S' = x : B\{P\}'$
15. Using Item 3, $\cdot \vdash W : S$
16. Hence $\cdot \vdash (V\, W) : T[W/x]$
17. By substitution on Item 13: $\cdot \vdash d[W/x] :: T[W/x] \Longrightarrow T'[W/x]$
18. Hence $\cdot \vdash (V\, W)\langle d[W/x] \rangle : T'[W/x]$

Case $\dfrac{W\langle k \rangle \ \mathsf{value}}{(V\langle (x : k) \to d \rangle)\, W \longrightarrow (V\, (W\langle k \rangle))\langle d \rangle}$ Coerce-Arg-Fun.

1. From the premise, we obtain that $k = (x' : k') \to d'$ a function coercion.
2. By inversion on $\cdot \vdash (V\langle (x : k) \to d \rangle)\, W : T_0$ we obtain

3. $\cdot \vdash (V\langle(x : k) \to d\rangle) : (x : S') \to T'$
4. $\cdot \vdash W : S'$
5. $\cdot \vdash T_0$ where $T_0 = T'[W/x]$
6. Further inversion on Item 3 yields
7. $\cdot \vdash V : (x : S) \to T$ and
8. $\cdot \vdash (x : k) \to d :: (x : S) \to T \Longrightarrow (x : S') \to T'$
9. By inversion $\cdot \vdash k :: S' \Longrightarrow S$ and $x : S \vdash d :: T \Longrightarrow T'$
10. By the premise, we know that $\lfloor S \rfloor = G \to G'$
11. Hence x does not occur free in T, T', and d
12. Hence $\cdot \vdash\ :: T \Longrightarrow T'$
13. Thus $\cdot \vdash W\langle k\rangle : S$ (using Item 9)
14. By Item 11, $\cdot \vdash T[W\langle k\rangle/x]$ because $T[W\langle k\rangle/x] = T$
15. Hence $\cdot \vdash V(W\langle k\rangle) : T$
16. We conclude $\cdot \vdash (V(W\langle k\rangle))\langle d\rangle : T'$ □

References

1. Aspinall, D., Compagnoni, A.B.: Subtyping dependent types. Theoret. Comput. Sci. **266**(1–2), 273–309 (2001)
2. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: The 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016 (2016)
3. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Peyton-Jones, S. (ed.) Proceedings ICFP 2002, pp. 48–59. ACM, New York (2002)
4. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: Proceedings of the 37th ACM Symposium POPL, pp. 353–364. ACM, Madrid, January 2010
5. Henglein, F.: Dynamic typing: syntax and proof theory. Sci. Comput. Program. **22**, 197–230 (1994)
6. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Programming (TFP) (2007)
7. Knowles, K.L., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. **32**(2), 1–34 (2010)
8. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: Lévy, J.J., Mayr, E.W., Mitchell, J.C. (eds.) IFIP TCS, pp. 437–450. Kluwer, Netherlands (2004)
9. Siek, J., Thiemann, P., Wadler, P.: Blame and coercion: together again for the first time. In: Blackburn, S. (ed.) PLDI, pp. 425–435. ACM, Portland (2015)
10. Sjöberg, V., Casinghino, C., Ahn, K.Y., Collins, N.,Eades III, H.D., Fu, P., Kimmell, G., Sheard, T., Stump, A., Weirich, S.: Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In: Chapman, J., Levy, P.B. (eds) Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP 2012, Tallinn, Estonia, 25 March 2012, vol. 76 ofEPTCS, pp. 112–162 (2012)
11. Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Danvy, O. (ed.) Proceedings ICFP, Tokyo, Japan, September 2011, pp. 266–278. ACM, New York (2011)
12. Torlak, E., Bodík, R.: Growing solver-aided languages with rosette. In: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (eds.) Onward! 2013, pp. 135–152. ACM, Indianapolis (2013)

13. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: O'Boyle, M.F.P., Pingali, K. (eds.) PLDI, p. 54. ACM, New York (2014)
14. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009)
15. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994)

Certifying Data in Multiparty Session Types

Bernardo Toninho and Nobuko Yoshida(✉)

Imperial College London, London, UK
{b.toninho,n.yoshida}@imperial.ac.uk

Abstract. Multiparty session types (MPST) are a typing discipline for ensuring the coordination and orchestration of multi-agent communication in concurrent and distributed programs. However, by mostly focusing on the communication aspects of concurrency, MPST are often unable to capture important data invariants in programs. In this work we propose to increase the expressiveness of MPST by considering a notion of value dependencies in order to certify invariants of exchanged data in concurrent and distributed settings.

1 Introduction

Theoretical principles can have transformational effects on computing practice. Well-known examples include program logics and the structured programming discipline. Many theoretical principles established by Philip Wadler have already produced a broad impact on current practices. Wadler's work was instrumental in the introduction of generic types to Java [10], which are now an established feature of statically typed languages such as Java, C#, and the .NET framework. He was a co-designer of Haskell, and features he designed have influenced a wide range of programming languages such as F# and Scala; and database languages such as Ferry and LINQ.

At the core of Wadler's long list of contributions is the notion of types as the fundamental tool for abstraction and reasoning about programs, and as a means of exposing a program's true meaning. In more recent work, Phil has devoted some his efforts to tackling the challenges of communication, concurrency and distributed computation. Naturally, and fruitfully, the answer presents itself in type form.

Meeting these challenges, our mobility group [13] is working with Wadler within the scope of our EPSRC project, *ABCD: A Basis for Concurrency and Distribution* [1]. We quote from *Ambition and Vision* which was (mainly) written by Wadler:

Ambition and Vision [1]***.*** *The data type is one of computing's most successful concepts. The notion of data type appears in programming languages from the oldest to the newest, and it covers concepts ranging from a single bit to organised tables containing petabytes of data. Types act as the fundamental unit of compositionality: the first thing a programmer writes or reads about each method or*

This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, and EP/L00058X/1; and by EU FetOpen UPSCALE 612985.

module is the data types it acts upon, and type discipline guarantees that each call of a method matches its definition and each import of a module matches its export. Data types play a central role in all aspects of software, from architectural design to interactive development environments to efficient compilation.

The ambition of our project is to position *session types* as the analogue of the data type for concurrency and distribution. Session types impose structure to sessions, in the same way data types impose structure to data instances. Session types were first devised two decades ago by Takeuchi, Honda, Kubo and Vasconcelos [8,17], and later developed by Wadler and others [3,4,12,20,21]. Session types build upon data types, as data types specify the lowest level of data exchange, upon which more complex protocols are built. Just as data type discipline matches use and definition of a method, and import and export of a module, a session type discipline ensures consistency and compatibility between the two ends of a communication. Session types offer a mechanism for ensuring communication safety (and a variety of other fundamental properties such as absence of deadlocks or races) of systems involving two interacting parties.

A more general view of a session is that it combines *multiple* interactions forming a meaningful scenario into a single logical unit, offering a basic programming abstraction for communicating processes. Given that in a wide range of application scenarios it is often necessary to specify and ensure the coordination of multiple communicating agents, Honda et al. [5,9] introduced *multiparty* session types (MPST), enabling the specification of interactions involving multiple peers from a global perspective, which is then automatically mapped (or *projected*) to *local* types that may be checked against the individual endpoint processes. Using this framework, communication safety is ensured among multiple endpoints.

This paper seeks to extend Wadler's viewpoint of a session type: we propose a session type discipline for expressing and certifying global properties that may depend on the exchanged data, by introducing value dependent types for multiparty sessions.

Our proposed typing discipline ensures that implementations of a multiparty conversation not only adhere to the session discipline but also satisfy rich constraints imposed on the exchanged data, which may be explicitly witnessed at runtime by *proof objects*. Our aim is to thus raise the standard of types in concurrency to that of data types: the programmer with a precise description of the interaction patterns followed by the communicating parties but also specify (and *certify*) the global invariants of data that are required and ensured throughout the multiparty communication.

2 Multiparty Session Types and Certified Data

This section motivates a technique to certify properties of exchanged data in multiparty session types (MPST) through a notion of value dependencies [19]. We begin with a brief introduction of the original MPST framework and its shortcomings with respect to expressing certain functional constraints on global protocols. We then address these issues through the use of value dependencies in the framework.

$$\begin{aligned} G \upharpoonright \mathsf{p} &= \mathsf{q}!(\mathsf{Int}); \mathsf{r}!(\mathsf{String}); \mathbf{end} \\ G \upharpoonright \mathsf{q} &= \mathsf{p}?(\mathsf{Int}); \oplus \mathsf{r}(\mathsf{yes} : G' \upharpoonright \mathsf{q}; \mathsf{no} : \mathbf{end}) \\ G \upharpoonright \mathsf{r} &= \mathsf{p}?(\mathsf{String}); \& \mathsf{q}(\mathsf{yes} : G' \upharpoonright \mathsf{r}; \mathsf{no} : \mathbf{end}) \end{aligned}$$

Fig. 1. Projections of global type G (with $\mathsf{p} \notin G'$)

In MPST, we begin with a *global type*, consisting of a global view of the interactions shared amongst the several interested parties. For instance, the following consists of a global type specification of a toy protocol involving three parties:

$$G = \mathsf{p} \to \mathsf{q} : (\mathsf{Int}).\mathsf{p} \to \mathsf{r} : (\mathsf{String}).\mathsf{q} \to \mathsf{r} : \big(\mathsf{yes}{:}G'; \mathsf{no}{:}\mathbf{end}\big) \tag{1}$$

In the specification G above, participant p sends an integer and a string to participants q and r, respectively. Afterwards, q will either send to r a no message, ending the global interaction; or a yes message, causing the interaction to proceed to G'. We may assume that G specifies a portion of some coordinated agreement, such as that between a service broker p, giving a price quote to a client q while making a tentative reservation of the service to provider s (encoded as a string), which is then accepted or rejected by q.

Given a global type, we must define a notion of *projection*, which constructs the view for each endpoint of the global interaction as a *local type*. For G, the projection of G for p, written $G \upharpoonright \mathsf{p}$, is given by the local type $\mathsf{q}!(\mathsf{Int}); \mathsf{r}!(\mathsf{String}); \mathbf{end}$, assuming p does not participate in G', which describes the parts of the global interaction that pertain to actions of participant p. Given the projected local types (Fig. 1) for each communicating party we may then check that the global specification is satisfied by the interactions of the several endpoint processes to ensure deadlock-freedom.

The framework sketched above is only suited for describing the shape of communication. While we may argue that G does indeed specify the interactions between a service broker p, a client q, and a provider r, such a global specification is satisfied by many process instances of p, q and r that may not in fact offer the desired functionality. For instance, the implementation of the broker p may send an incorrect price to client q, or the wrong service identifier to provider r and the system would still be correct according to G. The crucial issue is that while a global type specifies precisely *how* parties communicate, it only captures *what* the parties should communicate in a very loose sense (for example, "send a string" vs. "send a string corresponding to the service code for which the client was sent the price").

To overcome this issue, we propose the adoption of *value dependent* multiparty session types, which refine multiparty session types by adding type dependencies to specifications of exchanged data (extending the work of [15,19] for the binary setting).

The technical challenge here is reconciling the global specification of the distributed interaction, which may reference properties depending on data spread across multiple endpoints, with the local knowledge of each participant. Projection must ensure that whenever two endpoint processes exchange a proof object, the object is consistent with the knowledge of both endpoints. Specifically, the sender must know each term referenced by the proof object and propagate the relevant information to the receiver in a consistent way. Another issue is that the local types generated for each endpoint may not necessarily have matching dependencies due to the potentially incomplete views of the global agreement (for instance, participant p may assert some relationship between two data elements to q, where q only knows one of the datum). We must nevertheless ensure that endpoint projections are well-formed given the local knowledge of each endpoint and preserve the intended data dependencies, given the partial view of the system.

2.1 Value Dependent Multiparty Session Types

The key motivation for using value dependent types is to enable type level specifications of properties of data used in computation. Given that MPST have a natural distributed interpretation, we wish not only to express properties of exchanged data but also to support the ability for processes to exchange *proof objects* witnessing the properties of interest, providing a degree of certified communication in some sense. For instance, a value dependent version of G above can be:

$$\begin{aligned} G_{Dep} = {} & \mathsf{p} \to \mathsf{q} : (x{:}\mathsf{Int}).\mathsf{p} \to \mathsf{r} : (y{:}\mathsf{String}). \\ & \mathsf{p} \to \mathsf{q} : (z{:}\mathsf{isPrice}(x,y)).\mathsf{q} \to \mathsf{r} : \big(\mathsf{yes}{:}G'; \mathsf{no}{:}\mathbf{end}\big) \end{aligned} \tag{2}$$

where the predicate $\mathsf{isPrice}(x, y)$ holds only if the integer x is indeed the price for service y. In the specification G_{Dep}, p must also send to q a *proof* of the relationship between the previously sent price and the service code. While the notion of proofs as first-class objects might seem somewhat foreign insofar as one might simply expect some runtime verification mechanism that ensures the received data is in the required form (as specified by the type-level *assertions*), explicit proof exchange is a more general approach: proof generation might not be decidable in general, whereas proof checking should be. Moreover, even when proof generation is decidable, it can often require more computational resources than checking the validity of a proof object.

By leveraging the Curry-Howard correspondence between propositions and types (and proofs and programs), we can represent such proof objects as terms in a language with a suitable (dependent) type discipline, such that the only well-typed instances of processes implementing the role of p will be those that not only adhere to the session discipline but also satisfy the functional constraints encoded in the dependently typed values. The framework also ensures that proof objects are explicitly exchanged between communicating parties, which is of practical significance in a distributed setting.

Global Types. The syntax for value dependent MPST is given in Fig. 2. A message exchange $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G$ specifies communication between sender p and receiver q of a value of type τ, bound to x in G. The type structure of τ is somewhat generic, with the following requirements: we assume a dependently typed λ-calculus with dependent functions $\Pi x{:}\tau.\sigma$ and pairs $\Sigma x{:}\tau.\sigma$, where x binds its occurrence in Π and Σ. In our theory and examples, we generalise dependent pair types $\Sigma x{:}\tau.\sigma$ to $\Sigma l.\sigma$, where l is a *list* of type bindings of the form $x_i{:}\tau_i$.

$$\begin{array}{rcl} G & ::= & \mathsf{p} \to \mathsf{q} : (x{:}\tau).G \\ & | & \mathsf{p} \to \mathsf{q} : \left(\mathsf{l}_j{:}G_j\right)_{j\in J} \\ & | & \mathsf{p} \to \mathsf{q} : (T).G \\ & | & \mu t\ (x = M{:}\tau).G \mid t\langle M\rangle \\ & | & \mathbf{end} \end{array} \qquad \begin{array}{rcll} T, U & ::= & \mathsf{p}!(x{:}\tau);T & \mid \mathsf{p}?(x{:}\tau);T \\ & | & \oplus\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I} & \mid \&\mathsf{p}\left(\mathsf{l}_i{:}T_i\right)_{i\in I} \\ & | & \mathsf{p}!(U);T & \mid \mathsf{p}?(U);T \\ & | & \mu t\ (x = M{:}\tau).T \mid t\langle M\rangle & \\ & | & \mathbf{end} & \end{array}$$

$$\tau, \sigma ::= \Pi x{:}\tau.\sigma \mid \Sigma x{:}\tau.\sigma \mid b \mid \mathcal{S}(M)$$

Fig. 2. Syntax of global and local types

We manipulate such lists using Haskell-style notation. We assume some base types b and singleton types [16], written $\mathcal{S}(M)$, where M is a value of some base type b and $\mathcal{S}(M)$ denotes a value of type b equal to M. For example, if we assume natural numbers Nat as base types then the natural number 5 can be typed with both Nat and $\mathsf{Nat}(5)$. We require type preservation and progress for this language of message values, as well as decidability of type-checking (although, crucially, not of type inhabitance).

The branching $\mathsf{p} \to \mathsf{q} : \left(l_j{:}G_j\right)_{j\in J}$ denotes a selection made by p between a set of behaviours G_j identified by labels l_j, achieved by the emission of a label l_i with $\{\mathsf{l}_i : i \in J\}$ from p to q. The session then continues as G_i for all participants.

Session delegation $\mathsf{p} \to \mathsf{q} : (T).G$ denotes that participant p delegates to q its interactions with a session channel of local type T (defined below), achieved by sending the channel endpoint, after which the interaction proceeds as G. Note that there is no binding for T since we only consider dependencies of *values*, rather than on sessions.

Recursive global types $\mu t\ (x = M{:}\tau).G$, where t and x bind its occurrences in G, enable the specification of how a recursive interaction should proceed among the different participants. The parameter x is a recursion variable standing for a term M of type τ, which defines the initial value of x in the first recursive instance, acting as a parameter of the recursion. A recursion is instantiated with $t\langle M\rangle$, where M denotes the value taken by x in the next instance. We assume that recursive type definitions are contractive and, for the sake of simplicity, that there is at least one occurrence of t in G. We consider recursive types in the typical equirecursive sense, up to unfolding. Finally, **end** denotes a lack of further interactions. We often omit **end** and write message exchange and recursion as $\mathsf{p} \to \mathsf{q} : (\tau).G$ and $\mu t.G$ if x does not occur in G.

We write $fv(G)$ for the free variables of G, defined inductively in the usual way. We state that a global type G is *closed* (resp. *open*) if $fv(G) = \emptyset$ (resp. $fv(G) \neq \emptyset$). We write $\mathbb{C}[\![-]\!]$ for a global type context (i.e., a global type with a hole). $G \sqsubseteq G'$ stands that G is a subterm of G'.

Definition 2.1 (Global Type Context). *Given a global type, we define its subterms via the following notion of context:*

$$\begin{array}{rl} \mathbb{C} :: = & \_ \mid \mathsf{p} \to \mathsf{r} : (x{:}\tau).\mathbb{C} \mid \mathsf{p} \to \mathsf{r} : (T).\mathbb{C} \mid \mathsf{p} \to \mathsf{r} : \left(l_i : \mathbb{C}_i\right)_{i\in I} \mid \mu t(x = M{:}\tau).\mathbb{C} \\ | & \mathbf{end} \mid t\langle M\rangle \end{array}$$

with the hole $\_$ *occurring in at most one* $\mathbb{C}_i$.

$$
\begin{aligned}
G_{BSD} \triangleq\ & \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{query} : \mathsf{Nat}).\\
& \mathsf{Distr} \to \mathsf{Seller} : (\mathsf{stock} : \mathsf{Int}).\\
& \mathsf{Seller} \to \mathsf{Buyer} : (\mathsf{q} : (\mathsf{Int}(\mathsf{stock}), \mathsf{Double})).\\
& \mathsf{Buyer} \to \mathsf{Seller} : (\mathsf{ok} : \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{ok} : G_{ok}),\\
& \qquad\qquad\qquad\quad \mathsf{quit} : \mathsf{Buyer} \to \mathsf{Distr} : (\mathsf{quit} : \mathbf{end})\\
G_{ok} \triangleq\ & \mu t(x = \langle \pi_2(\mathsf{q}), \mathsf{inl\ refl} \rangle : \Sigma y : \mathsf{Double}(\pi_2(\mathsf{q})).y \geq \pi_2(\mathsf{q})).\\
& \mathsf{Buyer} \to \mathsf{Seller} : (\mathsf{offer} : \Sigma z : \mathsf{Double}.z \geq \pi_2(\mathsf{q})).\\
& \mathsf{Seller} \to \mathsf{Buyer} : (\mathsf{hag} : G_{hag}, \mathsf{exit} : \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{cancel} : \mathbf{end}),\\
& \qquad\qquad\qquad\quad \mathsf{sell} : \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{commit} : \mathbf{end}))\\
G_{hag} \triangleq\ & \mathsf{Seller} \to \mathsf{Distr} : (\mathsf{hag} : t\langle \pi_1(\mathsf{offer}), \pi_2(\mathsf{offer})\rangle)
\end{aligned}
$$

Fig. 3. Buyer - seller - distributor global type

Local (Endpoint) Types. Value dependent *local types* specify the behaviour and data constraints of each endpoint involved in the multiparty session. The send types $\mathsf{p}!(x{:}\tau);T$ and $\mathsf{p}!(U);T$ denote, respectively, sending a value M of type τ to participant p and proceeding with the behaviour $T\{M/x\}$ or sending a channel of type U and continuing with behaviour T. The selection type $\oplus\mathsf{p}\big(\mathsf{l}_i{:}T_i\big)_{i\in I}$ encodes the transmission to p of a label l_i following by the communication specified in T_i. Receive types $(\mathsf{p}?(x{:}\tau);T$ and $\mathsf{p}?(U);T)$ and branch types $\oplus\mathsf{p}\big(\mathsf{l}_i{:}T_i\big)_{i\in I}$ specify the dual behaviours of sending and selection. Recursive types $\mu t\ (x = M{:}\tau).T$ (and their instantiations $t\langle M\rangle$) specify a recursive behaviour T parameterised by a term M of type τ, bound to x.

2.2 Examples of Value Dependent Global Types

We now introduce two examples of value dependent global types, showcasing their heightened expressiveness.

Three Party Interaction: Buyer - Seller - Distributor. We specify the interaction patterns between three parties: a buyer, a seller and a distributor, illustrating the combined use of recursion and dependencies (Fig. 3).

The session begins with the buyer requesting a query of a product from the distributor. The distributor then communicates with the seller, sending the number of items currently available. The seller sends to the buyer the number of available product and an initial price. The buyer and the seller then initiate in a recursive negotiation, where the buyer selects to either proceed with the negotiation or to quit the protocol. In the latter case, the seller notifies the distributor of the cancellation. In the former, the buyer sends the seller an offer, upon which the seller must decide whether to continue negotiating, to terminate the negotiation by rejecting the offer, or to terminate the negotiation by accepting the offer. The decision is then forwarded to the distributor.

This interaction, beyond the equality constraints between the stock message sent from the distributor to the seller and then from the seller to the buyer, captures in a relatively simple way the encoding of the loop invariant – that each offer made by the buyer is always increasing, and at least as much as the initial quote.

MapReduce. We specify a distributed computation where a client sends to a server some data upon which the server is intended to run some potentially computationally expensive computation, represented by a map-style function f and a reduce-style function g.

$$\begin{aligned} G_{MR} \triangleq\ & \mathsf{Client} \to \mathsf{Server} : (\mathsf{d} : \mathsf{String}).\\ & \mathsf{Server} \to \mathsf{Worker}_1 : (\mathsf{d}_1 : \mathsf{String}).\mathsf{Server} \to \mathsf{Worker}_2 : (\mathsf{d}_2 : \mathsf{String}).\\ & \mathsf{Server} \to \mathsf{Aggr} : (\mathsf{p} : \mathsf{d} = \mathsf{d}_1 +\!\!+ \mathsf{d}_2).\\ & \mathsf{Worker}_1 \to \mathsf{Aggr} : (\mathsf{r}_1 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1)).\\ & \mathsf{Worker}_2 \to \mathsf{Aggr} : (\mathsf{r}_2 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2)).\\ & \mathsf{Aggr} \to \mathsf{Server} : (\mathsf{r}_3 : \Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)))\\ & \mathsf{Server} \to \mathsf{Client} : (\mathsf{res} : \mathsf{String}(\pi_1(\mathsf{r}_3))) \end{aligned}$$

Upon receiving the data from the client, the server divides it into two parts which are then sent to be processed by the two workers. The system includes an aggregator service, which is informed by the server of the division of the data. The workers then send to the aggregator the result of the computation f on their respective data partitions, which then sends back to the server the aggregation result (computed using the aggregation function g). Finally, the server sends back to the client the final result.

The crucial aspect of this simple example is that not only are we describing the structure of communication (and to some extent, the topology of the service), we are specifying in a very precise way the actual functionality of the global coordination.

Recursive Game. To clarify the interaction of recursion and value dependencies, we encode a simple toy game protocol between three parties: Alice, Bob and Carol.

$$\begin{aligned} G_{ABC} \triangleq\ & \mathsf{Carol} \to \mathsf{Alice} : (n : \Sigma y{:}\mathsf{Nat}.y > 0).\\ & \mathsf{Carol} \to \mathsf{Bob} : (n' : \mathsf{Nat}(\pi_1(n))).\\ & \mu t(x = n : \Sigma y{:}\mathsf{Nat}.y > 0).\\ & \mathsf{Alice} \to \mathsf{Carol} : (m : \mathsf{Nat}).\\ & \mathsf{Bob} \to \mathsf{Carol} : (m' : \mathsf{Nat}).\\ & \mathsf{Carol} \to \mathsf{Alice} : (\mathsf{correct} : G_{c1}, \mathsf{wrong} : t\langle x-1, M\rangle)\\ G_{c1} \quad \triangleq\ & \mathsf{Carol} \to \mathsf{Bob}(\mathsf{correct} : \mathsf{end}, \mathsf{wrong} : t\langle x-1, M\rangle) \end{aligned}$$

In the protocol above, Carol sends both Alice and Bob a number of total tries n the two participants are allowed to attempt to guess some random number generated by Carol. The protocol then proceeds by repeatedly accepting guesses from both Alice and Bob until they both guess correctly, upon which the protocol terminates, or until the number of tries n runs out.

While very minimal in its features, this example showcases how the combination of recursion and value dependencies allows us to specify sophisticated global types, such as counting down from a sent or received number, insofar as we are able to make the actual communication structure of the protocol depend on previously received data.

2.3 Well-Formedness of Global Types

We detail the well-formedness conditions on global value-dependent MPST. In contrast with the work on design-by-contract [2], which introduces assertions to MPST, we do not in general enforce the property that all well-formed global types are realisable by some well-typed endpoint processes. In [2], global type well-formedness entails that assertions expressed in a global type are possible to satisfy, by restricting the assertion language to decidable logics. Given our aim of maintaining a general dependent type theory as our proof language, we opt for a different design.

In our general setting, we can use a larger set of well-formed global types for which no process realisers may exist. The decision problem of determining if such process realisers exist is itself undecidable. Our goal is to define well-formedness of global and local types such that:

1. Projection of a well-formed global type produces well-formed local types by a simple projection rule; and
2. If a collection of processes which satisfy local types exist, then the global specification is satisfied.

Below we define simple *well-formedness conditions* which are sufficient to ensure the above properties. The first condition defines a binding restriction on recursions; the second captures the fact that in a message exchange between two participants, the sender should always know all the message variables mentioned in the message's type. We note that history-sensitivity has been shown decidable in [2], where a compositional proof system for history sensitivity is presented.

Definition 2.2 (Well-Formedness Conditions).

1. (*recursion*) Let G be a closed global type. We say that G has well-formed recursion iff for all $t\langle M\rangle \in G, x \in fv(M)$, there exists $\mathbb{C}[\![-]\!]$ such that either: $G = \mathbb{C}[\![\mathsf{p} \to \mathsf{q} : (x{:}\sigma).G']\!]$ or $G = \mathbb{C}[\![\mu t\ (x = M : \tau).G']\!]$.
2. (*history sensitivity*) Given a global type G, we say that p *ensures* τ in G iff there is $\mathbb{C}$ such that $G = \mathbb{C}[\![\mathsf{p} \to \mathsf{q} : (x{:}\tau).G']\!]$. Then for any natural number n, G is n-history sensitive on a message variable x iff for all G' such that G' is a n-times unfolding of G, and for all types τ in G' such that $x \in fv(\tau)$ there is $\mathsf{p} \to \mathsf{q} : (x{:}\tau').G'' \sqsubseteq G'$ such that p or q ensures τ in G''. We say that G is *history sensitive* iff it is n-history sensitive for all natural numbers n on all message variables in G.

G is well-formed if all recursions in G are well-formed and G is history sensitive.

Hereafter we consider only well-formed global types.

3 Projection and Data Dependencies

We now motivate some of the challenges of defining projection of a global type while respecting the partial local knowledge of each participant.

Recall the global type G_{Dep} of Sect. 2.1 (Eq. 2), which is a well-formed global type. In the final interaction between participants p and q, p is supposed to send a proof that the previously received integer value x is indeed the price for the service code sent to r, identified by the string y. From the perspective of p, the value of both x and y are known. However, q only knows the value of x since y was sent only to r. Thus, if we consider a typical notion of projection that traverses the type G_{Dep} and collects the direction of communication accordingly we obtain the following local types for p and q (for some T_{q}):

$$\begin{aligned} G_{Dep}\upharpoonright \mathsf{p} &= \mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\mathsf{isPrice}(x,y)); \mathbf{end} \\ G_{Dep}\upharpoonright \mathsf{q} &= \mathsf{p}?(x{:}\mathsf{Int}); \mathsf{p}?(z{:}\mathsf{isPrice}(x,y)); T_{\mathsf{q}} \end{aligned}$$

The local type for q cannot be correct since it contains a free variable y (given q's local knowledge), which is not free in the local type for p. In order to generate adequate local types for both endpoints we must ensure that the two types respect the local knowledge of each participant.

Intuitively, the type for the endpoint corresponding to participant p must bundle in the message identified by z all the unknown information from participant q's perspective. However, if we modify the projection for participant p to,

$$\mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\varSigma y'{:}\mathsf{String}.\mathsf{isPrice}(x,y')); \mathbf{end} \tag{3}$$

we do not preserve the semantics of G_{Dep}, in the sense that a process with the type above may send to q *any* price, provided it is indeed the price of a service in the system.

In order to preserve both the semantics of data dependencies in global types *and* generate well-formed local types for both endpoints, we make use of singleton types and subtyping, which is formally defined in Sect. 3.1. Crucially, we make use of singleton types to implicitly refer to the equality constraints induced by dependencies in a global type. In the example above, generating the following local type for p,

$$\mathsf{q}!(x{:}\mathsf{Int}); \mathsf{r}!(y{:}\mathsf{String}); \mathsf{q}!(z{:}\varSigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x,y')); \mathbf{end} \tag{4}$$

we can preserve the semantics of G_{Dep}, in the sense that p may only send x and y such that one is the price of the other. Moreover, we exploit the fact that for any base type b, if $M : b$ then $\mathcal{S}(M) \leq b$ in order to produce the following local type for endpoint q,

$$\mathsf{p}?(x{:}\mathsf{Int}); \mathsf{p}?(z{:}\varSigma y'{:}\mathsf{String}.\mathsf{isPrice}(x,y')); T_{\mathsf{q}} \tag{5}$$

The type above not only respects the local knowledge of endpoint q but is also compatible with the interactions specified by the local type for endpoint p due to the subtyping of singletons, since $\varSigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x,y') \leq \varSigma y'{:}\mathsf{String}.\mathsf{isPrice}(x,y')$ by the usual covariant subtyping rules for $\varSigma$-types and the fact that a singleton is always a subtype of its corresponding base type (we note that session subtyping for message input is covariant in the message type; and dually, contravariant for output).

$$\frac{\Psi \vdash M : b}{\Psi \vdash \mathcal{S}(M) \leq b}\ (\text{SUB-}\mathcal{S}) \qquad \frac{\Psi \vdash M_1 \equiv M_2 : b}{\Psi \vdash \mathcal{S}(M_1) \leq \mathcal{S}(M_2)}\ (\text{SUBEQ-}\mathcal{S})$$

$$\frac{\begin{array}{c}\Psi \vdash \Pi x{:}\tau_1'.\tau_1'' \\ \Psi \vdash \tau_2' \leq \tau_1' \qquad \Psi, x{:}\tau_2' \vdash \tau_1'' \leq \tau_2''\end{array}}{\Psi \vdash \Pi x{:}\tau_1'.\tau_1'' \leq \Pi x{:}\tau_2'.\tau_2''}\ (\text{SUB-}\Pi) \qquad \frac{\begin{array}{c}\Psi \vdash \Sigma x{:}\tau_1'.\tau_1'' \\ \Psi \vdash \tau_1' \leq \tau_2' \qquad \Psi, x{:}\tau_1' \vdash \tau_1'' \leq \tau_2''\end{array}}{\Psi \vdash \Sigma x{:}\tau_1'.\tau_1'' \leq \Sigma x{:}\tau_2'.\tau_2''}\ (\text{SUB-}\Sigma)$$

$$\frac{\Psi \vdash M \equiv N : b}{\Psi \vdash \mathcal{S}(M) \equiv \mathcal{S}(N)}\ (\text{TEQ-}\mathcal{S}) \qquad \frac{\Psi \vdash M \equiv N : \sigma \qquad \Psi \vdash \sigma \leq \tau}{\Psi \vdash M \equiv N : \tau}\ (\text{EQ-}\leq)$$

$$\frac{\Psi \vdash \tau \leq \tau' \qquad \Psi, x{:}\tau' \vdash T \leq T'}{\Psi \vdash \mathsf{p}?(x{:}\tau).T \leq \mathsf{p}?(x{:}\tau').T'}\ (\text{SUB-}?) \qquad \frac{\Psi \vdash \tau' \leq \tau \qquad \Psi, x{:}\tau \vdash T \leq T'}{\Psi \vdash \mathsf{p}!(x{:}\tau).T \leq \mathsf{p}!(x{:}\tau').T'}\ (\text{SUB-}!)$$

Fig. 4. Subtyping for local and data types (Abridged)

3.1 Defining Projection

Having discussed the main challenges of preserving global data dependencies in local types, we define a notion of projection that generates *compatible* message types (in the sense of Definition 3.2) for well-formed global types.

We begin by introducing the subtyping rules for both local and data types. The rules are mostly standard from the literature of subtyping in session types [7] and singleton types [16]. For conciseness we only consider session subtyping for input and output types. Subtyping for choices and branching are orthogonal. The subtyping judgement, written $\Psi \vdash \tau \leq \sigma$ for data types and $\Psi \vdash T \leq S$ for local types, denotes that τ (resp. T) is a subtype of σ (resp. S), where Ψ is a context tracking free variables in types. Note that if T is a subtype of U, then a process implementing type T may be safely used wherever one of type U is expected. We write $\Psi \vdash M : \tau$ for the typing judgement of terms M, which we maintain mostly unspecified. We write $\Psi \vdash \tau$ for the well-formedness of τ and $\Psi \vdash M \equiv N : \tau$ for definitional equality of M and N. The key subtyping rules are given in Fig. 4.

The key rules for the development of a well-defined notion of projection is the singleton subtyping rule (SUB-$\mathcal{S}$), which specifies that a singleton for a base type is always a subtype of its base type and the rules for subtyping of input and output local types, enabling receiving processes to receive instances of the singleton type when expecting to receive instances of the corresponding base types.

We make precise the notion of a participant knowing the identity of a message or recursion variable occurring in a global type. Intuitively, a participant knows the identity of a message variable if it is involved in corresponding communication. Similarly, knowing a recursion variable requires knowledge of all message variables that occur in the recursive parameter.

Definition 3.1 (Knowledge). Let G be a closed global type and $\mathsf{p} \in G$. We say that p knows $x{:}\tau$ in G iff there is $\mathbb{C}$ such that either:

- $G = \mathbb{C}[\![\mathsf{s} \to \mathsf{r} : (x : \tau).G']\!]$ with $\mathsf{p} \in \{\mathsf{s}, \mathsf{r}\}$; or
- $G = \mathbb{C}[\![\mu t(x = M : \tau).G']\!]$ where for all $y \in \mathit{fv}(M) \cup \bigcup_{t\langle M'\rangle \in G'} \mathit{fv}(M') \setminus \{x\}$ and p knows y in G.

We say that a participant p knows M in G iff p knows all the free variables of M in G.

Equipped with our notion of subtyping and knowledge, we define compatibility between message types in Definition 3.2, appealing to a consistent *priming* of the variables in a type. Given a variable $x{:}\tau$ with $x \in \mathit{fv}(\sigma)$, we say that x' is a primed version of x iff $x'{:}\tau(x)$. A priming of type σ is a pointwise priming of (some) of its free variables. We maintain the connection between a primed variable and its unprimed version (we write $primedVars(r)$ to denote the primed variables of set r).

Definition 3.2 (Compatible Message Types). *Given a well-formed global type G with $\mathsf{p} \to \mathsf{q} : (x : \tau).G' \sqsubseteq G$ we say that the pair of data types (σ_1, σ_2) is compatible with p and q for message x iff*

1. *$\Psi \vdash \sigma_1$ and $\Psi \vdash \sigma_2$, for some Ψ;*
2. *p (resp. q) knows all the (free) variables in σ_1 (resp. σ_2);*
3. *$\Psi \vdash \sigma_1 \leq \sigma_2$ for some Ψ;*
4. *$\Psi \vdash \sigma_1 \equiv \Sigma l.\tau'$, for some priming of τ and some (possibly empty) list l.*

Two message types σ_1 and σ_2 are deemed compatible from the perspective of participants p and q if both types are well-formed, their free variables are known by the corresponding participants and they are related by subtyping. Moreover, we enforce that compatible message types must be dependent tuples (without loss of generality). For example, in the global type G_{Dep} discussed above, for the last message exchange between participants p and q, the pair of message types $\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y')$ and $\Sigma y'{:}\mathsf{String}.\mathsf{isPrice}(x, y')$ is compatible for p and q, respectively.

We make use of an auxiliary function, dubbed *compatible type binding generation* (CTB), that given a message exchange $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G'$ in a global type G produces a dependent tuple $\Sigma l.\tau'$, where τ' is a *priming* of τ (τ and τ' differ only on the names of free variables of base type, where $x' \in \mathit{fv}(\tau')$ corresponds to $x \in \mathit{fv}(\tau)$) and l is a list of variable bindings (occurring in τ') that are known by participant p and not known by q, making use of singleton types to preserve the value dependencies specified in the global type G.

Definition 3.3 (Compatible Type Binding Generation). *For any closed global type G, with $\mathsf{p} \to \mathsf{q} : (x : \tau).G' \sqsubseteq G$. We generate a compatible type binding for $x{:}\tau$, written $CTB(x{:}\tau)$, as follows. If τ is a base type then $CTB(x{:}\tau) = [x{:}\tau]$. Otherwise, the compatible type binding for $x{:}\tau$ is given by the recursive function $F(x{:}\tau)$, given below making use of typical list manipulation notation:*

1. *If τ is a base type, then $F(x{:}\tau) = [x'{:}\tau(x)]$; otherwise,*
2. *Let u be the list of bindings corresponding to the free variables of τ, known by p and not by q.*
3. *If $u = [\,]$ then $F(x{:}\tau) = [x{:}\tau]$; otherwise,*
4. *Let $r = \mathsf{fold}\ (\lambda b.\lambda acc\ .\mathsf{merge}(F(b), acc))\ [\,]\ u_1$.*
5. *Let $\tau' = \tau\{primedVars(r)'/primedVars(r)\}$, then $F(x{:}\tau) = r \mathbin{++} [x{:}\tau']$.*

We note that in recursive calls to F, the participants p *and* q *are fixed in the sense that $F(b)$ considers variables in the binding b known by* p *and unknown by* q. *Moreover, usages of CTB tacitly assume that we convert the resulting list into a dependent tuple in the natural dependency-preserving way.*

For instance, in the global type G_{Dep} (Eq. 2), the CTB for the third exchange between p and q produces the type $\Sigma y'{:}\mathsf{String}(y).\mathsf{isPrice}(x, y')$ which may be used as the message type for the output of p, bundling all the necessary data that is unknown by q at the given point in the protocol. The key insight is that CTB consists of a terminating function that computes a tuple bundling all the unknown information from the perspective of the recipient of a message, using singleton types to preserve data dependencies from the perspective of the sender.

Theorem 3.4 (Compatible Type Binding Generation – Termination). *Compatible type binding generation (Definition 3.3) is a terminating function.*

Proof. We first point out that the fact that we consider *closed* global types enforces some constraints on free variables in data types. In particular, a data type's free variables must have all been defined by *previous* communication actions, which immediately excludes circular dependencies where two data types mutually depend on each other (e.g. the binding for a free variable y of a type $x{:}\mathcal{P}(y)$ being of the form $y{:}\mathcal{Q}(x)$).

Function $CTB(x{:}\tau)$ in Definition 3.3 inspects bindings of free variables of τ, known by participant p and unknown by q (c.f. Definition 3.1). By construction, these variables are bound by previous interactions involving p. Since there is no possibility for circularity, the free variables of a data type and the free variables of types in their binding occurrences form a *directed acyclic graph* (DAG).

The termination of F follows from the observation that it simply performs a traversal of this DAG, producing a reverse topological ordering of the graph. Specifically, F traverses the subgraph of this DAG made up of variables known by p and not by q (itself a DAG). This is straightforward to see: the terminal nodes of the graph are those when we reach a base type or have no unknown variables; for non-terminal nodes, that is, those with unknown variables, we perform a depth-first search traversal of the DAG, collecting the outcomes in a merged list with the appropriately primed type as the last element. We note that this traversal produces a reverse topological ordering of the DAG. $\square$

To ensure that projection produces well-formed local types for both endpoints, we make use of *singleton erasure* (Definition 3.5) to erase singletons from a dependent tuple. Intuitively, we use CTB to generate the message type for the sender and its singleton erasure to generate the type for the recipient, observing that the two are related by subtyping.

Definition 3.5 (Singleton Erasure). Given a type τ of the form $\Sigma l.\sigma$, we write $\tau^\dagger$ for its singleton erased version, that is, where each primed binding in l of the form $x_i'{:}\sigma_i(x)$ is replaced by $x_i'{:}\sigma_i$.

Finally, for projection of choices and branchings we appeal to a merge operator along the lines of [6], written $T \sqcup T'$, ensuring that if the locally observable

$$\mathsf{s} \to \mathsf{r} : (x{:}\tau).G' \upharpoonright \mathsf{p} = \begin{cases} \mathsf{r}!(x{:}(CTB(x{:}\tau))); (G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{s} \\ \mathsf{s}?(x{:}(CTB(x{:}\tau))^{\dagger}); (G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} = \mathsf{r} \\ G' \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$\mathsf{s} \to \mathsf{r} : \Big(l_j{:}G_j\Big)_{j\in J} \upharpoonright \mathsf{p} = \begin{cases} \oplus\mathsf{r}\big(l_j : G_j \upharpoonright \mathsf{p}\big)_{j\in J} & \text{if } \mathsf{p} = \mathsf{s} \\ \&\mathsf{s}\big(l_j : G_j \upharpoonright \mathsf{p}\big)_{j\in J} & \text{if } \mathsf{p} = \mathsf{r} \\ \sqcup_{j\in J} G_i \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$\mu t(x = M{:}\tau).G' \upharpoonright \mathsf{p} = \begin{cases} \mu t(x = M{:}\tau).(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \in G' \text{ and } \mathsf{p} \text{ knows } M \\ \mu t.(G' \upharpoonright \mathsf{p}) & \text{if } \mathsf{p} \in G' \text{ and } M \text{ unknown to } \mathsf{p} \\ \mathbf{end} & \text{otherwise} \end{cases}$$

$$t\langle M\rangle \upharpoonright \mathsf{p} = \begin{cases} t\langle M\rangle & \text{if } \mathsf{p} \text{ knows } M \\ t & \text{otherwise} \end{cases}$$

$$\mathbf{end} \upharpoonright \mathsf{p} = \mathbf{end}$$

Fig. 5. Projection.

behaviour of the local type is not independent of the chosen branch then it is identifiable via a unique choice/branching label (the merge operator is otherwise undefined).

Definition 3.6 (Merge). *Let* $T = \&\mathsf{r}\big(\mathsf{l}_i : T_i\big)_{i\in I}$ *and* $T' = \&\mathsf{r}\big(\mathsf{l}'_j : T'_j\big)_{j\in J}$. *The merge* $T \sqcup T'$ *of* T *and* T' *is defined as:*

$$\begin{aligned} T \sqcup T' &\triangleq \&\mathsf{r}\big(\mathsf{l}_h : T_h\big)_{h\in I\setminus J} \cup \big(\mathsf{l}'_h : T'_h\big)_{h\in J\setminus I} \cup \big(\mathsf{l}_h : T_h \sqcup T'_h\big)_{h\in I\cap J} \\ T \sqcup T &\triangleq T \end{aligned}$$

if $\mathsf{l}_h = \mathsf{l}'_h$ *for each* $h \in I \cap J$. *Merge is homomorphic (i.e.* $\mathcal{C}[T_1] \sqcup \mathcal{C}[T_2] = \mathcal{C}[T_1 \sqcup T_2]$*) and is undefined otherwise.*

Definition 3.7 (Global Projection). Let G by a global type. The projection of G in a participant p is defined by the function $G \upharpoonright \mathsf{p}$ in Fig. 5. If no side conditions hold then projection is undefined.

Example 3.8 (MapReduce). A projection of MapReduce in Sect. 2.2 from the viewpoint of participant Server is given below (projections for the other roles are given in Fig. 6):

$$\begin{aligned} G_{MR} \upharpoonright \mathsf{Server} \triangleq\ & \mathsf{Client}?(\mathsf{d}{:}\mathsf{String}); \mathsf{Worker}_1!(\mathsf{d}_1{:}\mathsf{String}); \mathsf{Worker}_2!(\mathsf{d}_2{:}\mathsf{String}); \\ & \mathsf{Aggr}!(\mathsf{p}{:}\Sigma\mathsf{d}'{:}\mathsf{String}(\mathsf{d}), \mathsf{d}'_1{:}\mathsf{String}(\mathsf{d}_1), \mathsf{d}'_2{:}\mathsf{String}(\mathsf{d}_2).\mathsf{d}' = \mathsf{d}'_1 \mathbin{++} \mathsf{d}'_2); \\ & \mathsf{Aggr}?(\mathsf{r}_3{:}\Sigma\mathsf{r}_1{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1), \\ & \qquad \mathsf{r}_2{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2), \\ & \qquad \mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2))); \\ & \mathsf{Client}!(\mathsf{res}{:}\Sigma\mathsf{d}'_1{:}\mathsf{String}(\mathsf{d}_1), \mathsf{d}'_2{:}\mathsf{String}(\mathsf{d}_2), \\ & \qquad \mathsf{r}_1{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}'_1), \\ & \qquad \mathsf{r}_2{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}'_2), \\ & \qquad \mathsf{r}_3{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)). \\ & \qquad \mathsf{String}(\pi_1(\mathsf{r}_3))); \mathbf{end} \end{aligned}$$

$$\begin{array}{lll}
G_{MR} \upharpoonright \mathsf{Client} & \triangleq & \mathsf{Server}!(\mathsf{d}{:}\mathsf{String}); \\
& & \mathsf{Server}?(\mathsf{res}{:}\Sigma\mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String}, \\
& & \qquad \mathsf{r}_1{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1), \\
& & \qquad \mathsf{r}_2{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1), \\
& & \qquad \mathsf{r}_3{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2)). \\
& & \qquad\quad \mathsf{String}(\pi_1(\mathsf{r}_3))); \mathbf{end} \\
\\
G_{MR} \upharpoonright \mathsf{Worker}_1 & \triangleq & \mathsf{Server}?(\mathsf{d}_1 : \mathsf{String}); \\
& & \mathsf{Aggr}!(\mathsf{r}_1 : \Sigma\mathsf{r} : \mathsf{String}.r = f(d_1)); \mathbf{end} \\
\\
G_{MR} \upharpoonright \mathsf{Aggr} & \triangleq & \mathsf{Server}?(\mathsf{p}{:}\Sigma\mathsf{d}{:}\mathsf{String}, \mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String}.\mathsf{d} = \mathsf{d}_1 \mathrel{++} \mathsf{d}_2); \\
& & \mathsf{Worker}_1?(\mathsf{r}_1{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_1)); \\
& & \mathsf{Worker}_2?(\mathsf{r}_1{:}\Sigma\mathsf{r}{:}\mathsf{String}.\mathsf{r} = f(\mathsf{d}_2)); \\
& & \mathsf{Server}!(\mathsf{r}_3{:}\Sigma r{:}\mathsf{String}.\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2))); \mathbf{end}
\end{array}$$

Fig. 6. Projections for G_{MR} – Client, Worker and Aggr roles.

The projection for the server role illustrates the key elements in our notion of endpoint projection. In the third message (the output to the aggregator), we bundle the information unknown by the aggregator in order to ensure the type is well-formed from the perspective of the recipient. Moreover, the usage of singletons preserves the dependencies specified in the global type (i.e. that the objects in question are indeed those received from the client and subsequently sent to the two worker endpoints). Note that in the input from the aggregator, the projected type does not require singletons since the server endpoint knows the identities of d_1 and d_2.

4 Value Dependent Processes and Typing

This section presents semantics and a typing system of value dependent processes.

4.1 Syntax and Operational Semantics

We define the process syntax, introducing the operational semantics, which is an extension of the synchronous multiparty session π-calculus studied in [11]. We use s to range over *session* names, c to range over *channels* which are either variables z, x or *session* names with *role* $s[\mathsf{p}]$, a to range over *shared* names. The process $\overline{a}[\mathsf{n}](z).P$ is a session initiation *request*, established through synchronisation by rule $\langle$Init$\rangle$, with the complementary accepting processes $a[\mathsf{p}](z).P$ (with $2 \leq \mathsf{p} \leq n$) on a shared channel a. We use $c[\mathsf{p}]$ in all session interactions, where c denotes a channel and p the participant implemented the by other endpoint process. Interactions within a session are: $c[\mathsf{p}]!(M); P$ sends the message M to participant p, continuing as P; and $c[\mathsf{p}]?(x); P$ receives a message or a channel from participant p, binding it to variable x in the continuation P (by rule $\langle$Com$\rangle$) where terms are reduced to values (denoted by $M \Downarrow V$); process $c[\mathsf{p}]!(s); P$ delegates channel s to participant p and continues as P (by rule $\langle$Del$\rangle$). $c[\mathsf{p}] \lhd l; P$

$$
\begin{array}{ll}
\langle\text{Init}\rangle & \overline{a}[\mathsf{n}](z).P_1 \mid \prod_{i\in\{2,\ldots,n\}} a[\mathsf{i}](z).P_i \to (\nu s)(\Pi_{i\in\{1,\ldots n\}} P_i\{s[\mathsf{i}]/z\}) \qquad s \notin fn(P_i) \\
\langle\text{Com}\rangle & s[\mathsf{p}][\mathsf{q}]!\langle M\rangle; P \mid s[\mathsf{q}][\mathsf{p}]?(x); Q \to P \mid Q\{V/x\} \qquad M \Downarrow V \\
\langle\text{Del}\rangle & s[\mathsf{p}][\mathsf{q}]!\langle s'[\mathsf{p}']\rangle; P \mid s[\mathsf{q}][\mathsf{p}]?(x); Q \to P \mid Q\{s'[\mathsf{p}']/x\} \\
\langle\text{Sel}\rangle & s[\mathsf{p}][\mathsf{q}] \triangleright \big(l_i{:}P_i\big)_{i\in I} \mid s[\mathsf{q}][\mathsf{p}] \triangleleft l_j; Q \to P_j \mid Q \qquad j \in I \\
\langle\text{Rec}\rangle & P\{M/x\}\{\mu X(x).P/X\} \mid R \to Q \quad\Longrightarrow\quad \mu X(x = M).P \mid R \to Q \\
\langle\text{NuG}\rangle & P \to P' \quad\Longrightarrow\quad (\nu a : \mathcal{G})P \to (\nu a{:}\mathcal{G})P' \\
\langle\text{NuS}\rangle & P \to P' \quad\Longrightarrow\quad (\nu s)P \to (\nu s)P' \\
\langle\text{Par}\rangle & P \to P' \quad\Longrightarrow\quad P \mid Q \to P' \mid Q \\
\langle\text{Cong}\rangle & (P \equiv P' \quad P' \to Q' \quad Q' \equiv Q) \quad\Longrightarrow\quad P \to Q
\end{array}
$$

$$
\begin{gathered}
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu a : \mathcal{G})P \mid Q \equiv (\nu a : \mathcal{G})(P \mid Q) \text{ if } a \notin fn(Q) \qquad (\nu s)P \mid Q \equiv (\nu s)(P \mid Q) \text{ if } s \notin fn(P) \\
(\nu a : \mathcal{G})(\nu a' : \mathcal{G}')P \equiv (\nu a' : \mathcal{G}')(\nu a : \mathcal{G})P \qquad (\nu s)(\nu s')P \equiv (\nu s')(\nu s)P \\
(\nu a : \mathcal{G})\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)\mathbf{0} \equiv \mathbf{0} \qquad (\nu a : \mathcal{G})(\nu s)P \equiv (\nu s)(\nu a : \mathcal{G})P
\end{gathered}
$$

Fig. 7. Operational semantics of processes – reduction and structural congruence

and $c[\mathsf{p}] \triangleright (l_i{:}P_i)_{i\in I}$ denote, respectively, selecting a label l by communicating with participant p and continuing as P and receiving a label l_i from participant p and continuing as P_i (by rule $\langle$Sel$\rangle$). Recursive process definitions $\mu X(x = M).P$ have the recursion variable x as a formal parameter, instantiated with M in the first iteration (we assume P always contains at least one recursive call on X) (by rule $\langle$Rec$\rangle$). The remaining operational semantics, structure congruence $\equiv$ and context rules which closed under parallel and shared and session name restrictions, are standard.

4.2 Typing System

We now introduce the typing system assigning local types to channels in processes. The key typing rules are given in Fig. 8. We omit the rules that are not particular to our development, such as those for choice, branching, inactivity and session initiation for conciseness. We define the judgement $\Psi; \Gamma; \Delta \vdash P$, where Ψ is a typing context for message terms, Γ a mapping of shared names to global types and process variables to the specification of their variables, and Δ a (linear) mapping of channels to local types. The intuitive reading of the typing judgement is that P uses channels (and recursion variables) according to the types specified in Γ and Δ and message variables according to the types specified in Ψ. We write $\Gamma \vdash a{:}G$ iff $a{:}G \in \Gamma$. We also make use of a typing judgement for message terms $\Psi \vdash M : \tau$, denoting that M has type τ under the typing assumptions recorded in Ψ. We omit this typing judgement for the sake of generality of the underlying type theory. We recall the requirement of

$$\text{(VSEND)}\;\frac{\Psi \vdash M{:}\tau \quad \Psi; \Gamma; \Delta, c{:}T\{M/x\} \vdash P}{\Psi; \Gamma; \Delta, c{:}\mathsf{p}!(x{:}\tau).T \vdash c[\mathsf{p}]!(M); P} \qquad \text{(VRECV)}\;\frac{\Psi, x{:}\tau; \Gamma; \Delta, c{:}T \vdash P}{\Psi; \Gamma; \Delta, c{:}\mathsf{p}?(x{:}\tau).T \vdash c[\mathsf{p}]?(x); P}$$

$$\text{(REC)}\;\frac{\Psi \vdash M : \tau \quad \Psi, x{:}\tau; \Gamma, X : (c, t, x); c{:}T \vdash P}{\Psi; \Gamma; c{:}\mu t(x = M : \tau).T \vdash \mu X(x = M).P}$$

$$\text{(VAR)}\;\frac{\Psi, x{:}\tau \vdash M : \tau}{\Psi, x{:}\tau; \Gamma, X{:}(c, t, x); c{:}t\langle M\rangle \vdash X\langle M\rangle} \qquad \text{(SUB)}\;\frac{\Psi \vdash T \leq T' \quad \Psi; \Gamma; \Delta, c{:}T \vdash P}{\Psi; \Gamma; \Delta, c{:}T' \vdash P}$$

$$\text{(SRES)}\;\frac{\Psi; \Gamma; \Delta, s[1]{:}T_1, \ldots s[n] : T_n \vdash P \quad \mathsf{co}(s[1]{:}T_1, \ldots s[n] : T_n)}{\Psi; \Gamma; \Delta \vdash (\boldsymbol{\nu} s)P}$$

Fig. 8. Local typing rules.

the usual type safety results of progress and type preservation (and so, a substitution principle) in the presence of singleton types and subtyping (as detailed in Sect. 3.1).

Rule (VSEND) types sending of data messages. Sending a datum M binds it to x in the continuation type, as expected in a (value) dependently-typed setting. Sending a channel requires its existence in the context with the appropriate type. Dually, rule (VRECV) types the reception of data, where the process that expects to receive a data message of type τ is warranted to use it in its continuation.

The typing rule for (REC) recursive process definitions assigns a channel with a parameterised recursive type by registering in Γ the necessary information regarding the process recursion variable (the channel name, the recursive type variable and the parameter variable), where the local typing environment must be empty. Rule (VAR) simply matches the process recursion variable with the type recursion variable according to the information if Γ, checking that the recursive parameter is appropriately typed. The remaining rules are standard in the MPST literature [9,11,22], of which we highlight the (SRES) rule for session channel restriction, requiring that all the several role annotatted endpoints be *coherent* (Definition 4.5). Coherence relies on a notion of partial projection (Definition 4.2) and session duality (Definition 4.3), which we now introduce.

Partial projection is defined as a function taking an endpoint type and a role identifier. Intuitively, it extracts from the endpoint type the behaviour that pertains only to the specified role, erasing all role annotations (since the type is now completely localised to a single role – i.e. a binary session type). We range over binary session types with S, T. Duality is defined in the natural way, matching inputs with outputs (appealing to subtyping in the case for data communication to ensure compatibility of the data types); and branching with selection.

$$
\begin{aligned}
(\mathsf{r}!(x{:}\tau);T)\upharpoonright \mathsf{p} &= \begin{cases} !(x:\tau);(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright\mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}?(x{:}\tau);T)\upharpoonright \mathsf{p} &= \begin{cases} ?(x{:}\tau);(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright\mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}!(U);T)\upharpoonright \mathsf{p} &= \begin{cases} !(U);(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright\mathsf{p} & \text{otherwise}\end{cases}\\
(\mathsf{r}?(U);T)\upharpoonright \mathsf{p} &= \begin{cases} ?(U);(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}=\mathsf{r}\\ T\upharpoonright\mathsf{p} & \text{otherwise}\end{cases}\\
(\oplus\mathsf{r}(l_i{:}T_i)_{i\in I})\upharpoonright \mathsf{p} &= \begin{cases} \oplus(l_i{:}(T_i\upharpoonright\mathsf{p}))_{i\in I} & \text{if } \mathsf{p}=\mathsf{r}\\ \sqcup_{i\in I}(T_i\upharpoonright\mathsf{p}) & \text{otherwise}\end{cases}\\
(\&\mathsf{r}(l_i{:}T_i)_{i\in I})\upharpoonright \mathsf{p} &= \begin{cases} \&(l_i{:}(T_i\upharpoonright\mathsf{p}))_{i\in I} & \text{if } \mathsf{p}=\mathsf{r}\\ \sqcup_{i\in I}(T_i\upharpoonright\mathsf{p}) & \text{otherwise}\end{cases}\\
(\mu t(x=M:\tau).T)\upharpoonright \mathsf{p} &= \begin{cases} \mu t(x=M:\tau).(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}\in\mathcal{T} \text{ and } \mathsf{p} \text{ knows } M\\ \mu t.(T\upharpoonright\mathsf{p}) & \text{if } \mathsf{p}\in T \text{ and } \mathsf{p} \text{ doesn't know } M\\ \mathbf{end} & \text{otherwise}\end{cases}\\
t\langle M\rangle\upharpoonright \mathsf{p} &= \begin{cases} t\langle M\rangle & \text{if } \mathsf{p} \text{ knows } M\\ t & \text{otherwise}\end{cases}\\
\mathbf{end}\upharpoonright\mathsf{p} &= \mathbf{end}
\end{aligned}
$$

Fig. 9. Partial projection.

Definition 4.1 (Binary Type Merge). *Let $T = \oplus\big(\mathsf{l}_i : T_i\big)_{i\in I}$ and $T' = \oplus\big(\mathsf{l}'_j : T'_j\big)_{j\in J}$. The merge $T \sqcup T'$ of T and T' is defined as:*

$$
\begin{aligned}
T\sqcup T' &\triangleq \oplus\big(\mathsf{l}_h : T_h\big)_{h\in I\setminus J} \cup \big(\mathsf{l}'_h : T'_h\big)_{h\in J\setminus I} \cup \big(\mathsf{l}_h : T_h \sqcup T'_h\big)_{h\in I\cap J}\\
T\sqcup T &\triangleq T
\end{aligned}
$$

if $\mathsf{l}_h = \mathsf{l}'_h$ for each $h \in I\cap J$. Merge is homomorphic (i.e. $\mathcal{C}[T_1]\sqcup\mathcal{C}[T_2] = \mathcal{C}[T_1\sqcup T_2]$) and is undefined otherwise.

Definition 4.2 (Partial Projection). *The partial projection of a local type T onto p, denoted by $T\upharpoonright\mathsf{p}$, is defined by the rules of Fig. 9.*

Definition 4.3 (Duality). *The duality relation between projections of local types is the minimal symmetric relation satisfying:*

$$
\begin{gathered}
\mathbf{end} \bowtie \mathbf{end} \qquad t\langle M\rangle \bowtie t\langle M'\rangle\\
T\bowtie T' \Longrightarrow (\mu t(x=M:\tau).T) \bowtie (\mu t(x=M':\tau).T')\\
T\bowtie T' \wedge \tau\leq\tau' \Longrightarrow !(x{:}\tau);T \bowtie ?(x{:}\tau');T'\\
T\bowtie T' \Longrightarrow !(U);T \bowtie ?(U);T'\\
\forall i\in I\ T_i\bowtie T'_i \Longrightarrow \oplus(l_i{:}T_i) \bowtie \&(l_i:T'_i)
\end{gathered}
$$

Duality is crucial to ensure compatibility between the different participants in a multiparty conversions. This notion is made precise in the following lemma.

Lemma 4.4. *Let G be a global type and $\mathsf{p} \neq \mathsf{q}$. Then $(G \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (G \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$.*

Coherence thus ensures compatibility of all participants in a multiparty session, requiring that for each role in the multiparty conversation, all performed actions are matched by a dual action performed by the expected recipient.

Definition 4.5 (Coherence). *A session environment Δ is coherent for the session s, written $\mathsf{co}(\Delta, s)$ if $s[\mathsf{p}] : T \in \Delta$ and $s[\mathsf{q}] : T' \in \Delta$ imply $T \upharpoonright \mathsf{q} \bowtie T' \upharpoonright \mathsf{p}$. A session environment is coherent if it is coherent for all sessions which occur in it.*

5 Safety Properties of Value Dependencies

This section lists the main properties of the typing system. Recalling the notion of *history sensitivity* (Definition 2.2), which intuitively requires that in a global type G, for each interaction in which s sends some data of type τ, all free variables of type τ must have been defined in a previous interaction of G involving s, we state soundness of compatible type binding generation.

Theorem 5.1 (Compatible Type Binding Generation is Sound). *Let G be a well-formed, history sensitive global type such that $\mathsf{p} \to \mathsf{q} : (x{:}\tau).G' \sqsubseteq G$. We have that the pair of data types $((CTB(x{:}\tau)), (CTB(x{:}\tau))^\dagger)$ is compatible with p and q.*

Intuitively, it is easy to see that CTB generates pairs of compatible types given the subtyping rules for singletons (and Definition 3.5 of singleton erasure), combined with the fact that CTB collects only data known by p and unknown by q, relevant to the message exchange of type τ.

Given that types intrinsically specify properties of exchanged data, subject congruence and reduction ensure that any well-typed process is guaranteed to conform with its behavioural specification in a strong sense. Subject reduction relies crucially on a substitution principle for types and processes, which is a lifted version of the substitution principle for the dependent data layer.

Lemma 5.2 (Term Substitution). *If $\Psi, x{:}\tau; \Gamma; \Delta \vdash P$ and $\Psi \vdash M{:}\tau$ then we have that $\Psi; \Gamma; \Delta\{M/x\} \vdash P\{M/x\}$.*

Theorem 5.3 (Subject Congruence and Reduction). *If $\Psi; \Gamma; \Delta \vdash P$ and $P \equiv Q$ then there is $\Delta' \equiv \Delta$ such that $\Psi; \Gamma; \Delta' \vdash Q$; and If $\Gamma \vdash P$ and $P \to P'$ then $\Gamma \vdash P'$.*

Note that adherence to the properties of data specified in types is intrinsic: values occurring in well-typed processes act as proof witnesses to the stated properties which are thus inherently satisfied, entailing a notion of communication safety, see [9, Th. 5.5].

Error Freedom. In order to characterise the kind of communication errors that are disallowed by our typing discipline, we define a notion of *extended* process which explicitly references message typing information by considering the following process constructs for (data) input and output:

$$P, Q ::= c[\mathsf{p}]!(M)\{x{:}\tau\}; P \mid c[\mathsf{p}](x)\{x{:}\tau\}; P \mid \ldots$$

We then define a typed reduction and labelled semantics, written $\longmapsto$ and $\xmapsto{\alpha}$, respectively. Typed reduction $\longmapsto$ is defined by the same rules as those of Fig. 7 but where the message synchronisation rule is replaced with (error is a special process construct denoting the error state):

$$\frac{M \Downarrow V \quad \cdot \vdash \tau \leq \tau' \quad \cdot \vdash V : \tau}{s[\mathsf{p}][\mathsf{q}]!(M)\{x{:}\tau\}; P \mid s[\mathsf{q}][\mathsf{p}]?(x)\{x{:}\tau'\}; Q \longmapsto P\{V/x\} \mid Q\{V/x\}}$$

$$\frac{M \Downarrow V \quad \cdot \not\vdash \tau \leq \tau' \vee \cdot \not\vdash V : \tau}{s[\mathsf{p}][\mathsf{q}]!(M)\{x{:}\tau\}; P \mid s[\mathsf{q}][\mathsf{p}]?(x)\{x{:}\tau'\}; Q \longmapsto \mathsf{error}}$$

Equipped with extended processes and typed reduction, we may then show that well-typed (extended) processes never reach an error state.

Theorem 5.4 (Error Freedom). *Let $\Gamma \vdash P$ and $P \longmapsto^* P'$. Then* error *is not a subterm of P'.*

6 Specification Without Communication

So far we have mostly been concerned with the challenges of certifying data exchanges in a multiparty setting by having process endpoints exchange explicit proof objects. However, it is quite often the case that we may wish to reference data and constraints that are reasonable at a specification level but that have little computational interest at runtime. For instance, consider the following global type,

$$\mathsf{p} \to \mathsf{q} : (x{:}\mathsf{Nat}).\mathsf{p} \to \mathsf{q} : (y{:}x > 2).G \tag{6}$$

In the example above, participant p sends q some natural number x, followed by a proof denoting that x is greater than 2. While such an exchange does ensure that a well-typed implementation of the endpoint p must necessarily send to q an integer greater than 2, the endpoint q may have little interest in actually receiving a proof that $x > 2$. Rather, the exchange denoted by the second message from p to q appears as an encoding artefact due to the fact that the framework requires explicit proof exchanges by default.

While it is the case that we could omit a second exchange by "currying" the two communication actions into a pair,

$$\mathsf{p} \to \mathsf{q} : (x{:}\Sigma a{:}\mathsf{Nat}.a > 2).G \tag{7}$$

the issue still remains that we are forced to send potentially unnecessary data.

We can alleviate this issue through the usage of a proof irrelevance modality, written $[\tau]$, denoting that there exists a term of type τ (and thus, a proof of τ), but the identity of the term itself is deemed computationally irrelevant. To make this notion precise, we appeal to a new class of typing assumptions $x{\div}\tau$, meaning that x stands for a term of type τ that is not computationally available; and to a promotion operation on contexts (written $\Psi^{\oplus}$) mapping computationally irrelevant assumptions to ordinary ones:

$$\frac{\Psi^{\oplus} \vdash M : \tau}{\Psi \vdash [M] : [\tau]}\ ([]\mathrm{I}) \qquad \frac{\Psi \vdash M : [\tau] \quad \Psi, x \div \tau \vdash N : \sigma}{\Psi \vdash \mathsf{let}\ [x] = M\ \mathsf{in}\ N : \sigma}\ ([]\mathrm{E})$$

Given that we are only warranted in using irrelevant assumptions within proof irrelevant terms, it is easy to see that proof irrelevance cannot affect the computational outcome of a program and so we may consistently erase proof irrelevant terms at runtime.

We combine this notion of proof irrelevance with an erasure operation that eliminates communication of proof irrelevant terms – since they may not be used in a computationally significant way, they bear no impact on the computational outcome of the session. For instance, we may rewrite the global type of (7) as:

$$\mathsf{p} \to \mathsf{q} : (x{:}\Sigma a{:}\mathsf{Nat}.[a > 2]).G \tag{8}$$

marking that the proof of $a > 2$ is not computationally significant, but must exist during type-checking.

With the combined use of proof irrelevance and erasure we ensure that the specification must still hold, in the sense that the proof objects must be present in endpoint processes for the purposes of type-checking, but are then omitted at runtime to minimise potentially unnecessary communication. An alternative approach, only feasible for decidable theories, would be to generate proofs automatically by appealing to some external decision procedure.

6.1 Erasure of Proof Irrelevant Terms

We introduce a simple erasure procedure on types, processes and terms that replaces proof irrelevance with the unit type (and the unit element at the term level). Recall that since proof irrelevant terms have no bearing on the computational outcome of programs, the erasure is safe w.r.t the behaviour of programs.

Definition 6.1 (Erasure). *We inductively define erasure on local types, terms and processes, written $T^{\downarrow}$ (resp. $\tau^{\downarrow}$, $M^{\downarrow}$ and $P^{\downarrow}$) by the following rules (we show only the most significant cases, all others simply traverse the underlying structure inductively):*

$$\begin{array}{llll}
(\mathsf{p}!(x{:}\tau);T)^{\downarrow} & \triangleq \mathsf{p}!(x{:}\tau^{\downarrow});T^{\downarrow} & (\mathsf{p}!(x{:}\tau);T)^{\downarrow} & \triangleq \mathsf{p}?(x{:}\tau^{\downarrow});T^{\downarrow} \\
(\oplus\mathsf{p}(\mathsf{l}_p{:}T_i)_{i\in I})^{\downarrow} & \triangleq (\oplus\mathsf{p}(\mathsf{l}_p{:}T_i\downarrow)_{i\in I}) & (\&\mathsf{p}(\mathsf{l}_p{:}T_i)_{i\in I})^{\downarrow} & \triangleq \&\mathsf{p}(\mathsf{l}_p{:}T_i^{\downarrow})_{i\in I} \\
(\mathsf{p}!(U);T)^{\downarrow} & \triangleq \mathsf{p}!(U^{\downarrow});T^{\downarrow} & (\mathsf{p}?(U);T)^{\downarrow} & \triangleq \mathsf{p}?(U^{\downarrow});T^{\downarrow} \\
(\mu t.(x = M : \tau).T)^{\downarrow} & \triangleq \mu t.(x = M^{\downarrow} : \tau^{\downarrow}).T^{\downarrow} & t\langle M\rangle^{\downarrow} & \triangleq t\langle M^{\downarrow}\rangle
\end{array}$$

$$\begin{array}{llll}
(\overline{a}[\mathsf{n}](z).P)^{\downarrow} & \triangleq \overline{a}[\mathsf{n}](z).P^{\downarrow} & (a[\mathsf{p}](z).P)^{\downarrow} & \triangleq a[\mathsf{p}](z).P^{\downarrow} \\
(c[\mathsf{p}]!(M);P)^{\downarrow} & \triangleq c[\mathsf{p}]!(M^{\downarrow});P^{\downarrow} & (c[\mathsf{p}]!(s);P)^{\downarrow} & \triangleq c[\mathsf{p}]!(s);P^{\downarrow} \\
(c[\mathsf{p}]?(x);P)^{\downarrow} & \triangleq c[\mathsf{p}]?(x);P^{\downarrow} & (c[\mathsf{p}] \lhd l;P)^{\downarrow} & \triangleq c[\mathsf{p}] \lhd l;P^{\downarrow} \\
(c[\mathsf{p}] \rhd \left(l_i{:}P_i\right)_{i\in I})^{\downarrow} & \triangleq c[\mathsf{p}] \rhd \left(l_i{:}P_i^{\downarrow}\right)_{i\in I} & (\mu X(x = M).P)^{\downarrow} & \triangleq \mu X(x = M^{\downarrow}).P^{\downarrow} \\
X\langle M\rangle^{\downarrow} & \triangleq X\langle M^{\downarrow}\rangle & &
\end{array}$$

$$\begin{array}{llll}
(\Pi x : \tau.\sigma)^{\downarrow} & \triangleq \Pi x : \tau^{\downarrow}.\sigma^{\downarrow} & (\Sigma x : \tau.\sigma)^{\downarrow} & \triangleq \Sigma x : \tau^{\downarrow}.\sigma^{\downarrow} \\
b^{\downarrow} & \triangleq b & \mathcal{S}(M)^{\downarrow} & \triangleq \mathcal{S}(M^{\downarrow}) \\
[\tau]^{\downarrow} & \triangleq \mathsf{unit} & (\lozenge_{\mathsf{p}}\tau)^{\downarrow} & \triangleq \lozenge_{\mathsf{p}}\tau^{\downarrow}
\end{array}$$

$$\begin{array}{llll}
[M]^{\downarrow} & \triangleq \langle\rangle & (\lambda x.M)^{\downarrow} & \triangleq \lambda x.M^{\downarrow} \\
\langle M, N\rangle^{\downarrow} & \triangleq \langle M^{\downarrow}, N^{\downarrow}\rangle & \langle\rangle^{\downarrow} & \triangleq \langle\rangle
\end{array}$$

The goal of the erasure function above is to essentially replace all instances of proof irrelevant objects with the unit element $\langle\rangle$. We note that it is not in general the case that $(G \upharpoonright \mathsf{p})^{\downarrow} = (G^{\downarrow}) \upharpoonright \mathsf{p}$, since projection of an erased global type may not need to preserve some dependencies that were present in the original global type. We may then consistently erase communication of messages of unit type.

Definition 6.2 (Communication Erasure). *We write T^*, P^*, M^* and τ^* for the following erasure (the remaining cases are obtained by homomorphic extension):*

$$(\mathsf{p} \to \mathsf{q} : (x{:}\mathsf{unit}).G)^* \triangleq G^* \qquad (\mathsf{p}!(x{:}\mathsf{unit}).T)^* \triangleq T^* \qquad (\mathsf{p}?(x{:}\mathsf{unit}).T)^* \triangleq T^*$$

To summarise, our proposed methodology for the usage of ghost variables in specifications is to mark specification-level terms as proof irrelevant at the level of global types. Projection is then performed on the global type, propagating the proof irrelevance accordingly to the local types which we use to type the several endpoint processes (which contain the explicit proof objects, now marked as proof irrelevant).

Having successfully checked the endpoints, we may then perform the erasure procedure(s) described above: by performing the erasure $\downarrow$ on the original global type, we generate local types in which instances of proof irrelevance have been replaced with unit; by applying the erasure * we eliminate irrelevant communication actions from types, erasing those actions from the endpoints and refactoring message payloads accordingly. We thus remove unnecessary communication actions but ensure that specified properties are satisfied.

MapReduce. In our running example of a certified MapReduce-style computation, where the global type G_{MR} specifies that the workers and aggregator endpoints indeed perform the appropriate computation, we may easily apply the ghost variable technique introduced in this section to eliminate several potentially unnecessary communication steps, including the potentially excessive passing of data that appears in local types as an artefact of endpoint projection.

Consider the following revision of G_{MR}, referred to as $G_{[MR]}$, where we mark the message from the server to the aggregator which informs of the partition of data as proof irrelevant ($\mathsf{Server} \to \mathsf{Aggr} : (\mathsf{p} : [\mathsf{d} = \mathsf{d}_1 \mathbin{++} \mathsf{d}_2])$), and similarly for the proofs that the sent data objects are indeed the results of performing the specified computation ($\mathsf{Worker}_i \to \mathsf{Aggr} : (\mathsf{r}_i : \Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_i)])$ and $\mathsf{Aggr} \to \mathsf{Server} : (\mathsf{r}_3 : \Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_2(\mathsf{r}_2))])$). We then have the following endpoint projections:

$$\begin{aligned}
G_{[MR]} \upharpoonright \mathsf{Client} \triangleq\ & \mathsf{Server!}(\mathsf{d}{:}\mathsf{String});\\
& \mathsf{Server?}(\mathsf{res}{:}\Sigma \mathsf{d}_1{:}\mathsf{String}, \mathsf{d}_2{:}\mathsf{String},\\
& \qquad \mathsf{r}_1{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_1)],\\
& \qquad \mathsf{r}_2{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = f(\mathsf{d}_1)],\\
& \qquad \mathsf{r}_3{:}\Sigma \mathsf{r}{:}\mathsf{String}.[\mathsf{r} = g(\pi_1(\mathsf{r}_1), \pi_1(\mathsf{r}_2)]).\\
& \qquad\quad \mathsf{String}(\pi_1(\mathsf{r}_3))); \mathbf{end}\\
G^{\downarrow}_{[MR]} \upharpoonright \mathsf{Client} \triangleq\ & \mathsf{Server!}(\mathsf{d}{:}\mathsf{String});\\
& \mathsf{Server?}(\mathsf{res}{:}\Sigma \mathsf{r}_3{:}(\Sigma \mathsf{r}{:}\mathsf{String}.\mathsf{unit}).\mathsf{String}(\pi_1(\mathsf{r}_3)))
\end{aligned}$$

By performing the erasure before projecting, we eliminate a substantial amount of dependency information from the local type for the Client. We note that transforming processes satisfying $G_{[MR]} \upharpoonright \mathsf{Client}$ into ones satisfying $G^{\downarrow}_{[MR]} \upharpoonright \mathsf{Client}$ can easily be achieved by some simple program transformations, related to those used in program generation for dependently typed functional languages [14,18].

6.2 Soundness of Erasure

We make precise the static and dynamic soundness of our erasure procedures. The static safety theorem (Theorem 6.3) states that a consistent usage of erasures does not violate the typing discipline.

Theorem 6.3 (Static Safety). *Let $\Psi; \Gamma; \Delta \vdash P$. Then we have that:*

(a) $\Psi^{\downarrow}; \Gamma^{\downarrow}; \Delta^{\downarrow} \vdash P^{\downarrow}$
(b) $(\Psi^{\downarrow})^*; (\Gamma^{\downarrow})^*; (\Delta^{\downarrow})^* \vdash (P^{\downarrow})^*$

We also show that erased processes have an operational correspondence with their un-erased counterparts.

Theorem 6.4 (Operational Correspondence). *Let $\Gamma \vdash P$:*

(a) If $P \to P'$ then $P^{\downarrow} \to P'^{\downarrow}$
(b) If $(P^{\downarrow})^* \to P'$ then $P \to^* Q$ such that $(Q^{\downarrow})^* \equiv P'$.

For the erasure of Definition 6.1, we have a very precise operational correspondence by virtue of the computational insignificance of proof irrelevant terms (Theorem 6.4 (a)). For the communication erasure of Definition 6.2, the processes in the image of the erasures may naturally produce less reductions, which account for the erased communication steps. However, we can ensure that a reduction in an erased process $(P^{\downarrow})^{*} \rightarrow P'$ can be matched by potentially a sequence of reductions in the original process P, such that applying the two erasures to the reduct of P produces a structurally equivalent process to P' (Theorem 6.4, (b)).

7 Conclusion

In this section we present some additional discussion of key design choices and avenues of future work and conclude.

7.1 Further Discussion

We discuss two fundamental considerations: some of the particulars of compatible type binding generation and how they may be potentially simplified through a fundamental use of proof irrelevance akin to that of Sect. 6; and how our design choices affect a potential implementation of the language discussed in this paper, specifically what should be verified statically and/or dynamically and in what circumstances does the type system help guide these choices.

Compatible Type Binding Generation. In Sect. 3 we have discussed the challenges of defining projection in the presence of value dependencies, given that each participant only has a partial view of the global protocol and thus the notion of projection from previous works on multiparty sessions produce endpoint types that are either not well-formed or fail to capture the appropriate data restrictions specified in global types.

We address this issue by bundling in each message exchange all the information that is unknown by message recipients such that the endpoint types for senders and receivers are well-formed, compatible and preserve the intended semantics of global types. A potential disadvantage of our approach is that the bundling procedure can insert a non-trivial amount of extra information in message exchanges. For instance, in the global type:

$$\begin{aligned} G \triangleq\ & A \rightarrow B : (x_1{:}\tau_1). \\ & A \rightarrow B : (x_2{:}\tau_2). \\ & \qquad \vdots \\ & A \rightarrow B : (x_n{:}\tau_n). \\ & A \rightarrow C : (y{:}\Sigma z : \tau.\mathcal{P}(z, x_1, \ldots, x_n)) \end{aligned}$$

Participant A sends to participant B a sequence of n messages, after which it sends to participant C a message of type $\Sigma z : \tau.\mathcal{P}(z, x_1, \ldots, x_n)$, such that C does not know x_1 through x_n. In this scenario, when projecting the exchange of message y, A must not only send the object of type $\Sigma z : \tau.\mathcal{P}(x_1, \ldots, x_n)$ but also the n messages previously sent to participant B which are unknown to C.

In Sect. 6 we have introduced a way of minimising potentially unnecessary communications through the usage of proof irrelevance and type-oriented erasure, where certain portions of data types are marked as irrelevant and subsequently erased. At the level of compatible type binding generation, it should also be possible to directly make use of proof irrelevance to reduce the communication overheads mentioned above, given that unknown message variables have a somewhat proof irrelevant flavour (i.e. they cannot be used precisely because they are unknown).

One potential approach to this issue is to introduce a proof-irrelevant Σ-type, written $\Sigma x \div \tau.\sigma$ and modify compatible type binding generation to quantify over unknown variables using these proof irrelevant pairs. The main issue is that it forces type families using such proof irrelevant variables to be themselves proof irrelevant. We leave these challenges for future work.

On Implementation. The main contribution of this paper is the conceptual extension of the multiparty session typed framework with value dependencies, the language presented in this paper leads itself towards more realistic implementation considerations. Our basic foundation is that proofs are exchanged as witnesses to the properties specified in types. These proof objects are no more than terms from a dependently-typed language, but one may wonder if it is indeed feasible to explicitly exchange proof objects instead of somehow verifying the necessary properties dynamically. Regardless, given the potential distributed nature of the framework, it seems natural to require that communicated proof objects be checked at runtime on the recipient side.

While there are circumstances where proof objects may not be exchanged and instead generated (or have the necessary properties checked) at runtime on the receiver side, this requires the property language to be decidable, which may be too restrictive (we note that we require proof *checking* to be decidable, which is a significantly weaker condition), although a potentially reasonable assumption in certain application scenarios, or for settings where programmers cannot be expected to write proof objects by hand. Another alternative worth considering is to have participants exchange digitally signed objects that hold them accountable for the existance of certain proofs.

While there seems to be a somewhat flexible range of possibilities in terms of proof communication, it is unavoidable that an implementation of the language integrates both static and dynamic checking in order to ensure that the specified properties indeed hold throughout execution. To this end, an extension of the system where trust amongst session participants is determined *a priori* might help guide the static/dynamic verification procedures. For instance, among trusted participants one may avoid dynamic checks entirely, among "somewhat" trusted participants, digital proof certificates might be required, but not the complete proof objects, whereas communication with untrusted agents would require the full range of dynamic and static checks.

We are currently investigating some of these avenues of research, in particular how the exchange of signed certificates interacts with assurances of data provenance. Another aspect that needs to be studied is the potential leakage of sensitive information that may occur while generating compatible endpoint types due to unknown message dependencies, which should be alleviated by the techniques discussed in the sections above.

7.2 Concluding Remarks

We shall conclude this paper with a quotation from the end of Ambition and Vision in [1] to record what we promised for the future:

Ambition and Vision [1]*. As with data types, we expect session types to play a role in all aspects of software. Today,* architects *model systems using types that are directly supported in the programming language, whereas they model communications using protocols that have no direct support in the programming language; tomorrow, they will model communication using session types that are directly supported in the programming language. Today,* programmers *use interactive development environments that prompt for methods based on types, and give immediate feedback indicating where code violates type discipline, whereas they have no similar support for coding communications; tomorrow, interactive development environments will prompt for messages based on session types, and give immediate feedback indicating where code violates session type discipline. Today,* software tools *exploit types to optimise code, whereas they do not exploit protocols; tomorrow, software tools will exploit session types to optimise communication. In short, architects, programmers, and software tools will all be aided by session types to reduce the cost of producing concurrent and distributed software, while increasing its reliability and efficiency.*

The work introduced in this paper is a small step to move towards the ambition and vision of the ABCD project statement, proposing a session type discipline that builds on the current usages of data types as a tool for modelling, developing and improving modern software, integrating them at a deep level with MPST in order to provide a unified framework for statically certified programs, from computation to communication.

References

1. ABCD: A basis for concurrency and distribution. http://groups.inf.ed.ac.uk/abcd/
2. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
3. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
4. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Types in Language Design and Implementation, pp. 1–12 (2012)
5. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Switzerland (2015)
6. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Logical Meth. Comput. Sci. **8**(4), 1–46 (2012)
7. Gay, S., Hole, M.: Subtyping for session types in the Pi calculus. Acta Informatica **42**(2–3), 191–225 (2005)

8. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284 (2008)
10. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
11. Kouzapas, D., Yoshida, N.: Globally governed session semantics. Logical Meth. Comput. Sci. **10**(4), 1–45 (2014)
12. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 560–584. Springer, Heidelberg (2015)
13. MRG: Mobility reading group. http://mrg.doc.ic.ac.uk/
14. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007
15. Pfenning, F., Caires, L., Toninho, B.: Proof-carrying code in a session-typed process calculus. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 21–36. Springer, Heidelberg (2011)
16. Stone, C.A., Harper, R.: Extensional equivalence and singleton types. ACM Trans. Comput. Log. **7**(4), 676–722 (2006)
17. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817. Springer, Heidelberg (1994)
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4pl2* (2013)
19. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP 2011, pp. 161–172 (2011)
20. Wadler, P.: Propositions as sessions. In: ICFP 2012, pp. 273–286 (2012)
21. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2–3), 384–418 (2014)
22. Yoshida, N., Deniélou, P.-M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010)

Recursion Equations as a Programming Language

D.A. Turner(✉)

Computing Laboratory, University of Kent, Canterbury, UK
D.A.Turner@kent.ac.uk

Abstract. This paper was written in 1981 and published in Darlington Henderson and Turner (1982) pp 1–28. The volume comprised the lecture notes from a summer school on Functional Programming held at Newcastle University in July 1981, attended by 80 people. The paper includes an overview of Kent Recursive Calculator, a simple functional programming system based on higher order recursion equations and a series of programming examples. It is probably the earliest paper using list comprehensions applied to lazy lists and has the first published account of the "list of successes" method of eliminating backtracking, here applied to the eight queens problem. The method didn't yet have a name. It was Phil Wadler who saw its importance and coined the phrase "list of successes" in his 1985 paper. It was also Phil who invented the term "list comprehensions" for what are here called "ZF expressions".

Keywords: Lazy evaluation · List comprehensions · "List of successes" · Kent recursive calculator

The last few years have seen a growing interest in functional (or applicative) languages as a potential alternative to conventional programming languages, particularly since Backus's Turing lecture (Backus 1978). This interest arises from two distinct causes, one coming from software considerations and the other from developments in the hardware. On the software side, there is mounting evidence that the collection of ideas that came to maturity in the late sixties and are loosely called "structured programming" have simply failed to deliver the reduction in software costs that was originally hoped for — perhaps because the break then proposed with traditional programming practices was insufficiently radical. At the same time, people on the hardware side are searching for new architectures, and therefore new methods of programming, that are capable of taking advantage of the possibility of a very large degree of concurrency in the machine, a possibility that is opening up because of the development of VLSI.

The importance of applicative languages lies in the fact that they hold out the promise of being able to solve both of these problems at the same time.

This paper is included with the kind permission of Cambridge University Press.

In both cases, the key step is the abolition of the assignment statement and with it the notion of sequencing. Before looking in more detail at an applicative language and its properties it is worthwhile to review the software and hardware arguments for considering such a radical change in our programming practice.

1 The Software Crisis and Its Causes

It is commonly observed that we have a software crisis and in a gathering of computer scientists it should not be necessary to multiply examples. Everyone has their own favourite horror story about a project that failed in some catastrophic way because of a bug in a program. Less spectacular but equally worrying is the high cost of producing software even for comparatively simple applications. It is by now clear that the largest single obstacle to the wider use of computers is our inability to produce cheap, reliable and manageable software. The more dramatic the advances on the hardware side the more embarrassing this fact becomes. Does it seem too unreasonable to suggest that there is something fundamentally wrong about the way in which we produce software?

I shall argue that the basic problem lies in the nature of existing programming languages. Existing programming languages emerged in a relatively short period between 1955 and 1960 — Fortran and Cobol set the pattern for later languages. They have evolved since primarily by way of becoming more complicated — the underlying principles have not changed. When we compare, say, Pascal with Fortran, the similarities are much more significant than the differences. More precisely, the differences are superficial but the similarities are fundamental. At a certain level of abstraction all the programming languages in production use today are the same. All are sequential, imperative languages with assignment as their basic action.

When we compare our programming languages with all previous mathematical notation, however, the differences are very striking. Mathematical notation has evolved over many centuries and obeys certain basic rules which are common to every area of mathematics and which give mathematical notation its deductive power. Let us briefly enumerate some of these basic properties of mathematical notation.

First of all mathematics is static. There is no equivalent in mathematics of the programming language notion of a procedure which gives a different answer each time you call it. The mathematical idea of a function is a fixed table of input–output pairs. Given the same f and the same x, the value of $f\ x$ must always be the same. This is so even when mathematics is used to describe processes of change as in physics. We proceed by making time a parameter. That is we formalise the notion of a three dimensional world which is changing by talking about a static four dimensional world. Physics as a science became possible only because Newton showed us how to reduce dynamics to statics in this way.

Secondly, and relatedly, there is in mathematics a certain kind of consistency in the use of names — consider for example the equation

$$x^2 - 2x + 1 = 0$$

which has only one solution, namely $x = 1$. Now, supposing someone were to say "no, there is another solution — we can take the first occurrence of x to be 3 and the second to be 5, giving $3 \times 3 - 2 \times 5 + 1 = 0$, which is also correct", what would we say? We would say that this proposed "solution" is invalid because it ignores a basic premise of the whole exercise, namely that x is supposed to stand for the *same* value throughout its scope, otherwise there would have been no point in always calling the value x. Paradoxical though it may sound, in mathematics variables do not vary; they stand for a constant value throughout their scope.

These basic properties of mathematical notation have been termed by logicians "referential transparency" (Russell and Whitehead 1925, Quine 1960). In mathematics, an expression is used only to refer to (or *denote*) a value and the same expression always denotes the same value (within the same scope).

Finally, we can relate this to the notion of equality, which plays a fundamental role in mathematical reasoning. Two expressions are said to be equal if and only if they denote the same value. An immediate consequence of referential transparency is that equality is substitutive — equal expressions are everywhere interchangeable. It is this which gives mathematical notation its deductive power.

Now, how do our conventional programming languages relate to this tradition? It is clear that they don't adhere to it, because in introducing the assignment statement, which can change the value of a variable in the middle of its scope, they have broken the basic ground rules of mathematical notation. Instead of being referentially transparent, programming languages are *referentially opaque*. In fact, the things that we call "variables" in languages like Algol are not really variables in the mathematical sense at all — in the last analysis they are names for registers in the store of a Von–Neumann computer.

It is because of this that it is difficult to reason about programs. Since expressions can change their value through time, equality is not substitutive. Indeed, in a programming language it does not even have to be true that an expression is equal to itself — because the presence of side effects may mean that evaluating the same expression twice in succession can produce two difference answers! In general it is not possible to reason about such programs on the basis of a static analysis of the program text — instead, we have to think of the program dynamically and follow the detailed flow of control, and this seems unreasonably difficult.

A particularly sharp symptom of the software crisis is the fact that after more than a decade of intensive effort — starting with say Floyd (1967) and Hoare (1969) — we still do not have anything resembling a practically viable set of techniques for giving formal proofs of program correctness on production programs. It seems likely that this is because existing programming languages lack the basic substitution properties on which a smooth running proof theory could be built.

Apart from the problems connected with their referential opacity, the other basic difficulty with existing programming languages is that they are very long–winded, in terms of the amount one has to write to achieve a given effect. Even a comparatively straightforward program like a compiler can easily run to ten thousand lines and there exist commercial packages up to a million lines long.

A number of studies carried out in industry have shown that a given programmer tends to produce a relatively fixed number of lines of code per year — typically around 1500 lines of debugged and documented code — and while the number of lines varies quite a lot from programmer to programmer, it is for a given individual *largely independent* of the language in which he is working — for example, it doesn't seem to matter whether it is assembly code or PL/1.

The significance of this result is that it means that the most important single variable in determining software production costs, apart from the quality of the programmers, is the level of language at which they are working. The reason why FORTRAN was such an enormous step forward, for example, is that programs written in FORTRAN are from five to ten times shorter than the equivalent assembly code. Other things being equal then, the FORTRAN programmer is from five to ten times more productive than the assembly code programmer.

Our problem today is that in the twenty five years that have elapsed since the invention of FORTRAN, we have failed to produce any further substantial improvement in this basic ratio of expressive power. If you compare a program written in a "modern" imperative language, such as PASCAL or ADA with its FORTRAN equivalent, you will not find it very much shorter — in fact, it might even be longer because of the extra declaratory information necessitated by the current fashion for very restrictive forms of strong typing.

It is becoming clear that in order to solve the software crisis, we have to find a way to move up to a whole new level of language that will be more expressive than our conventional high level languages by about the same ratio as our conventional languages were better than assembly code. That sort of increase in power can only arise by letting go of a level of detail that our present programming languages force us to express — which seems to imply a move to some kind of non-procedural language.

In fact, our experience to date with applicative programming is very promising in this respect. Programs written in languages like SASL (Turner 1976, 1981) are consistently an order of magnitude shorter than the equivalent programs in a conventional high level language. We will see some examples of programs in this style below, but at this stage we can give a general reason why the change from an imperative language to a descriptive one should lead to programs becoming so much shorter.

Expressed at an appropriate level of abstraction (for example as a dataflow graph — see Treleaven 1979), an algorithm is a partially ordered set of computations, the partial ordering being imposed by data dependencies. In order to execute an algorithm on a Von–Neumann computer, however, we have to convert this to a total ordering, in one of the many possible ways, and organise storage for the intermediate results. In an imperative programming language both of these tasks must be carried out by the programmer with the result that he has to specify a great deal of extra information. A second reason why conventional high level languages are so long winded is that they lack certain necessary abstraction tools, in particular higher order functions, of which we shall say more below.

In summary, a good case can be made out for saying that the fundamental cause of the software crisis is the imperative and machine oriented nature of our programming languages, and that to overcome it we have to abandon the use of side effects and programmer control of sequencing in favour of purely functional notation. The theoretical possibility of programming in a purely functional style has been known for two decades — the obstacle to its use in practice has always been the difficulty of achieving acceptable efficiency in the use of existing hardware while using such techniques. The current rebirth of interest in functional programming is largely triggered by the fact that developments are now taking place on the hardware side which seem likely to overturn this situation.

2 The Development of VLSI and the Challenge of Parallelism

The basic design of the computer was laid down by John Von Neumann in the 1940 s and has remained largely unaltered since (see Fig. 1). There is a single active processor and a large passive store — the connection between the processor and the store is relatively narrow, only one word in the store can be accessed at a time. Initially, and for a long time afterwards, the processor and the store were made of two fundamentally different technologies, and the processor was very much the more expensive of the two.

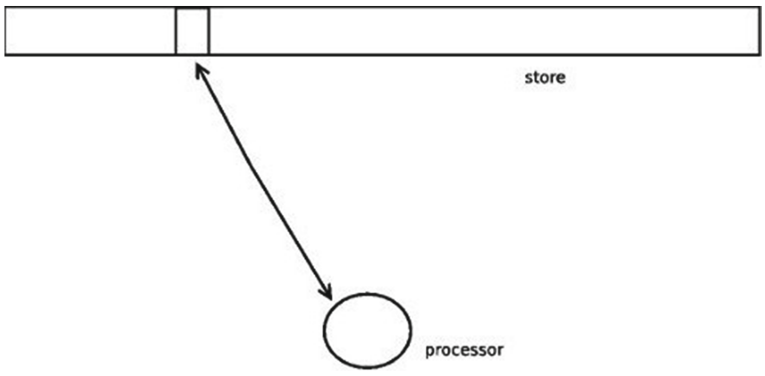


Fig. 1. The Von Neumann computer

The rationality of this arrangement is now being fatally undermined by the development of VLSI. First of all, note that processor and memory are now built of the same technology, namely VLSI chips. Moreover, processing power is becoming very cheap indeed. There is no longer any compelling reason for building mono–processor architectures — it would make equally good economic sense to build a machine which had a network of many processors.

Secondly, we have an obvious motive for doing so — to obtain increased performance. The speed of operation of a conventional Von Neumann computer is limited basically by the bandwidth of the connection between the processor and memory — Backus has called this "the Von Neumann bottleneck". In order to make such a computer go faster we have to improve the technology out of which the components are built, and there are obvious limits to this process. In a multiprocessor architecture, by contrast, we can obtain arbitrary increases in speed simply by adding more processors to the network — provided of course, and this is crucial, that we can find ways of programming it that exploit the potential concurrency.

The first step towards using the possibilities for parallelism opened up by VLSI has been taken by the development of the various "array processors" now appearing on the market — for example, the ICL "DAP". These capture a particular type of parallelism, which can be called "lockstep parallelism" in which the same instruction is performed simultaneously on a large number of data items.

This is appropriate only to certain specialised applications. A more general type of parallelism is where we have many processors each executing different instructions. A number of architectures of this general type are now under development, of which the best known are the various kinds of dataflow computer — see for example (Dennis 1979).

The potential performance of this type of architecture is enormous (thousands of megaflops, using current technology) but how can they be programmed? An idea that can be dismissed more or less straight away is that we should take some conventional sequential language and add facilities for explicitly creating and co–ordinating processes — the tasking facilities of ADA are an example of this approach. This may work where the number of processes is small, but when we are talking about thousands and thousands of independent processes, this cannot possibly be under the conscious control of the programmer.

Parallelism on this scale can only arise from some basic asynchronousness of the language being used. Workers in dataflow are converging on the use of functional languages as a solution to this problem — see Ackermann and Dennis (1979), Arvind, Gostelow and Plouffe (1978). Paradoxically then, notwithstanding their former reputation for inefficiency, it is precisely the need for higher performance that may ultimately force the adoption of functional languages.

Incidentally, the historical efficiency disadvantage of functional languages arises partly from the fact that they have been running on machines with inappropriate instruction sets. There are simple theoretical arguments which show that given an appropriately designed instruction set there is no reason in principle why functional programs should be less efficient than the corresponding imperative programs, even on a Von Neumann machine. It is therefore welcome that in addition to the work currently being done with parallel architectures, some recent efforts have been directed towards the development of sequential machines specifically adopted to functional languages (Clarke et al. 1980; Holloway et al. 1980).

3 A Simple Language Based on Higher Order Recursion Equations

For the sake of having a definite syntax to work with, I will give the ensuing examples of functional programming in the notation of KRC ("Kent Recursive Calculator") a system I have implemented at the University of Kent and which I have been using for teaching purposes. It is fairly closely based on the earlier language SASL (Turner 1976) which I developed while working at the University of St Andrews in the period 1972–1976, but I have added a new language feature based on Zermelo–Frankel set abstraction.

```
DEFINE(((A (LAMBDA (M N)
      (COND ((ZEROP N) (PLUS N 1))
            ((ZEROP N) (A (SUB1 M) 1))
            (T (A (SUB1 M) (A M (SUB1 N)))
      )))))
```

<u>Ackermann's function in LISP</u>

```
      A 0 n = n + 1
      A m 0 = A (m-1) 1
      A m n = A (m-1) (A m (n-1))
```

<u>Ackermann's function in KRC</u>

Fig. 2. Ackermann's function in LISP and KRC

Perhaps I should explain why I don't teach my students LISP (McCarthy et al. 1962) which is still the language most people first think of when functional programming is mentioned. There are two reasons — the first is that the syntax of LISP is so clumsy that it constitutes a real obstacle to comprehension. Figure 2 illustrates this with a definition of Ackermann's function in LISP — note that there are eighteen pairs of parentheses! — and for contrast a definition of the same function in KRC, which looks more like a piece of ordinary mathematics. The second and more serious reason is that the semantics of LISP are rather complicated and include a number of features which could in no sense be regarded as functional. The nett effect, at least in my experience, is that as a vehicle for teaching people about functional programming, LISP is apt to cause more confusion than enlightenment and I prefer to avoid using it.

KRC is purely a functional language — there are no side effects and no concept of flow of control. A program in KRC (actually, we call it a "script") is a set of equations giving mathematical definitions of various entities in which the user is interested. For example, a simple script might be

```
r = u / v
u = x + y
v = x - y
x = 23
y = 10
```

The order in which the above equations are listed is of no significance — we have shown them in alphabetical order but that is purely for clerical convenience. The KRC system is interactive and includes built in commands for editing scripts, saving them in and retrieving them from, files and so on. In particular, the user can ask to have expressions evaluated in the environment established by the script. So for example, typing

```
r?
```

here causes the value of r to be printed at the terminal.

The only ordering of calculations established by a KRC script is that implied by the data dependencies — so for example, in the above case u and v must be calculated before r, but that is the only constraint — note in particular that u and v could be calculated in parallel.

The types of object in KRC's universe of discourse are — numbers, strings, written e.g. `"pig"`, lists and functions. Numbers and strings have the sorts of properties one would expect with the usual sorts of operators defined on them. Lists are written using square brackets and commas, thus

```
days = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

elements of a list are accessed by indexing[1]. So for example, the expression

```
days 3
```

would here have the value `"wed"`. The operator `#` takes the length of a list, so

```
# days
```

is here 7. Another important operator on lists is "`:`" which adds a new element at the front, corresponding to the LISP function "CONS". So for example, the expression

```
0 : [1, 2, 3]
```

takes the value [0, 1, 2, 3]. The elements of a list can be of any type — enabling us to use lists of lists to represent matrices for example — and can also be of mixed type, enabling us to represent trees, etc.

Lists can be concatenated using an infix "++" operator and there is also a list difference operator, written "`--`". So for instance

```
[1, 2, 3, 4, 5] -- [1, 3, 5]
```

[1] KRC indexed lists starting from 1 rather than from 0.

has the value `[2, 4]`.

Finally, a useful piece of shorthand is the ".." notation, allowing, for example,

```
[1..100]
```

as a notation for the list of integers from 1 through 100. An interesting property of the implementation in this latter case, by the way, is that this list does not immediately occupy 100 words of store, but only about 3 — enough to store a formula for calculating the elements when they are accessed.

This is part of a general strategy called "lazy evaluation" (Henderson and Morris 1976, Turner 1976) whereby the KRC system consistently avoids performing any calculation until it becomes necessary. Perhaps the most important consequence of this is that it permits the system to accept definitions involving infinite data structures as well as finite ones. For example, the equation

```
x = 2 : x
```

defines x to be the infinite list all of whose elements are 2, and we also permit the form, e.g.

```
[1..]
```

meaning the list of all the natural numbers starting at 1.

Notice by the way that in the applicative style appropriate to a language like KRC, the use of explicit lists of values replaces the use of loops in an imperative language. For example, suppose we wanted to calculate the sum of the numbers from 1 to 1000. We would write

```
sum [1..1000]?
```

in which we first set up the list of values in which we are interested and then apply the library function for summing a list. Lazy evaluation enables us to set up intermediate data structures in this way without incurring a space penalty.

The fourth and final type of object in KRC's universe of discourse is the function. Functions are defined by including in the script one or more equations with the name of the function followed by some formal parameters on the left and an expression describing the corresponding value in the right. For example the factorial function could be defined by the following equation (`product` is a library function)

```
factorial n = product [1..n]
```

Sometimes there are several possible right hand sides — we can show them differentiated by "guards" (a guard is Boolean expression written on the far right of the equation, after a comma). Consider for example the following definition of a function for calculating greatest common divisor by Euclid's algorithm

```
gcd a b = a,           a=b
        = gcd (a-b) b, a>b
        = gcd a (b-a), b>a
```

An alternative to the use of guards on the right is the use of pattern matching on the left, in which one or more of the formal parameters are replaced by constants — an example of this is shown by the definition of Ackermann's function given in Fig. 2 above.

A more sophisticated type of pattern matching involves the use of list structures in formal parameter positions, for example in the following definition of a function which takes a pair of 2–lists representing complex numbers and returns their complex product

```
mult [a,b] [c,d] = [a*c - b*d, a*d + b*c]
```

The operator "`:`" is also allowed in pattern matching, as in this definition of the library function `sum`

```
sum [] = 0
sum (a:x) = a + sum x
```

Here `[]` represents the empty list and in the second equation, the formal parameter matches any non–empty list, whose first member corresponds to `a` and whose rest or "tail" to `x`.

The KRC "Prelude" contains definitions of many useful functions and the reader is recommended to study it for more examples of programming in the same style.

Notice by the way that because of the absence of side effects, KRC functions are purely static in nature — applied to the same argument, a given function always gives the same answer. Note also that functions are "first class citizens" — they can be made elements of lists, passed as parameters and returned as results.

3.1 Partial Application and Higher Order Functions

A particularly powerful kind of abstraction is a higher order function — that is, a function that returns another function as a result. In KRC, these are made available through the following very simple mechanism. If a function is to defined to have, say, n arguments, it can always be applied to less than n arguments, say m, and the result is a function of $(n - m)$ arguments in which the first m arguments have been "frozen in".

So for example, if we define

```
twice f x = f (f x)
sq x = x * x
f = twice sq
```

what is the effect of the function `f?` Answer — it takes the fourth power; `twice` is here being used as a higher order function.

Higher order functions can be used to bring about an extremely compressed programming style. Consider, for example, the definition of `sum` given earlier. This represents an extremely common pattern of recursion for "folding" a list using a given binary operator, in this case "`+`", and a given start value, in this case "`0`". We can capture the general pattern in the following definition of a 3 argument function, `fold`[2]

```
fold op s [] = s
fold op s (a:x) = op a (fold op s x)
```

We can now define `sum` in a single line by partially applying `fold`

```
sum = fold '+' 0
```

The advantage of doing things this way is that a great number of analogous functions become available without any further effort. For example

```
product = fold '*' 1
```

[Note the use of single quotes, e.g.'+', to denote an infix operator as a function.]

3.2 The Use of ZF Expressions

It is extremely useful to be able to quantify over a collection of objects without having to explicitly recurse down them. To this end, KRC includes a facility based on Zermelo–Frankel set abstraction. The basic idea of this kind of set abstraction is that one is allowed to write[3]

$$\{fx \mid x \in S\}$$

meaning "the set of all f x where x is a member of S". Notice that x is a local variable of the above construction. We make this into a KRC language feature by working with lists, rather than sets and using "`<-`" for the membership sign and "`;`" instead of the vertical bar — the bar being already in use to mean logical "or". So for example

```
{x * x; x <- [1..100]}
```

is a list of the squares of the first hundred numbers. The variable binding construct to the right of the semicolon is called a "generator". We further extend the power of the notation by allowing more than one generator, separated by semicolons, and also one or more Boolean expressions, further restricting the range of the variables if required.

So we can define the Cartesian product of two lists (a list of all pairs formed with a member from each) as follows

```
cp x y = {[a,b]; a<-x; b<-y}
```

[2] This function would now be called *foldr*; *foldr* and *foldl* with their now familiar definitions first appeared in the 1983 release of SASL.

[3] ZF expressions are now called "list comprehensions", a term coined by Phil Wadler in 1985. The ZF expressions of 1981 KRC differ from modern list comprehensions in one significant respect — outputs of multiple generators are interleaved to ensure that all combinations are reached, even with two or more infinite generators.

and a list of all Pythagorean triangles with sides less than 30 can be written

```
{[a,b,c]; a,b,c<-[1..30]; a*a+b*b=c*c}
```

notice that in an obvious piece of shorthand, we allow more than one variable to be bound by the same generator.

ZF expressions are particularly powerful when combined with recursion, as in the following definition of a function for generating all the subsets of a given set (here represented as a list) — note that `append` is a prelude function which joins a list of lists to form a single list.

```
subsets [] = [[]]
subsets (a:x) = append {[y,a:y]; y<-subsets x}
```

Another example is this definition of a function for generating all permutations of a list

```
perms [] = [[]]
perms x = {a:p; a<-x; p<-perms (x -- [a])}
```

Generators come into scope from left to right, so later generators can involve earlier ones, but not vice versa. This more or less completes our survey of KRC (it is not a very large language) and we now turn to some programming examples.

4 Some Programming Examples

4.1 Primes by the Method of Eratosthenes

Our first example is generating prime numbers by the sieve of Eratosthenes ("%" is the remainder operator)

```
primes = sieve [2..]
sieve (p:x) = p : sieve {n; n<-x; n%p>0}
```

Given the above as the script the command

```
primes?
```

will cause the KRC system to print prime numbers indefinitely (or rather until it runs out of space) producing the output

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 ... etc.]
```

It is instructive to compare this definition of the sieve of Eratosthenes using infinite lists, with a more operational one involving explicit notions of co–operating process — see (Kuo, Linck and Saadat 1978) for a solution in Hoare's CSP notation.

4.2 The Digits of e

Another example which demonstrates nicely the convenience of being able to work with infinite lists is the following program for printing the digits of e, for the idea of which I am grateful to E. W. Dijkstra. We know that e can be defined, writing $i!$ for $factorial(i)$

$$e = \sum_{i=0}^{\infty} 1/i!$$

Now we can choose to represent fractional numbers using a peculiar base system, in which the weight of the i'th digit is 1/i! — so note that the "carry factor" from the i'th digit back to the (i–1)'th is i. Written to this funny base e is just

$$2.1111111\ldots$$

the problem now being to convert from the funny base to decimal. The general algorithm for converting from any base to decimal can be stated in words as follows.

First print the integer part of the number, then take the remaining digits, multiply them all by 10 and renormalize using the appropriate carry factors — the new integer part will be the next decimal digit and this process can be repeated indefinitely. We can capture this in a purely functional way by representing the number before and after conversion as two infinite lists, and defining a function `convert` recursively, thus

```
e = convert (2 : ones)
ones = 1 : ones
convert (d:x) = d : convert (normalise 2 (0 : mult x))
mult x = {10 * a; a<-x}
normalise c (d:x) = carry c (d : normalise (c+1) x)
carry c (d:e:x) = d+e/c : e%c : x
```

The above will not quite do, however — if we try to print `e`, we get the first digit 2, followed by a long silence. The problem is that the recursion for `normalise` is not well founded — it tries to look infinitely far to the right before producing the first digit. We need some result which limits the distance from which a carry can propagate, or else we are stuck.

The necessary cut–off rule is provided by the observation that in the above conversion, the maximum possible carry from a digit to its immediately leftward neighbour is 9 (we leave the proof of this as an exercise for the reader). This leads us to rewrite the definition of `normalise` in the following more cautious form

```
normalise c (d:e:x) = d : normalise (c+1) (e : x), e+9 < c
                    = carry c (d : normalise (c+1)(e : x))
```

This simple modification is all that is required. If we now issue the command

```
e!
```

the system responds by printing the digits of e indefinitely (subject to space limitations)

```
2.7182818284590 ...
```

Note on KRC Printing Conventions. The use of “!” rather than “?” causes lists to be printed unformatted, without square brackets or commas. This is useful in case the user wants to organise his own formatting by including layout characters at various points in the data structure being printed.

The reader should study the function `show` defined in the KRC prelude — the instruction “`x?`” is actually equivalent to “`show x!`”

4.3 The Eight Queens Problem

For our final example of functional programming, we shall take the well known eight queens problem, which is representative of a large class of problems which, at least when programmed imperatively, seem to require the use of backtracking.

We have to find a way of placing eight queens on a chess board so that no queen is in check from any other. Queens can give check vertically, horizontally or diagonally (in two ways). A moment’s reflection tells us that in any solution, there must be exactly one queen in each column. So an obvious way to proceed is to start with an empty board and proceed from left to right, say, placing one queen in each column, always putting the new queen in a position where it cannot be checked by those already there — and if there is no such position, we have boxed ourselves into a blind alley. A reasonable representation of a board is as a list of integers, giving the row numbers of the queens so far placed on it. So for example

```
[2,5,3]
```

represents a board with queens in the first three columns at the positions shown. So the empty board is represented just by the empty list, `[]`.

We proceed by defining a function `queens n` that returns a list of all the solutions to the “n queens” problem — that is the problem of placing n queens on an n by 8 board

```
queens 0 = [[]]
queens n = {b++[q]; q<-[1..8]; b<-queens(n-1); safe q b}

safe q b = and {\checks q b i; i<-[1..#b]}
checks q b i = q = b i | abs (q - b i) = #b - i + 1
```

This is everything that is needed, to print the solutions, we can say

```
layn (queens 8)!
```

which will cause them to be printed one per line, with numbered lines (`abs`, `and`, `layn` are defined in the KRC prelude).

Notice by the way that if we decide to print only the first solution, e.g. by saying

```
hd (queens 8)?
```

then because of lazy evaluation, the other solutions do not even get generated. So we would still program in the above way, even if we only wanted one solution. The key abstraction that enables us to get rid of the whole problem of backtracking (here and in all similar cases) is to think in terms of a function that returns all the solutions at a given level, instead of only one of them.

Acknowledgements. This paper was first published in Darlington et al. (1982) and I am grateful to CUP for permission to include it in this collection dedicated to Phil Wadler.

The Appendix which follows includes most of the KRC prelude as it stood in 1981. Many of the definitions — hd, tl, filter, map, product, sum, take, drop, and so on, will be familiar to contemporary functional programmers but it may be of historical interest.

A working implementation of Kent Recursive Calculator, including both the 1981 prelude and a more modern prelude, can found at http://krc-lang.org — I am most grateful to Martin Guy for translating my code from an obsolete dialect of BCPL to C and getting it working.

Appendix - The KRC Prelude of Standard Definitions

In the following note that entries of the form

```
name :- explanatory text ;
```

are comments and that infix "." *is functional composition.*

```
abs x = x, x >= 0
      = -x

and = fold '&' "TRUE"

append = fold '++' []

char :- predicate, "TRUE" on strings of length one, "FALSE"
        otherwise (defined in machine code);

cjustify n x = cjustify' (n - printwidth x) x
cjustify' n x = [spaces (n / 2),x,spaces ((n + 1) / 2)]

code :- converts a character to its ascii code number
        (defined in machine code);

compose = fold '.' I

concat :- takes a list of strings and concatenates them to
          make one string (defined in machine code);
```

```
cons a x = a : x

decode :- converts an integer to the character of that ascii
          number (defined in machine code);

digit x = char x & "0" <= x <= "9"

digitval x = code x - code "0", digit x

drop 0 x = x
drop n [] = []
drop n (a:x) = drop (n - 1) x

even x = x % 2 = 0

explode :- explodes a string into a list of its constituent
           characters (defined in machine code);

filter f x = {a; a<-x; f a}

fold op s [] = s
fold op s (a:x) = op a (fold op s x)

for a b f = map f [a..b]

function :- type testing predicate (defined in machine code);

hd (a:x) = a

I x = x

insert :- auxiliary function used by sort, inserts a
          number into a sorted list in the correct position;
insert a [] = [a]
insert a (b:x) = a : b : x, a <= b
               = b : insert a x

interleave :- merges two lists into one by taking members from
              them alternately;

interleave (a:x) y = a : interleave y x
interleave [] y = y

intersection [x] = x
intersection (x:xx) = filter (member x) (intersection xx)
```

```
lay [] = []
lay (a:x) = show a : nl : lay x

layn x = layn' 1 x
layn' n [] = []
layn' n (a:x) = rjustify 4 n:") ":show a:nl:layn' (n + 1) x

letter x = uppercase x | lowercase x

list :- type testing predicate (defined in machine code);

listdiff :- defines the action of the "--" operator;
listdiff [] y = []
listdiff x [] = x
listdiff (a:x) (b:y) = listdiff x y, a = b
                     = listdiff (a : listdiff x [b]) y

ljustify n x = [x,spaces (n - printwidth x)]

lowercase x = char x & code "a" <= code x <= code "z"

map f x = {f a; a<-x}

max [a] = a
max [a,b] = a, a >= b
          = b
max (a:x) = max [a,max x]

member [] a = "FALSE"
member (a:x) b = a = b | member x b

min = hd . sort

mkset x = mkset' x []
mkset' [] y = []
mkset' (a:x) y = mkset' x y, member y a
               = a : mkset' x (a:y)

neg x = -x

nl = decode 10

not x = \x

np = decode 12
```

```
number :- type testing predicate (defined in machine code);

odd x = \even x

or = fold '|' "FALSE"

perms [] = [[]]
perms x = {a : p; a<-x; p<-perms (x -- [a])}

printwidth :- for any x, gives width of x on printing with '!'
              (defined in machine code);

product = fold '*' 1

quote = decode 34

read :- takes a file or device name and returns a list of
        characters (defined in machine code);

reverse [] = []
reverse (a:x) = reverse x ++ [a]

rjustify n x = [spaces (n - printwidth x),x]

show x = "nl", x = nl
       = "np", x = np
       = "tab", x = tab
       = "vt", x = vt
       = [quote,x,quote], string x
       = ["[",show' x,"]"], list x
       = x
show' [] = []
show' [a] = [show a]
show' (a:x) = show a : "," : show' x

sort [] = []
sort (a:x) = insert a (sort x)

spaces 0 = []
spaces n = "" : spaces (n - 1)

string :- type testing predicate (defined in machine code);

sum = fold '+' 0

tab = decode 9
```

```
take 0 x = []
take n [] = []
take n (a:x) = a : take (n - 1) x

tl (a:x) = x

union = mkset . append

uppercase x = char x & code "A" <= code x <= code "Z"

vt = decode 11

write :- marks data to be sent to a file on output with !, thus
            write "filename" x
        where x is any KRC data item (defined in machine code);

zip x = [], hd x = []
      = map hd x : zip (map tl x)
```

References

Ackermann, W.B., Dennis, J.B.: VAL - preliminary reference manual. MIT Laboratory for Computer Science, June 1979

Arvind, Gostelow, K.P., Plouffe, W.: An asynchronous programming language and computing machine. University of California at Irvine, December 1978

Backus, J.: Can Programming be liberated from the Von Neumann style: a functional style and its algebra of programs. CACM **21**(8), 613–641 (1978)

Clarke, J.W., Gladstone, P.J.S., Maclean, C.D., Norman, A.C.: SKIM - S, K, I reduction machine. In: Proceedings LISP Conference, Stanford (1980)

Darlington, J., Henderson, P., Turner, D.A. (eds.): Functional Programming and Its Applications. Cambridge University Press, Cambridge (1982)

Dennis, J.B.: The varieties of data flow computers. MIT Computation Structures Group, Memo 183, August 1979

Floyd, R.W.: Assigning meanings to programs. Proc. Am. Math. Soc. Symp. Appl. Math. **19**, 19–31 (1967)

Henderson, P., Morris, J.M.: A lazy evaluator. In: Proceedings 3rd POPL Symposium, Atlanta, Georgia (1976)

Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12**(10), 567–583 (1969)

Holloway, J., Steele, G., Sussman, G.J., Bell, A.: The scheme 79 Chip. In: Proceedings LISP Conference, Stanford (1980)

Kuo, S.S., Linck, M.H., Saadat, S.: A guide to CSP. Oxford University Programming Research Group, Technical Monograph PRG-14, August 1978

McCarthy, J., et al.: LISP 1.5 Programmers Manual. MIT Press, Cambridge (1962)

Quine, W.V.O.: Word and Object. MIT Press, Cambridge (1960)

Russell, B., Whitehead, A.N.: Principia Mathematica. Cambridge University Press, Cambridge (1925)

Treleaven, P.C.: Exploiting program concurrency in computing systems. IEEE Computer, pp. 42–50, January 1979

Turner, D.A.: SASL Language Manual. St Andrews University Department of Computational Science Technical report (1976)

Turner, D.A.: Aspects of the Implementation of Programming Languages. Oxford University D. Phil. thesis (1981)

Turner, D.A.: Recursion equations as a programming language. In: Darlington et al., pp. 1–28. Cambridge University Press, Cambridge (1982)

Author Index