Jeremy Gibbons
Pablo Nogueira (Eds.)

LNCS 7342

Mathematics of Program Construction

11th International Conference, MPC 2012
Madrid, Spain, June 2012
Proceedings

Springer

Lecture Notes in Computer Science 7342

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Jeremy Gibbons   Pablo Nogueira (Eds.)

Mathematics of Program Construction

11th International Conference, MPC 2012
Madrid, Spain, June 25-27, 2012
Proceedings

Springer

Volume Editors

Jeremy Gibbons
Oxford University
Department of Computer Science
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
E-mail: jeremy.gibbons@cs.ox.ac.uk

Pablo Nogueira
Universidad Politécnica de Madrid
Facultad de Informática, Campus de Montegancedo s/n
28660 Boadilla del Monte, Madrid, Spain
E-mail: pablo.nogueira@upm.es

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31112-3 e-ISBN 978-3-642-31113-0
DOI 10.1007/978-3-642-31113-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939357

CR Subject Classification (1998): F.3, D.2.4, D.1.1, F.4.1, D.3, F.4, G.2, D.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2012
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of MPC 2012, the 11th International Conference on the Mathematics of Program Construction. This conference series aims to promote the development of mathematical principles and techniques that are demonstrably practical and effective in the process of constructing computer programs, broadly interpreted. The focus is on techniques that combine precision with conciseness, enabling programs to be constructed by formal calculation.

The conference was held in Madrid, Spain, during June 25–27, 2012. The previous ten conferences were held in 1989 in Twente, The Netherlands (with proceedings published as LNCS 375); in 1992 in Oxford, UK (LNCS 669); in 1995 in Kloster Irsee, Germany (LNCS 947); in 1998 in Marstrand, Sweden (LNCS 1422); in 2000 in Ponte de Lima, Portugal (LNCS 1837); in 2002 in Dagstuhl, Germany (LNCS 2386); in 2004, in Stirling, UK (LNCS 3125); in 2006 in Kuressaare, Estonia (LNCS 4014); in 2008 in Marseille-Luminy, France (LNCS 5133); and in 2010 in Lac-Beauport, Canada (LNCS 6120).

There were 27 submissions—rather fewer than in previous years. Each submission was reviewed by at least four members of the Program Committee, with an additional review by one of the Program Chairs. The Program Committee selected 13 papers to appear at the conference. Of these 13 papers, 6 had an additional round of ‘shepherding’ by a member of the Program Committee in order to improve the presentation and tailor it for the MPC audience. There were also three invited talks at the conference; these are represented here by one paper and two abstracts.

The MPC conference series takes great pride in the thoroughness of its reviewing. We are very grateful to the members of the Program Committee and the external referees for their care and diligence in reviewing the submitted papers. The review process and compilation of the proceedings were greatly helped by Andrei Voronkov’s EasyChair system, which we can highly recommend.

June 2012

Jeremy Gibbons
Pablo Nogueira

Organization

Program Committee

Ralph-Johan Back	Abo Akademi University, Finland
Roland Backhouse	University of Nottingham, UK
Eerke Boiten	University of Kent, UK
William R. Cook	University of Texas at Austin, USA
Jules Desharnais	Université Laval, Canada
Jeremy Gibbons	University of Oxford, UK
Lindsay Groves	Victoria University of Wellington, New Zealand
Ian J. Hayes	University of Queensland, Australia
Ralf Hinze	University of Oxford, UK
Graham Hutton	University of Nottingham, UK
Johan Jeuring	Open Universiteit Nederland and Universiteit Utrecht, The Netherlands
Christian Lengauer	University of Passau, Germany
Larissa Meinicke	University of Queensland, Australia
Carroll Morgan	University of New South Wales, Australia
Shin-Cheng Mu	Academia Sinica, Taiwan
Bernhard Möller	Institut für Informatik, Universität Augsburg, Germany
David Naumann	Stevens Institute of Technology, USA
Pablo Nogueira	Universidad Politécnica de Madrid, Spain
Jose Oliveira	Universidade do Minho, Portugal
Steve Reeves	The University of Waikato, New Zealand
Wouter Swierstra	Universiteit Utrecht, The Netherlands
Anya Tafliovich	University of Toronto Scarborough, Canada

Additional Reviewers

Paulo Sérgio Almeida	Roland Glueck	Nicolas Pouillard
Gilles Barthe	Stefan Hallerstede	Viorel Preoteasa
James Chapman	Ángel Herranz	Patrick Roocks
Juan Manuel Crespo	Wim Hesselink	Cesar Sanchez
Sharon Curtis	Martin Hofmann	Jeremy Siek
Han-Hing Dang	Peter Höfner	Ana Sokolova
Brijesh Dongol	Daniel James	Kim Solin
Steve Dunne	Mauro Jaskelioff	Tarmo Uustalu
Jonathan Edwards	Björn Lisper	Nicolas Wu
Joao Ferreira	Bruno Oliveira	Andreas Zelend

Local Organization

Pablo Nogueira (Chair)	Universidad Politécnica de Madrid
Ricardo Peña	Universidad Complutense de Madrid
Álvaro García Pérez	IMDEA Software Institute and Universidad Politécnica de Madrid
Manuel Montenegro	Universidad Complutense de Madrid

Host Institutions

- Universidad Complutense de Madrid.
- Universidad Politécnica de Madrid.

Acknowledgements

We are grateful to the Madrid Convention Bureau for their help and support in the organization of the conference. We are also grateful to the Spanish *Ministerio de Economía y Competitividad* for their financial support via Acción Complementaria TIN2011-16141-E.

Table of Contents

Invited Talks

Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs ... 1
Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin

The Laws of Programming Unify Process Calculi ... 7
Tony Hoare and Stephan van Staden

The Geometry of Synthesis: How to Make Hardware Out of Software (Abstract) ... 23
Dan R. Ghica

Security and Information Flow

Scheduler-Independent Declassification ... 25
Alexander Lux, Heiko Mantel, and Matthias Perner

Elementary Probability Theory in the Eindhoven Style ... 48
Carroll Morgan

Synchronous and Real-Time Systems

Scheduling and Buffer Sizing of n-Synchronous Systems: Typing of Ultimately Periodic Clocks in Lucy-n ... 74
Louis Mandel and Florence Plateau

Deriving Real-Time Action Systems Controllers from Multiscale System Specifications ... 102
Brijesh Dongol and Ian J. Hayes

Algorithms and Games

Calculating Graph Algorithms for Dominance and Shortest Path ... 132
Ilya Sergey, Jan Midtgaard, and Dave Clarke

First-Past-the-Post Games ... 157
Roland Backhouse

Program Calculi

Reverse Exchange for Concurrency and Local Reasoning 177
Han-Hing Dang and Bernhard Möller

Unifying Correctness Statements 198
Walter Guttmann

Tool Support

Dependently Typed Programming Based on Automated Theorem Proving 220
Alasdair Armstrong, Simon Foster, and Georg Struth

Algebras and Datatypes

An Algebraic Calculus of Database Preferences 241
Bernhard Möller, Patrick Roocks, and Markus Endres

Modular Tree Automata 263
Patrick Bahr

Categorical Functional Programming

Constructing Applicative Functors 300
Ross Paterson

Kan Extensions for Program Optimisation *Or:* Art and Dan Explain an Old Trick 324
Ralf Hinze

Author Index 363

Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs

Gilles Barthe[1], Benjamin Grégoire[2], and Santiago Zanella Béguelin[3]

[1] IMDEA Software Institute
[2] INRIA Sophia Antipolis - Méditerranée
[3] Microsoft Research

Provable security. The goal of provable security is to verify rigorously the security of cryptographic systems. A provable security argument proceeds in three steps:

1. Define a security goal and an adversarial model;
2. Define the cryptographic system and the security assumptions upon which the security of the system hinges;
3. Show by reduction that any attack against the cryptographic system can be used to build an efficient algorithm that breaks a security assumption.

The provable security paradigm originates from the work of Goldwasser and Micali [10] and plays a central role in modern cryptography. Since its inception, the focus of provable security has gradually shifted towards practice-oriented provable security [4]. The central goal of practice-oriented provable security is to develop and analyze efficient cryptographic systems that can be used for practical purposes, and to provide concrete guarantees that quantify their strength as a function of the values of their parameters (e.g. the key size of a public-key encryption scheme).

The code-based approach [5] realizes the practice-oriented provable security paradigm by means of programming-language techniques and of a systematic way of organizing proofs. In the code-based approach, security hypotheses and goals are cast in terms of the probability of events with respect to distributions induced by probabilistic programs. Typically, proofs that follow the code-based approach adopt some form of imperative pseudocode as a convenient and expressive notation to represent programs (equivalently, games). The pWHILE language is a procedural, probabilistic imperative programming language that provides a precise formalism for programs. Commands in pWHILE are defined as follows:

$$
\begin{array}{rll}
\mathcal{C} ::= & \mathsf{skip} & \text{nop} \\
| & \mathcal{V} \leftarrow \mathcal{E} & \text{assignment} \\
| & \mathcal{V} \xleftarrow{\$} \mathcal{D}\mathcal{E} & \text{random sampling} \\
| & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
| & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{while loop} \\
| & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
| & \mathcal{C};\ \mathcal{C} & \text{sequence}
\end{array}
$$

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 1–6, 2012.
© Springer-Verlag Berlin Heidelberg 2012

where $\mathcal{E}$ is a set of expressions, $\mathcal{DE}$ is a set of distribution expressions, and $\mathcal{P}$ is a set of procedures. pWHILE distinguishes between concrete procedures, whose code is defined, and abstract procedures, whose code remains unspecified. Quantification over adversaries in cryptographic proofs is achieved by representing them as abstract procedures parametrized by a set of oracles; these oracles must be instantiated as other procedures in the program.

The meaning of games is defined by a denotational semantics: given an interpretation of abstract procedures, the semantics $[\![c]\!]$ of a game c takes as input an initial memory, i.e. a mapping from variable to values, and returns a sub-distribution on memories. Since we typically consider discrete datatypes, the set $\mathcal{M}$ of memories is discrete and sub-distributions on memories are simply maps $d : \mathcal{M} \to [0,1]$ such that $\sum_{m \in \mathcal{M}} d\ m \leq 1$. We let $\mathcal{D}(\mathcal{M})$ denote the set of sub-distributions over $\mathcal{M}$, and we use the notation $\Pr[c, m : E]$ to denote the probability of the event E in the sub-distribution $[\![c]\!]\ m$.

Proofs in provable security are by reduction. For simplicity, assume that the system under consideration is proved secure under a single assumption. Let $\mathcal{A}$ be an adversary against the security of the system. The goal of the reduction proof is to show that there exists an adversary $\mathcal{B}$ such that the success probability of $\mathcal{A}$ in the attack game is upper bounded by a function of the success probability of $\mathcal{B}$ in breaking the security assumption. A recommended practice is that proofs be constructive, in the sense that the adversary $\mathcal{B}$ is given explicitly as a program that invokes $\mathcal{A}$ as a sub-procedure.

In addition to defining the security goal and hypotheses as probabilistic programs, the code-based approach recommends that proofs are structured as sequences, or trees, of games, so that transitions between two successive games are easier to justify. A typical transition between two games c_1 and c_2, requires establishing an inequality of the form

$$\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B] + \epsilon \qquad (*)$$

where A and B are events whose probabilities are taken over the sub-distributions $[\![c_1]\!]\ m_1$ and $[\![c_2]\!]\ m_2$ respectively, and ϵ is an arithmetic expression that may depend on the resources allocated to the adversary or, when the transition involves a failure event F, an expression of the form $\Pr[c_i, m_i : F]$. The proof concludes by combining the inequalities proven for each transition to bound the success probability of the reduction.

Verified security. Verified security [2,1] is an emerging approach to security proofs of cryptographic systems. It adheres to the same principles as (practice-oriented) provable security, but revisits its realization from a formal verification perspective. When taking a verified security approach, proofs are mechanically built and verified with the help of state-of-the-art verification tools. The idea of verified security appears in inspiring articles from Halevi [11] and Bellare and Rogaway [5], and is realized by tools such that CertiCrypt [2] and EasyCrypt [1]. Both support the code-based approach, and capture many common reasoning patterns in cryptographic proofs. CertiCrypt and EasyCrypt have been used to

verify examples of prominent cryptographic constructions, including encryption schemes, signature schemes, hash function designs, and zero-knowledge proofs. Although they rely on the same foundations, the two tools have a complementary design: the CertiCrypt framework is entirely programmed and machine-checked in the Coq proof assistant, from which it inherits expressiveness and strong guarantees. In contrast, EasyCrypt implements a verification condition generator that sends proof obligations to SMT solvers, and inherits from them a high degree of automation.

Relational Hoare Logics and liftings. It is important to note that inequalities of the form (∗) involve two programs, and hence go beyond program verification. The common foundation of CertiCrypt and EasyCrypt is pRHL, a relational logic to reason about probabilistic programs. Its starting point is relational Hoare logic [6], a variant of Hoare logic that reasons about two programs. Judgments of Benton's relational Hoare logic are of the form:

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

where c_1 and c_2 are WHILE programs and Ψ and Φ are relations on memories, respectively called the pre-condition and the post-condition. The above judgment is valid if the post-condition is valid for all executions of c_1 and c_2 starting from initial memories that satisfy the pre-condition, i.e. for every pair of initial memories m_1, m_2 such that $m_1 \ \Psi \ m_2$, if the evaluations of c_1 in m_1 and c_2 in m_2 terminate with final memories m_1' and m_2' respectively, then $m_1' \ \Phi \ m_2'$ holds.

Probabilistic relational Hoare logic (pRHL) considers similar judgments

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

except that c_1 and c_2 are pWHILE programs. Since the evaluation of a pWHILE program w.r.t. an initial memory yields a sub-distribution over memories, giving a meaning to a pRHL judgment requires interpreting post-conditions as relations over sub-distributions. To this end, pRHL relies on a lifting operator $\mathcal{L}$ which transforms a binary relation into a binary relation on the space of sub-distributions over its underlying sets. Lifting can be used to define the validity of a pRHL judgment: for any two pWHILE programs c_1 and c_2 and relations on memories Ψ and Φ, the judgment $\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$ is valid if for every pair of memories m_1 and m_2, $m_1 \ \Psi \ m_2$ implies $(\llbracket c_1 \rrbracket \ m_1) \, \mathcal{L}(\Phi) \, (\llbracket c_2 \rrbracket \ m_2)$. The ability to derive probability claims from valid pRHL judgments is essential to justify its use as an intermediate tool to prove security of cryptographic systems. Formally, if $\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$ and $\Phi \Rightarrow A\langle 1 \rangle \Rightarrow B\langle 2 \rangle$, then for all memories m_1 and m_2, $m_1 \, \Psi \, m_2$ implies $\Pr[c_1, m_1 : A] \leq \Pr[c_2, m_2 : B]$.

The definition of lifting is adopted from probabilistic process algebra [12]. Let A and B be two discrete sets, and let $R \subseteq A \times B$. The lifting $\mathcal{L}(R)$ of R is the relation on $\mathcal{D}(A) \times \mathcal{D}(B)$ such that, for every sub-distribution d_1 over A and d_2 over B, $d_1 \, \mathcal{L}(R) \, d_2$ if there exists $d \in \mathcal{D}(A \times B)$ such that:

1. for every $(a, b) \in A \times B$, if $d(a, b) > 0$ then $a \ R \ b$
2. for every $a \in A$, $d_1(a) = \sum_{b \in B} d(a, b)$
3. for every $b \in B$, $d_2(a) = \sum_{a \in A} d(a, b)$

The definition of lifting has close connections with the Kantorovich metric (see e.g. [7] for a recent overview), and with flow networks. The relationship with flow networks is best explained pictorially. Figure 1 represents two sub-distributions d_1 (over some set A) and d_2 (over some set B) as a source and a sink respectively. In both cases, the capacity of an edge between an element of the set and the source/sink is the probability of this element; we let p_i denote $d_1(a_i)$ and q_i denote $d_2(b_i)$. Then, let R be a relation between elements of A and elements of B. We use dashed arrows to represent edges (a, b) such that $a\ R\ b$. The definition of lifting requires exhibiting a sub-distribution d, called the witness, such that for every (a, b), if $d(a, b) > 0$ then $a\ R\ b$. In the picture below, it amounts to asserting that d is completely defined by the values of the probabilities $r_1 \ldots r_6$ that are used to decorate the dashed arrows; more precisely, $r_6 = d(a_5, b_5)$, $r_5 = d(a_5, b_4)$, $r_4 = d(a_3, b_4)$, $r_3 = d(a_3, b_3)$, $r_2 = d(a_2, b_2)$ and $r_1 = d(a_1, b_1)$. The remaining constraints can be interpreted as an assertion that d is a maximal flow. Consider that edges are oriented from left to right; then we can define the incoming (resp. outgoing) flow of a node as the sum of the probabilities attached to its incoming (resp. outgoing) edges. Then, constraints 2 and 3 assert that the incoming flow is equal to the outcoming flow for each node; in other words, $d_1\,\mathcal{L}(R)\,d_2$ can be reduced to a maximal flow problem in the network induced by d_1, d_2 and R. The constraints for the maximal flow problem are:

$$\begin{array}{ll}
p_5 = r_5 + r_6 & q_5 = r_6 \\
 & q_4 = r_4 + r_5 \\
p_3 = r_3 + r_4 & q_3 = r_3 \\
p_2 = r_2 & q_2 = r_2 \\
p_1 = r_1 & q_1 = r_1
\end{array}$$

Any sub-distribution that satisfies the constraints is a witness of the relationship of d_1 and d_2 w.r.t. the lifting of R. Interestingly, the connection between lifting and flow networks opens the possibility of relying on existing flow network algorithms to check whether two distributions are related by lifting.

While pRHL is sufficient for many purposes, a number of cryptographic notions, such as statistical zero-knowledge proofs, require reasoning about approximate equivalence. Recall that the statistical distance between two distributions d_1 and d_2 is defined as

$$\varDelta(d_1, d_2) \stackrel{\text{def}}{=} \max_A |d_1\ A - d_2\ A|$$

One can define an extension of pRHL that supports approximate reasoning, via judgments of the form $\models c_2 \sim_\delta \varPsi : c_1 \Rightarrow \varPhi$, where $\delta \in [0, 1]$. Such a judgment is valid if for every pair of memories m_1 and m_2, $m_1\ \varPsi\ m_2$ implies $(\llbracket c_1 \rrbracket\ m_1)\,\mathcal{L}_\delta(\varPhi)\,(\llbracket c_2 \rrbracket\ m_2)$, where the approximate lifting $\mathcal{L}_\delta(R)$ of R is defined by the clause $d_1\,\mathcal{L}_\delta(R)\,d_2$ if there exists $d \in \mathcal{D}(A \times B)$ such that:

1. for every $(a, b) \in A \times B$, if $d(a, b) > 0$ then $a\ R\ b$
2. $\pi_1(d) \le d_1$ and $\varDelta(d_1, \pi_1(d)) \le \delta$
3. $\pi_2(d) \le d_2$ and $\varDelta(d_2, \pi_2(d)) \le \delta$

where the sub-distributions $\pi_1(d)$ and $\pi_2(d)$ are defined by the clauses

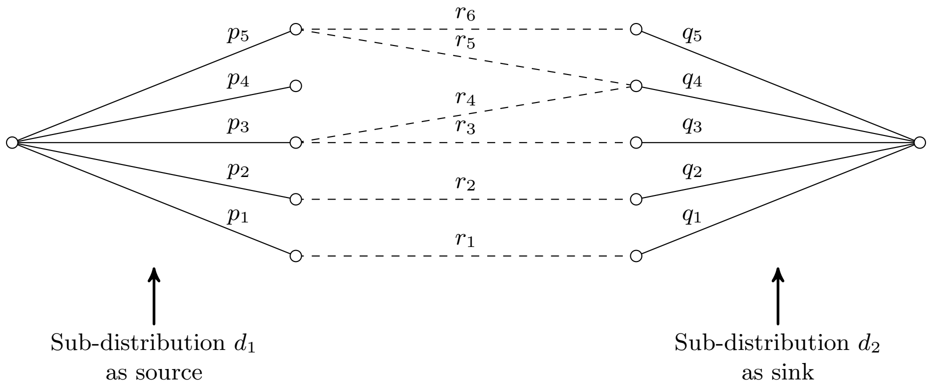


Fig. 1. Lifting as a maximal flow problem

$$\pi_1(d) = \sum_{b \in B} d\ (a, b) \qquad\qquad \pi_2(d) = \sum_{a \in A} d\ (a, b)$$

and $\leq$ denotes the pointwise extension of inequality on the reals, i.e. $d \leq d'$ iff for every $a \in A$, we have $d\ a \leq d'\ a$. The definition of approximate lifting was also considered in the context of probabilistic algebra [13,8], and admits a characterization in terms of flow networks.

CertiCrypt and EasyCrypt implement apRHL [3], an extension of pRHL whose judgments are of the form $\models c_2 \sim_{\alpha,\delta} \Psi : c_1 \Rightarrow \Phi$, where $\alpha \geq 1$ and $\delta \in [0, 1]$. Such a judgment is valid if for every pair of memories m_1 and m_2, $m_1\ \Psi\ m_2$ implies $(\llbracket c_1 \rrbracket\ m_1)\, \mathcal{L}_{\alpha,\delta}(\Phi)\, (\llbracket c_2 \rrbracket\ m_2)$, where $\mathcal{L}_{\alpha,\delta}(R)$ denotes the approximate lifting of R. Formally, $d_1\, \mathcal{L}_{\alpha,\delta}(R)\, d_2$ if there exists $d \in \mathcal{D}(A \times B)$ such that:

1. for every $(a, b) \in A \times B$, if $d(a, b) > 0$ then $a\ R\ b$
2. $\pi_1(d) \leq d_1$ and $\Delta_\alpha(d_1, \pi_1(d)) \leq \delta$
3. $\pi_2(d) \leq d_2$ and $\Delta_\alpha(d_2, \pi_2(d)) \leq \delta$

where

$$\Delta_\alpha(d_1, d_2) \stackrel{\text{def}}{=} \max_A(\max\{d_1\ A - \alpha\ (d_2\ A), d_2\ A - \alpha\ (d_1\ A), 0\})$$

This definition coincides with δ-lifting for the case $\alpha = 1$. The resulting logic apRHL allows reasoning about statistical distance between programs and about (computational) differential privacy [9].

References

1. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)

2. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 90–101. ACM, New York (2009)
3. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic reasoning for differential privacy. In: 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012. ACM (2012)
4. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: 1st ACM Conference on Computer and Communications Security, CCS 1993, pp. 62–73. ACM, New York (1993)
5. Bellare, M., Rogaway, P.: The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 14–25. ACM, New York (2004)
7. Deng, Y., Du, W.: Logical, metric, and algorithmic characterisations of probabilistic bisimulation. Technical Report CMU-CS-11-110, Carnegie Mellon University (March 2011)
8. Desharnais, J., Laviolette, F., Tracol, M.: Approximate analysis of probabilistic processes: Logic, simulation and games. In: 5th International Conference on Quantitative Evaluation of Systems, QEST 2008, pp. 264–273. IEEE Computer Society (2008)
9. Dwork, C.: Differential Privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)
10. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. 28(2), 270–299 (1984)
11. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005)
12. Jonsson, B., Yi, W., Larsen, K.G.: Probabilistic extensions of process algebras. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier, Amsterdam (2001)
13. Segala, R., Turrini, A.: Approximated computationally bounded simulation relations for probabilistic automata. In: 20th IEEE Computer Security Foundations Symposium, CSF 2007, pp. 140–156 (2007)

The Laws of Programming Unify Process Calculi*

Tony Hoare[1] and Stephan van Staden[2]

[1] Microsoft Research, Cambridge, United Kingdom
[2] ETH Zurich, Switzerland
Stephan.vanStaden@inf.ethz.ch

Abstract. We survey the well-known algebraic laws of sequential programming, and propose some less familiar laws for concurrent programming. On the basis of these laws, we derive the rules of a number of classical programming and process calculi, for example, those due to Hoare, Milner, and Kahn. The algebra is simpler than each of the calculi derived from it, and stronger than all the calculi put together. We end with a section describing the role of unification in Science and Engineering.

1 Introduction

The basic ideas and content of the algebraic laws of sequential programming are familiar [1]. That paper treated the main program structuring operators, including sequential composition, choice, and recursion. This paper introduces additional laws to deal with conjunction and concurrent execution of programs. The formulation is purely algebraic and lacks negative statements, so all the earlier axioms and theorems survive the introduction of additional laws.

The unification of process calculi is based on the earlier unification of programs, program designs, and program specifications. We regard them all as descriptions of the events that occur in and around a computer that is executing a program. The program itself is the most precise description of its own execution. The most abstract description is the user specification, which mentions only aspects of execution that are observable and controllable by the user. An assertion can be regarded as a description of all executions that end in a state satisfying the assertion. Each kind of description has a different role in program development and execution, and they are usually expressed in different notations. But we ignore the distinctions between them, because they all obey the same algebraic laws.

The main novel content of the paper is a unifying treatment of a varied collection of programming calculi, which have been proposed as formalisations of the meaning of sequential and concurrent programming languages. They have been successfully applied in human and mechanical reasoning about the properties of programs expressed in the given calculus. Examples of such calculi are due to

* This paper is dedicated in friendship and admiration to the memory of Robin Milner and Gilles Kahn.

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 7–22, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Hoare [2], Milner [3], Kahn [4], Dijkstra [5], Back [6], Morgan [7], Plotkin [8], and Jones [9]: in this paper, we shall concentrate on the first three.

The method of unification is to define the basic judgement of each calculus in terms of one or more inequations between purely algebraic formulae and then prove all the rules of the calculus as theorems of the algebra. The algebra turns out to be simpler than each of the calculi derived from it, and stronger than all of them put together.

We make no claims that our algebraic laws are actually true of the operators of any particular programming language. We rely on the good will of the readers to check the individual laws against their intuitive understanding and experience of the essential concepts of programming. The demonstration that these properties are true of many historic programming calculi gives some independent evidence that the algebra is potentially useful, and that it corresponds to a widely held common understanding of the meaning of programs.

For a survey of omissions and deficiencies in the content and presentation of this paper, please see the sections on related work in the past and on further work for the future.

All theorems of this paper have been formally checked with Isabelle/HOL. A proof script is available online [10].

2 Laws of Programming

Programs, specifications and designs together form the set of descriptions that we consider. These descriptions are ordered according to refinement: $P \subseteq Q$ indicates that P refines Q. This refinement has several meanings. For example, it can say that the program P is more determinate than program Q (i.e. P has fewer behaviours or executions) or that the specification P is stronger than the specification Q (i.e. P implies Q). Generally, a description is more abstract or general compared to the descriptions that refine it, and the refined description is more deterministic. Refinement obeys three laws that make it a partial order:

- $P \subseteq P$
- $P \subseteq Q \quad \& \quad Q \subseteq R \quad \Rightarrow \quad P \subseteq R$
- $P \subseteq Q \quad \& \quad Q \subseteq P \quad \Rightarrow \quad P = Q$

Among the descriptions, there are three constants taken from programming languages and propositional logic: *skip*, $\perp$ and $\top$. The constant *skip* is a basic program that does nothing. Bottom $\perp$ represents the predicate False: it describes no execution. Bottom is the meaning of a program containing a fault like a syntax violation: the implementation is required to detect it, and to prevent the program from running. Top $\top$ is a program whose execution may have unbounded consequences. Think of a program with an error (e.g., input buffer overflow) that makes it vulnerable to virus attack. As a proposition, it can be identified with the predicate True. It is the programmer's responsibility to avoid submitting such a program for execution – the implementation is not required to detect it.

Apart from the constant descriptions, there are operators for forming descriptions in terms of others. The operators are likewise drawn from programming

languages and propositional logic. For example, sequential composition (;) and concurrent composition ($\|$) are binary operators from programming: the formula $P\,;Q$ describes the sequential composition of P and Q, while $P \parallel Q$ is a description of their concurrent behaviour.

Conjunction ($\wedge$) and disjunction ($\vee$) are operators familiar from propositional logic. If P and Q are programs, $P \vee Q$ is the nondeterministic choice between the components P and Q. The choice may be determined at some later stage in the design trajectory of the program, or by a specified condition tested at run time; failing this, it may be determined by an implementation of the language at compile time, or even nondeterministically at run time. It satisfies the following laws:

- $P \subseteq P \vee Q$ and $Q \subseteq P \vee Q$.
- Whenever $P \subseteq R$ and $Q \subseteq R$, then $P \vee Q \subseteq R$.

These laws say that ($\vee$) is the least upper bound with respect to the refinement order. Conjunction is its dual and corresponds to the greatest lower bound. Conjunction between programs is in general very inefficient to implement, because it may require exploration of all the executions of both operands, to find one that is common to both of them. Nevertheless, it is the most natural and useful way of composing a specification from a collection of requirements.

The logical operators satisfy familiar algebraic laws. For instance, conjunction and disjunction are both commutative and associative. Programming operators also enjoy similar algebraic properties. For example, saying that $(P\,;Q)\,;R$ and $P\,;(Q\,;R)$ describe the same set of computations is the same as stating that sequential composition is associative. The properties of most of the operators considered here are described and intuitively justified in [1]. Table 1 summarizes how the binary operators behave in isolation from each other.

Table 1. Basic properties of the operators

	$\vee$	$\wedge$	;	$\|$
Commutative	yes	yes	no	yes
Associative	yes	yes	yes	yes
Idempotent	yes	yes	no	no
Unit	$\bot$	$\top$	*skip*	*skip*
Zero	$\top$	$\bot$	$\bot$	$\bot$

In addition to such laws, distribution laws state the relationships between two (or more) operators. All the binary operators in the table distribute through ($\vee$), i.e. for $\circ \in \{\vee, \wedge, ;, \|\}$ we have:

- $P \circ (Q \vee R) = (P \circ Q) \vee (P \circ R)$
- $(P \vee Q) \circ R = (P \circ R) \vee (Q \circ R)$

Another distribution law, analogous to the exchange law of category theory, specifies how sequential and concurrent composition interact:

- $(P \parallel Q)\,;(R \parallel S) \subseteq (P\,;R) \parallel (Q\,;S)$

This is a form of mutual distribution between the two operators, where different components of each operand distribute through to different components of the other operand. The refinement in the law reflects the fact that concurrency introduces nondeterminism, whereas sequential composition does not. It says that the program $(P \parallel Q) ; (R \parallel S)$ has fewer behaviours than $(P ; R) \parallel (Q ; S)$.

But is the law in fact true of implementations of concurrency in real computers and in usable programming languages? Yes, it is true for all implementations which interleave the actions from the constituent threads, or which are sequentially consistent, in that they successfully simulate such an interleaving. Here is an informal proof. The right-hand side of the inclusion describes all interleavings of an execution of $(P ; R)$ with an execution of $(Q ; S)$. The left hand side describes all interleavings which synchronise at the two semicolons displayed in $(P ; R)$ and $(Q ; S)$. Thus the left hand side contains $<p_1, q, p_2, r_1, s, r_2>$, but it does not contain $<p_1, q, s, p_2, r_1, r_2>$, which is an interleaving of the right side (here the lower case letters denote sub-executions of the executions of the corresponding upper case programs).

The validity of the law can be exploited in an algorithm to compute one (or all) of the interleavings of two strings. If one of the arguments is empty, deliver the other argument as result. Otherwise, split each string arbitrarily into two parts $P ; R$ and $Q ; S$. Then (recursively) find an interleaving of P with Q and an interleaving of R with S. Concatenate the two results.

Iteration is another common programming construct. This unary operator is typically written as a postfix Kleene star: P^* describes the iteration where P is performed zero or more times in sequence. Iteration interacts with the other operators according to laws from Kleene algebra [11]:

- $\mathit{skip} \vee (P ; P^*) \subseteq P^*$
- $P \vee (Q ; R) \subseteq R \quad \Rightarrow \quad Q^* ; P \subseteq R$
- $\mathit{skip} \vee (P^* ; P) \subseteq P^*$
- $P \vee (R ; Q) \subseteq R \quad \Rightarrow \quad P ; Q^* \subseteq R$

The first law says that P^* has more behaviours than *skip*, and more behaviours than $P ; P^*$. A valid implementation of an iteration can therefore start by unfolding it into two cases, one of which does no iterations, and the other of which does at least one iteration. The second law implies that iteration is the least solution of the first inequation. It permits inductive proofs of the properties of an iteration. The other two laws simply swap the arguments of (;).

2.1 Lemmas

The laws mentioned so far are already sufficient to imply the central axioms of various calculi of programming. The proofs use a number of simple lemmas that follow from the algebra.

A binary operator that distributes through $(\vee)$ is monotone in both arguments. So for $\circ \in \{\vee, \wedge, ;, \|\}$:

(∘Monotone) $$P \subseteq P' \quad \& \quad Q \subseteq Q' \quad \Rightarrow \quad P \circ Q \subseteq P' \circ Q'$$

Also, the Exchange law has several consequences that can be proved as theorems with the help of the properties in Table 1. In particular:

- Two small exchange laws hold:

  (SmallExchange$_1$) $$P\,;(Q \parallel R) \subseteq (P\,;Q) \parallel R$$
  (SmallExchange$_2$) $$(P \parallel Q)\,;R \subseteq P \parallel (Q\,;R)$$

- Sequential composition refines concurrent composition, since it is a special case thereof:

  (SeqRefinesConc) $$P\,;Q \subseteq P \parallel Q$$

Although exchange is a less familiar form of distribution law, there are also other cases where operators exchange. For example,

(ConjExchange) $$(P \wedge Q)\,;(R \wedge S) \subseteq (P\,;R) \wedge (Q\,;S)$$

The same theorem holds when $(;)$ is replaced by $(\|)$ or any other monotonic operator. The dual property, where $(\vee)$ replaces $(\wedge)$ and the refinement order is reversed, also holds.

An interesting property of all our algebraic laws is shared with many of the fundamental laws of physics: they preserve the symmetry of time-reversal. Formally expressed, each law remains valid when sequential composition is replaced by backward sequential composition $(\breve{;})$, defined

$$P \,\breve{;}\, Q \;\stackrel{\mathbf{def}}{=}\; Q\,;P$$

As a consequence, every theorem of our algebra also respects time-reversal: swapping the arguments of every $(;)$ in a theorem yields another theorem. Swapping the arguments of $(;)$ once again will result in the original theorem, so time-reversal is a duality.

Of course, there are useful and realistic laws that do not respect time-reversal. For example, if *abort* stands for a program that never terminates and never has any interaction with its environment, it could realistically be stated to satisfy:

$$P \neq \bot \quad \Rightarrow \quad abort\,;P = abort$$

Such a law could be added to our algebra, but it would not respect time-reversal.

The algebra embodies other algebraic structures used in computer science. For example, if $\mathbb{D}$ is the set of descriptions, then:

- $(\mathbb{D}, \vee, ;, ^*, \bot, \mathit{skip})$ is a Kleene algebra;
- $(\mathbb{D}, \vee, ;, \bot, \mathit{skip})$ is an idempotent semiring;
- $(\mathbb{D}, \vee, \wedge, \bot, \top)$ is a bounded lattice.

3 Calculi of Programming

We consider three calculi of programming: Hoare logic, the Milner calculus and Kahn's natural semantics. Each of them uses a basic judgement with three operands. Surprisingly, the judgement of Hoare logic and the Milner calculus turns out to be the same. A calculus selects one of its judgement's operands as the inductive variable, and the semantics of the language is then defined by cases on the syntactic structure of a term in this position. Thus there is at least one rule for every constant and operator of the language. Each calculus may restrict some operands of its judgement to a subclass of expression. For example, two operands of the Hoare triple are descriptions of a state of the executing machine. The elements of a subclass may satisfy additional algebraic properties that are important for a calculus. Such distinctions may be introduced if and when necessary, and are not needed in this paper.

3.1 Hoare Logic

The purpose of Hoare's axiomatic approach to computer programming [2] is to establish partial or total correctness of computer programs. The basic judgement of Hoare logic[1] is the Hoare triple $P\,\{Q\}\,R$:

$$P\,\{Q\}\,R \;\stackrel{\mathbf{def}}{=}\; P\,;Q \subseteq R$$

It says that if P is a description of what has happened before Q starts, then R describes what has happened when Q has finished. In conventional presentations of Hoare logic, the variables P and R are required to be predicates describing a single state of the executing computer before and after the execution of Q respectively. This is just a special case of our more general definition, because a single-state predicate may be regarded as a description of all executions which leave the computer in a state that satisfies the predicate when they terminate. Such an interpretation preserves the intuitive meaning of the triple: in a state that satisfies the description P, every behaviour of Q conforms to the description R. The more usual interpretation of a predicate in the relational model of Hoare logic is as a subset of the identity relation [12]. The primitive judgement $P\,\{Q\}\,R$ of Hoare logic is then defined as $P\,;Q \subseteq Q\,;R$. However, a similar definition is not possible for the Milner calculus, which does not distinguish assertions from processes.

The basic laws of Hoare logic are defined by structural induction on Q. The rule (Hconj′) follows directly from lemma (ConjExchange). Floyd's original law for conjunction (Hconj) follows from it by idempotence of $(\wedge)$.

[1] Also called *axiomatic* or *deductive* semantics.

(Hskip) $P\,\{skip\}\,P$

(Hseq) $P\,\{Q\}\,R \quad \& \quad R\,\{Q'\}\,S \quad \Rightarrow \quad P\,\{Q\,;Q'\}\,S$

(Hchoice) $P\,\{Q\}\,R \quad \& \quad P\,\{Q'\}\,R \quad \Rightarrow \quad P\,\{Q \vee Q'\}\,R$

(Hconj′) $P\,\{Q\}\,R \quad \& \quad P'\,\{Q'\}\,R' \quad \Rightarrow \quad P \wedge P'\,\{Q \wedge Q'\}\,R \wedge R'$

(Hiter) $P\,\{Q\}\,P \quad \Rightarrow \quad P\,\{Q^*\}\,P$

Hoare logic also includes rules that operate only on the structure of assertions, such as the rule of consequence and rules for conjunction and disjunction. They too follow from the algebra:

(Hcons) $P' \subseteq P \quad \& \quad P\,\{Q\}\,R \quad \& \quad R \subseteq R' \quad \Rightarrow \quad P'\,\{Q\}\,R'$

(Hconj) $P\,\{Q\}\,R \quad \& \quad P'\,\{Q\}\,R' \quad \Rightarrow \quad P \wedge P'\,\{Q\}\,R \wedge R'$

(Hdisj) $P\,\{Q\}\,R \quad \& \quad P'\,\{Q\}\,R' \quad \Rightarrow \quad P \vee P'\,\{Q\}\,R \vee R'$

The Hoare rules that govern concurrent composition are formulated in separation logic [13,14].

(Hconc) $P\,\{Q\}\,R \quad \& \quad P'\,\{Q'\}\,R' \quad \Rightarrow \quad P \parallel P'\,\{Q \parallel Q'\}\,R \parallel R'$

(Hframe) $P\,\{Q\}\,R \quad \Rightarrow \quad F \parallel P\,\{Q\}\,F \parallel R$

The first of these laws permits a modular proof of a concurrent composition. All that is necessary is to prove each component of the composition separately, with separate preconditions and postconditions; and then the precondition of the composition is the composition of the preconditions of the components, and the postcondition is formed similarly.

The second law is the frame law which lies at the foundation of separation logic. It says that every Hoare triple can be validly strengthened by concurrent composition of its precondition and postcondition with the same assertion F. It was surprising to discover that the frame law is still valid when $(\parallel)$ is interpreted as interleaving. Less surprising is the fact that the same frame law also applies to sequential composition, but only in one direction:

(HseqFrame) $P\,\{Q\}\,R \quad \Rightarrow \quad F\,;P\,\{Q\}\,F\,;R$

Because of time-reversal symmetry, every law of the Hoare calculus gives rise to another law. The collection of these slightly permuted laws gives a new calculus, in which the variable chosen for induction is the first element of the triple rather than the second.

There is a third calculus based on the same judgement $P\,;Q \subseteq R$, where the variable chosen for the induction is the third parameter R. This variation will be treated next.

3.2 The Milner Calculus

The purpose of the Milner calculus is to demonstrate an abstract implementation of a programming language, and thereby provide guidance on its practical implementations.

This style of operational semantics was introduced by Milner for CCS in [3], and has been used to specify the operational meaning of other process calculi. It adopts the Milner transition $P \xrightarrow{Q} R$ as fundamental judgement.

$$P \xrightarrow{Q} R \stackrel{\mathbf{def}}{=} P \supseteq Q \,;\, R$$

The judgement $P \xrightarrow{Q} R$ says that one possible way of executing P is to execute Q first and then to execute R. All arguments of the triple are conventionally programs, and Q is usually confined to be a basic or primitive command, executed as a single action (in a 'small-step' version of the semantics).

The underlying algebra has revealed that the Hoare and the Milner calculi differ only in the ordering of the three parameters of their basic judgement. Hence every law of the Hoare calculus can be translated into a law of the Milner calculus, and vice-versa, using the equivalence $P\,\{Q\}\,R \iff R \xrightarrow{P} Q$. The change of order reflects the fact that in a deductive semantics, the proof starts with the more refined operand, whereas actual execution starts with the less refined operand. This is because execution of the specified step may require the implementation to make a nondeterministic choice of options available in the original program.

The Milner calculus chooses the variable P for induction, and has several laws that hold as theorems. A judgement $P \xrightarrow{Q} \mathit{skip}$ says that P can be completely executed by doing Q, since it remains to do nothing (*skip*). This is sometimes written as $P \xrightarrow{Q} \surd$ in process calculi. Hence the Milner rule for executing a basic action is:

(Maction) $$P \xrightarrow{P} \mathit{skip}$$

The execution of a sequential composition begins with an execution of its first component. The rule is similar to (HseqFrame), but framing happens to the right:

(Mseq$_1$) $$P \xrightarrow{Q} R \quad\Rightarrow\quad P\,;P' \xrightarrow{Q} R\,;P'$$

When R is *skip* in the above law, i.e. execution of the first component is finished, then it remains to execute the second component. Simplifying $\mathit{skip}\,;P'$ yields the rule:

(Mseq$_2$) $$P \xrightarrow{Q} \mathit{skip} \quad\Rightarrow\quad P\,;P' \xrightarrow{Q} P'$$

CCS uses prefixing – a restricted form of sequential composition in which the first operand must be a basic action. Combining (Maction) and (Mseq$_2$) gives Milner's axiom:

(Mprefixing) $$P\,;P' \xrightarrow{P} P'$$

A judgement $P \xrightarrow{\mathit{skip}} R$ says that without doing real work, the program P can be validly rearranged/rewritten as R, which is then executed instead. The notation

$P \longrightarrow R$ abbreviates $P \xrightarrow{skip} R$, and is equivalent to $P \supseteq R$. Hence rearrangement is simply refinement – a reduction in the program's nondeterminism. Unsurprisingly, a common rearrangement is the resolution of a nondeterministic choice:

(Mchoice) $$P \vee P' \longrightarrow P$$

The rule for choosing the second operand follows by the commutativity of $(\vee)$.

An iteration can be rearranged or unfolded into two cases, one which does no iterations, and the other which does at least one iteration:

(Miter$_1$) $$P^* \longrightarrow skip$$

(Miter$_2$) $$P^* \longrightarrow P\,;P^*$$

The concurrency law expresses the interleaving of actions from the operands. It corresponds to (Hframe):

(Mconc$_1$) $$P \xrightarrow{Q} R \quad \Rightarrow \quad P \parallel P' \xrightarrow{Q} R \parallel P'$$

In the case where R is *skip*, we can simplify $skip \parallel P'$ and get:

(Mconc$_2$) $$P \xrightarrow{Q} skip \quad \Rightarrow \quad P \parallel P' \xrightarrow{Q} P'$$

The rules for executing the second operand of a concurrent composition follow by the commutativity of $(\parallel)$.

The Milner calculus has a counterpart of (Hseq):

(Mseq) $$P \xrightarrow{Q} R \quad \& \quad R \xrightarrow{Q'} S \quad \Rightarrow \quad P \xrightarrow{Q\,;Q'} S$$

This law is typically omitted from small-step calculi, since the action $Q\,;Q'$ in its consequent is not guaranteed to be a basic one. However, (Mseq) can combine two rearrangements into a single one or combine a basic action with a rearrangement. Consequently, small-step Milner-style calculi have some freedom with respect to the choice of rules. For example, by combining a rearrangement with a basic action, it is simple to derive alternative rules for nondeterministic choice:

(Mchoice$'$) $$P \xrightarrow{Q} R \quad \Rightarrow \quad P \vee P' \xrightarrow{Q} R$$

Finally, the rule (Mcons) corresponds to (Hcons):

(Mcons) $$P \longrightarrow P' \quad \& \quad P' \xrightarrow{Q} R \quad \& \quad R \longrightarrow R' \quad \Rightarrow \quad P \xrightarrow{Q} R'$$

3.3 Natural Semantics

The natural semantics[2] of Kahn [4] demonstrates an abstract implementation of a programming language by showing how a program manipulates computational

[2] Also called *evaluation* or *big-step* semantics.

states. The fundamental judgement of natural semantics is the Kahn reduction $\langle P,\, s\rangle \longrightarrow s'$. It is not just a re-ordering of the parameters of the two earlier judgements, because it contains the semicolon on the right of refinement rather than on the left:

$$\langle P,\, s\rangle \longrightarrow s' \;\stackrel{\mathbf{def}}{=}\; s\,;P \supseteq s'$$

The judgement $\langle P,\, s\rangle \longrightarrow s'$ says that if s describes the state before execution of P, then s' describes a possible final state. The s describes a single input and s' a single output state in conventional presentations, but there is no need to impose such restrictions. For example, we can think about a state as recording an execution history, or more generally, a set of possible execution histories. Then $s\,;P$ describes the full set of possible execution histories that result from executing P in state s. A state s' describes a possible final state of the execution when it is contained in $s\,;P$, i.e. every execution history that s' describes as possible is indeed possible.

The rules of natural semantics are based on the structure of P. They follow as theorems from the algebraic properties of the operators involved.

(Kskip) $\quad \langle \mathit{skip},\, s\rangle \longrightarrow s$

(Kseq) $\quad \langle P,\, s\rangle \longrightarrow s' \;\;\&\;\; \langle P',\, s'\rangle \longrightarrow s'' \;\;\Rightarrow\;\; \langle P\,;P',\, s\rangle \longrightarrow s''$

(Kchoice) $\quad \langle P,\, s\rangle \longrightarrow s' \;\;\Rightarrow\;\; \langle P \vee P',\, s\rangle \longrightarrow s'$

The omitted rule for executing the second operand of a nondeterministic choice follows from the commutativity of $(\vee)$.

Iteration is specified with two laws:

(Kiter$_1$) $\quad \langle P^*,\, s\rangle \longrightarrow s$

(Kiter$_2$) $\quad \langle P,\, s\rangle \longrightarrow s' \;\;\&\;\; \langle P^*,\, s'\rangle \longrightarrow s'' \;\;\Rightarrow\;\; \langle P^*,\, s\rangle \longrightarrow s''$

For concurrency:

(Kconc) $\quad \langle P,\, s\rangle \longrightarrow s' \;\;\&\;\; \langle P',\, s'\rangle \longrightarrow s'' \;\;\Rightarrow\;\; \langle P \parallel P',\, s\rangle \longrightarrow s''$

There is a similar rule for executing the second operand of $(\parallel)$ first, but the rules are trivial and not very interesting. As Nielson and Nielson remark [15, p. 50], "in a natural semantics the execution of the immediate constituents [of a parallel composition] is an *atomic entity* so we cannot express interleaving of computations". In contrast to this, a small-step semantics like the Milner calculus can easily express interleaving.

4 Related Work

The results reported in the previous sections are a contribution to a much larger endeavour, to which many researchers are making essential and complementary contributions, both published and on-going.

Our collection of algebraic laws is not intended to be complete. It omits nearly all the basic actions of programming, such as assignments, inputs, outputs, assumptions, assertions, allocations, disposals, throws. Many of these are treated in [16]. The technique of concurrent abstract predicates [17] suggests a way that such primitive commands can be defined algebraically, together with the primitive predicates which specify their intended effect.

Pure algebra is not allowed to express the negation of an equation. As a consequence, algebra cannot be used for disproving false conjectures, or for detecting program errors. The solution is to accompany the algebra with a collection of mathematical models, each of which satisfies all its laws. A false conjecture can be shown to be unproveable by finding one of these models (test case) of which the conjecture is not true. Modern mechanical model checkers are good at this, and they are beginning to be used for generating test cases that reveal dangerous programming errors [18]. Familiar models of our programming algebra are Boolean Algebra, regular expressions of Kleene [19], the relational calculus [20], partial orders [21]. Concurrency in these models has been explored by [22].

Each of the above models represents a program as a set of all its possible executions. An execution of a program is a set of events that have occurred in and around a computer that has executed the program. The events may be related to each other by a dependency relation between them. The dependency may be attributed either to control flow [23] or to data flow [16]. For example, the 'executions' of a regular expression are the strings of the relevant regular language. In this case, each event is the occurrence of a character, and the control flow is a total ordering. Sequential composition is modeled by concatenation of strings, and concurrent composition by interleaving. The relational calculus requires that the length of the string is just two. Sequential composition requires that the last element of the first pair is the same as the initial element of the second pair; this element is then omitted from the concatenation. In a partial order model, events that are independent (with no dependency in either direction) are regarded as 'truly' concurrent. Of course, in a particular concrete execution, an implementation may finish one of them before the other one starts, but the model simply ignores this distinction. Partially ordered executions can be illustrated graphically, with events drawn as nodes and dependencies drawn as arrows between them. Such a graph provides a formalisation of the intuition of a programmer who understands programs in terms of their execution.

4.1 Further Work

Our collection of algebraic laws is not intended to be definitive or prescriptive. Individual laws can be omitted, or they can be adapted for particular programming languages or implementations. The consequences of such variations can be explored by algebraic methods. Additional laws can be introduced to simplify reasoning about useful subsets of programs, like race-free programs that rely on the ownership protocol of separation logic. This is the subject of on-going research [24]. And finally, additional axioms can be introduced to describe additional features of programming, such as throws, catches, contracts, transactions.

The particular calculi treated in this paper are all weaker than the algebra from which they are derived. The main source of the weakness is that some of the parameters of the basic judgement are restricted to a subset of the possible operands. For example, the precondition and postcondition of a Hoare triple are required to be assertions, describing a set of machine states. Further restrictions are imposed by selection of rules. For each operator of the calculus, there is often only one law, serving as its 'definition'. In each definition, the operator occurs in a uniform position. Reducing the power of a calculus in these ways potentially makes it simpler to use for its intended purpose, and more efficient in mechanisation. For each calculus, it would be interesting to explore the reason for each restriction adopted, whether practical, philosophical, or merely a historical accident.

In general, a program verification tool will be more efficient if it exploits the characteristics of a particular form of the specification for the program. Since the results of this paper are completely general, there is plenty of scope for research into beneficial specialisations. Here are some of the possibilities.

A specification may require only conditional correctness (if execution is finite, the final state will satisfy a certain predicate), or it may require termination (all executions of the program are finite). It may specify a relation between initial and final state (the final value of an array is a sorted permutation of the initial value). It may specify non-functional properties such as security (e.g. there is no leakage of data from high to low security threads), persistence (no executions terminate), fairness (every thread will eventually make progress), liveness (no infinite internal activity), timing (a response is always given to a request before a certain deadline), and even probability (as execution progresses, the ratio of a-events to b-events tends to unity).

Finally, it is possible to specify important properties of a program that are independent of its application or purpose, for example, conformance to some declared program design pattern or protocol. For most programs, it is highly desirable to rule out given generic errors, like overflows, null references, other exceptions, deadlocks, concurrency races, and space leaks. Absence of generic error is a specification that is the target of modern program analysis systems. They are popular, because they can be applied directly to legacy code that has been written without any other more explicit specification, and even without assertions.

5 Unification in Science and Engineering

In the natural sciences, the quest for a unifying theory is an integral part of the scientific culture. The aim is to show that a single theory applies to a wide range of highly disparate phenomena. For example, the gravitational theory of Newton applies very accurately both to apples falling towards the earth and to planets falling towards the sun. In many cases, a more homogeneous subset of the phenomena is already covered by a more specialised scientific theory. In these cases, the specialised theories must be derived mathematically from the claimed

theory that claims to unify them. For example, Newton's theory of gravitation unifies the elegant planetary theory of Kepler, as well as the less elegant (and less accurate) Ptolemaic theories of astronomy.

The benefit of a unified theory to the progress of science is that it is supported by all the evidence that has already been accumulated for all of the previous theories separately. Furthermore, each of the previous theories then inherits all the extra support given by the total sum of evidence contributed by all the other theories. Unification is a very cost-effective way of approximating more closely the highest certainty that all of science seeks.

Practising engineers have different concerns from the scientist, including deadlines and budgets for their current projects. The engineer will therefore continue to use familiar more specialised theories that have been found from experience to be well adapted to the particular features of the current project, or the needs of the current client. Indeed, the innovative engineer will often specialise the theory even further, adapting it so closely to current needs that there will never be an opportunity for repeated use. In conclusion, the separate theories that are subsumed by a unifying theory often retain all their practical value, and they are in no way belittled or superseded by the unification.

Unification does not depend on the identity of the theorems being unified. Quite often, a specialised application does not require all the laws of the unifying theory. For example, Milner's CCS and other related process calculi do not require the distribution law $P\,;(Q \vee R) = (P\,;Q) \vee (P\,;R)$. Instead, they rely only on the weaker property that (;) is monotone in its second argument. This is because Milner wanted to distinguish processes which differ in the time at which their nondeterminism is resolved. In general, omission of unwanted axioms gives more flexibility in the implementation of an algebra. In other cases, an axiom in one theory may be replaced even by an axiom in another theory that contradicts it. This is valuable too in understanding the conceptual relationships between a family of theories, because the choice of axioms formalises simply and abstractly the nature and scale of the differences between them.

The real practical value of unification lies in its contribution to the transfer of the results of scientific research into engineering practice. One of the main factors that inhibit the engineer (and the sensible manager) from adopting a scientific theory is that scientists do not yet agree what that theory should be. Fortunately, there is an agreed method of resolving a scientific dispute. An experiment is designed whose result is predicted differently by all the theories that are party to the dispute. The engineer can then have increased confidence in the winner.

But sometimes, no such decisive experiment can be discovered. This may be because, in spite of differences in their presentation, the theories are in fact entirely consistent. In this case, the only way of resolving the issue is to find a theory that unifies them all. Quantum theory provides an example. Three separate mathematical presentations of quantum theory were put forward by Heisenberg, Schrödinger and Dirac. Then Dirac showed that they were all derivable from a single unified theory. This is what made possible the award of a

Nobel Prize to all three of them. And quantum theory is now accepted as the nearest to a theory of everything that physics has to offer.

A second potential contribution of a unified theory to the practising engineer is in the design and use of a suite of software tools that assist in automation of the design process. Since every major engineering enterprise today combines a range of technologies, it is important that all the specialised members of the tool suite should be based on a common theory, so that they can communicate consistently among each other on standard interfaces which are based upon the unification. Standardisation has led to a continuing exponential increase in the power of proof tools that support software engineering. It also facilitates competition among the tools, and permits independent evolution of separate tools for joint use in a design automation toolset.

Finally, the education of the general scientist and engineer will surely be facilitated by reducing the number of independently developed theories to a single theory, presented in a single coherent framework and notation. That in itself is sufficient justification for conduct by academics of research into unification of theories.

6 Conclusion

One of the criteria of success of a unifying theory is that it is simpler than each of the theories that can be derived from it. Thus Kepler's theory of elliptical orbits for the planets was simpler than each of the many theories of planetary epicycles which it has now replaced.

In the simpler domain of computer programming, algebra seems simpler than each of the calculi in several respects. (1) Its primitive operators have only two operands instead of the triples of a programming calculus. (2) The basic algebraic properties of these operators can be postulated independently, or (by distribution laws) just a few at a time. (3) The properties (associativity, commutativity, ...) are familiar from many other branches of algebra. (4) The basic rule of deduction is the substitution of equals, or (using monotonicity) the substitution of one member of an ordered pair by another. The calculi include at least one proof rule for each operator, often with two triples as antecedent and one as consequent. Of course, perception of simplicity is necessarily subjective, and the reader will have to make a personal judgement on the validity of these claims.

Another criterion of success of a unifying theory is that it is more powerful than the conjunction of all the theories that can be derived from it. As a result, it is possible to conjecture new scientific discoveries on the basis of the theory. For example, Newton's theory predicted the existence of yet undiscovered planets (Uranus), Einstein's theory predicted atomic weapons, and quantum theory has predicted the existence of new subatomic particles subsequently confirmed by experiment.

It is often possible to derive an algebraic law from a rule of a calculus, by expanding the algebraic definition of the judgement involved (e.g., $P ; Q \subseteq R$). In many cases, this gives the same law that was used to derive the rule from

the algebra. But this equivalent law will always be inequational. If the algebraic law is also inequational (e.g., the exchange law), the rule and the law will be equivalent (in the sense of inter-provability). But in general, the algebraic law will be an equation (for example, an association or a distribution law). In this way, the algebra will be more powerful than each of our calculi.

However, the conjunction of the calculi is still weaker than the laws, for a different reason. It is because each calculus places constraints on the operands of the judgement. The Milner calculus requires one of the three to be a basic action, and the Hoare calculus requires two of the three to describe just a single state. In Hoare logic, the restriction on the two assertional arguments can be mitigated (even circumvented) by allowing the two assertions to share a common 'logical variable', which does not appear anywhere in the program given as the middle argument.

The Milner calculus uses a sophisticated inductive technique (bisimulation) to prove additional laws. It is based on an induction principle, that the given rules are the only way of proving equations. Now additional rules can be derived by induction on the proof of the antecedent. Adding new non-derived rules might invalidate the results, so incrementality is lost.

It is too early to say whether our unification of theories will lead to unexpected new insights or new discoveries. The main benefit may be to researchers into the principles of programming, who will never again have to prove the relative soundness or relative completeness of deductive and operational presentations of the semantics of the same language. A second benefit may be the revelation that concurrency can be treated formally by the same algebraic techniques as sequential composition.

Acknowledgements. For useful comments and suggestions we are grateful to Thomas Dinsdale-Young, Sophia Drossopoulou, Peter O'Hearn, Rasmus Petersen, Andreas Podelski, Vlad Scherbina, Georg Struth, Jim Woodcock, and to the referees of MPC 2012. We are grateful also to other attendants at the meeting of IFIP WG 2.3, held in Winchester 18–23 September, 2011; also the Separation logic Workshop held in Queen Mary University, London, 5–6 October 2011. Van Staden was supported by ETH Research Grant ETH-15 10-1.

References

1. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Commun. ACM 30, 672–686 (1987)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12, 576–580 (1969)
3. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
4. Kahn, G.: Natural Semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)

5. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
6. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer (1998)
7. Morgan, C.: Programming from specifications, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
8. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (September 1981)
9. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (June 1981); printed as: Programming Research Group, Technical Monograph 25
10. Isabelle/HOL proofs (2012), http://se.inf.ethz.ch/people/vanstaden/LawsOfProgrammingUnifyProcessCalculi.thy
11. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. 110, 366–390 (1994)
12. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Logic 1(1), 60–76 (2000)
13. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
14. O'Hearn, P.W.: Resources, Concurrency and Local Reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
15. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction, revised edn. (July 1999), http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html, original edition published 1992 by John Wiley & Sons
16. Hoare, T., Wickerson, J.: Unifying models of data flow. In: Broy, M., Leuxner, C., Hoare, T. (eds.) Proceedings of the 2010 Marktoberdorf Summer School on Software and Systems Safety. IOS Press (2011)
17. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
18. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Queue 10(1), 20:20–20:27 (2012)
19. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata Studies, pp. 3–41. Princeton University Press (1956)
20. Tarski, A.: On the calculus of relations. J. Symb. Log. 6(3), 73–89 (1941)
21. Gischer, J.L.: The equational theory of pomsets. Theor. Comput. Sci. 61, 199–224 (1988)
22. Hoare, C.A.R.T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009)
23. Wehrman, I., Hoare, C.A.R., O'Hearn, P.W.: Graphical models of separation logic. Inf. Process. Lett. 109(17), 1001–1004 (2009)
24. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On Locality and the Exchange Law for Concurrent Processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)

The Geometry of Synthesis
How to Make Hardware Out of Software

Dan R. Ghica

University of Birmingham, UK
d.r.ghica@cs.bham.ac.uk

Abstract. High-level synthesis or "hardware compilation" is a behavioural synthesis method in which circuits are specified using conventional programming languages. Such languages are generally recognised as more accessible than hardware description languages, and it is expected that their use would significantly increase design productivity. The Geometry of Synthesis is a new hardware compilation technique which achieves this goal in a semantic-directed fashion, by noting that functional programming languages and diagrammatic descriptions of hardware share a common mathematical structure, and by using the game-semantic model of the programming language to reduce all computational effects to signal-like message passing. As a consequence, this technique has mature support for higher-order functions [1], local (assignable) state [2], concurrency [3] and (affine) recursion [4]. Moreover, the compiler can support features such as separate compilation, libraries and a foreign-function interface [5]. The programming language of GoS, Verity, is an "Algol-like" language [6] extended with concurrency features [7]. The interplay between the call-by-name function mechanism and local effects, an approach specific to Algol, is the key ingredient which makes it possible for a large class of programs in this language to have finitely representable semantic models which can be synthesised as stand-alone static circuits. The compiler is available as an open-source download. [1]

References

1. Ghica, D.R., Smith, A.: Geometry of Synthesis III: resource management through type inference. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 345–356. ACM (2011)
2. Ghica, D.R.: Geometry of Synthesis: a structured approach to VLSI design. In: Hofmann, M., Felleisen, M. (eds.) POPL, pp. 363–375. ACM (2007)
3. Ghica, D.R., Smith, A.: Geometry of Synthesis II: From games to delay-insensitive circuits. Electr. Notes Theor. Comput. Sci. 265, 301–324 (2010)
4. Ghica, D.R., Smith, A., Singh, S.: Geometry of Synthesis IV: compiling affine recursion into static hardware. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP, pp. 221–233. ACM (2011)
5. Ghica, D.R.: Function interface models for hardware compilation. In: Singh, S., Jobstmann, B., Kishinevsky, M., Brandt, J. (eds.) MEMOCODE, pp. 131–142. IEEE (2011)

[1] http://veritygos.org/

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 23–24, 2012.
© Springer-Verlag Berlin Heidelberg 2012

6. Reynolds, J.C.: The essence of Algol. In: O'Hearn, P.W., Tennent, R.D. (eds.) ALGOL-like Languages, vol. 1, pp. 67–88. Birkhauser Boston Inc., Cambridge (1997)
7. Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. Ann. Pure Appl. Logic 151(2-3), 89–114 (2008)

Scheduler-Independent Declassification

Alexander Lux, Heiko Mantel, and Matthias Perner

Computer Science, TU Darmstadt, Germany
{lux,mantel,perner}@cs.tu-darmstadt.de

Abstract. The controlled declassification of secrets has received much attention in research on information-flow security, though mostly for sequential programming languages. In this article, we aim at guaranteeing the security of concurrent programs. We propose the novel security property WHAT&WHERE that allows one to limit what information may be declassified where in a program. We show that our property provides adequate security guarantees independent of the scheduling algorithm (which is non-trivial due to the refinement paradox) and present a security type system that reliably enforces the property. In a second scheduler-independence result, we show that an earlier proposed security condition is adequate for the same range of schedulers. These are the first scheduler-independence results in the presence of declassification.

1 Introduction

When giving a program access to secrets, one would like to know that the program does not leak them to untrusted sinks. Such a confidentiality requirement can be formalized by information-flow properties like, e.g., *noninterference* [12].

Noninterference-like properties require that a program's output to untrusted sinks is independent of secrets. Such a lack of dependence obviously ensures that public outputs do not reveal any secrets. While being an adequate characterization of confidentiality, the requirement is often too restrictive. The desired functionality of a program might inherently require some correlation between secrets and public output. Examples are password-based authentication mechanisms (a response to an authentication attempt depends on the secret password), encryption algorithms (a cipher-text depends on the secret plain-text), and online stores (electronic goods shall be kept secret until they have been ordered).

Hence, it is necessary to relax noninterference-like properties such that a deliberate release of some secret information becomes possible. While this desire has existed since the early days of research on information-flow control (e.g. in the Bell/La Padula Model secrets can be released by so called trusted processes [8]), solutions for controlling declassification are just about to achieve a satisfactory level of maturity (see [33] for an overview). However, research on declassification has mostly focused on sequential programs so far, while controlling declassification in multi-threaded programs is not yet equally well understood.

Generalizing definitions of information-flow security for sequential programs to security properties that are suitable for concurrent systems is known to be non-trivial. Already in the eighties, Sutherland [34] and McCullough [23] proposed

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 25–47, 2012.
© Springer-Verlag Berlin Heidelberg 2012

noninterference-like properties for distributed systems. These were first steps in a still ongoing exploration of sensible definitions of information-flow security [19]. The information-flow security of multi-threaded programs, on which we focus in this article, is also non-trivial. Due to the refinement paradox [14], the scheduling of threads requires special attention. In particular, it does not suffice to simply assume a possibilistic scheduler, because a program might have secure information-flow if executed with the fictitious possibilistic scheduler, but be insecure if executed, e.g., with a Round-Robin or uniform scheduler.

Our first main contribution is the formal definition of two schemas for noninterference-like properties for multi-threaded programs. Our schemas $\text{WHAT}^{\mathfrak{s}}$ and $\text{WHAT\&WHERE}^{\mathfrak{s}}$ are parametric in a scheduler model $\mathfrak{s}$. Both schemas can be used to capture confidentiality requirements, but they differ in how declassification is controlled. If the scheduler is known then $\mathfrak{s}$ can be specified concretely and, after instantiating one of our schemas with $\mathfrak{s}$, one obtains a property that adequately captures information-flow security for this scheduler.

However, often the concrete scheduler is not known in advance. While, in principle, one could leave the scheduler parametric and use, e.g., $\forall\mathfrak{s}.\text{WHAT}^{\mathfrak{s}}$ as security condition, such a universal quantification over all possible schedulers is rather inconvenient, in program analysis as well as in program construction. Fortunately, an explicit universal quantification over schedulers can be avoided.

Our second main contribution is the definition of a novel security condition WHAT&WHERE and a scheduler-independence result, which shows that WHAT&WHERE implies $\text{WHAT\&WHERE}^{\mathfrak{s}}$ for all possible scheduler models $\mathfrak{s}$. A compositionality result shows that our novel property is compatible with compositional reasoning about security. Based on this result, we derive a security type system for verifying our novel security property efficiently.

Our third main contribution is a scheduler-independence result showing that our previously proposed property WHAT_1 [20] implies $\text{WHAT}^{\mathfrak{s}}$ for all $\mathfrak{s}$.

Previous scheduler-independence results were limited to information-flow properties that forbid declassification (e.g. [31,36,22]). With this article, we close this gap by developing the first scheduler-independence results for information-flow properties that support controlled declassification. Scheduler independence provides the basis for verifying security without knowing the scheduler under which a program will be run. Our scheduler-independence results also reduce the conceptual complexity of constructing secure programs. They free the developer from having to consider concrete schedulers when reasoning about security.

Proofs of all theorems in this article are available on the authors' web-pages.

2 Preliminaries

2.1 Multi-threaded Programs

Multi-threaded programs perform computations in concurrent threads that can communicate with each other, e.g. via shared memory. When the number of threads exceeds the number of available processing units, scheduling becomes necessary. Usually, the schedule for running threads is determined dynamically

at run-time based on previous scheduling decisions and on observations about the current configuration, such as the number of currently active threads.

In this article, we focus on multi-threaded programs that run on a single-core CPU with a shared memory for inter-thread communication. In this section, we present our model of program execution (a small-step operational semantics), our model of scheduler decisions (a labeled transition system), and an integration of these two models. The resulting system model is similar to the one in [22].

Semantics of Commands and Expressions. We assume a set of *commands* $\mathcal{C}$, a set of *expressions* $\mathcal{E}$, a set of *program variables* $\mathcal{Var}$, and a set of *values* $\mathcal{Val}$. We leave these sets underspecified, but give example instantiations in Section 2.2.

We define the set of *memory states* by the function space $\mathcal{Mem} = \mathcal{Var} \rightarrow \mathcal{Val}$. A function $m \in \mathcal{Mem}$ models which values are currently stored in the program variables. We define the set of *program states* by $\mathcal{C}_\epsilon = \mathcal{C} \cup \{\epsilon\}$. A program state from $\mathcal{C}$ models which part of the program remains to be executed while the special symbol ϵ models termination. We define the set of *thread pools* by $\mathcal{C}^*$ (i.e. the set of finite lists of commands). Each command in a thread pool is the program state of an individual thread in a multi-threaded program. We refer to threads by their position $k \in \mathbb{N}_0$ in a thread pool $thr \in \mathcal{C}^*$. If a thread is uniquely determined by $thr[k]$, i.e. the command at position k, then we sometimes refer to the thread by this command. We define $\#(thr)$ to equal the number of threads in the thread pool $thr \in \mathcal{C}^*$. The list $\langle c_0, c_1, \ldots, c_{n-1} \rangle$ with $c_0, c_1, \ldots, c_{n-1} \in \mathcal{C}$ models a thread pool with n threads. The list $\langle\rangle$ models the empty thread pool. Note that the symbol ϵ does not appear in thread pools.

We model *evaluation of expressions* by the function $eval : \mathcal{E} \times \mathcal{Mem} \rightarrow \mathcal{Val}$, where $eval(e, m)$ equals the value to which $e \in \mathcal{E}$ evaluates in $m \in \mathcal{Mem}$.

We model *execution steps* by judgments of the form $\langle\!| c_1, m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c_2, m_2 |\!\rangle$ where $c_1 \in \mathcal{C}$, $c_2 \in \mathcal{C}_\epsilon$, $m_1, m_2 \in \mathcal{Mem}$, and $\alpha \in \mathcal{C}^*$. Intuitively, this judgment models that a command c_1 is executed in a memory state m_1 resulting in a program state c_2 and a memory state m_2. The label $\alpha \in \mathcal{C}^*$ carries information about threads spawned by the execution step. If the execution step does not spawn new threads then $\alpha = \langle\rangle$ holds, otherwise we have $\alpha = \langle c_0, c_1, \ldots, c_{n-1} \rangle$ where $c_0, c_1, \ldots, c_{n-1} \in \mathcal{C}$ are the threads spawned in this order.

We assume deterministic commands, i.e. for each $c_1 \in \mathcal{C}$ and $m_1 \in \mathcal{Mem}$, there exists exactly one tuple $(\alpha, c_2, m_2) \in \mathcal{C}^* \times \mathcal{C}_\epsilon \times \mathcal{Mem}$ such that $\langle\!| c_1, m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c_2, m_2 |\!\rangle$ is derivable. As an alternative notation for the effect of a command on the memory, we define the function $[\![\bullet]\!] : \mathcal{C} \rightarrow (\mathcal{Mem} \rightarrow \mathcal{Mem})$ by $[\![c_1]\!](m_1) = m_2$ iff $\exists c_2 \in \mathcal{C}_\epsilon.\ \exists \alpha \in \mathcal{C}^*.\ \langle\!| c_1, m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c_2, m_2 |\!\rangle$.

As a notational convention, we use $v \in \mathcal{Val}$ to denote values, $x \in \mathcal{Var}$ to denote variables, $m \in \mathcal{Mem}$ to denote memory states, $c \in \mathcal{C}_\epsilon$ to denote program states, $e \in \mathcal{E}$ to denote expressions, $thr \in \mathcal{C}^*$ to denote thread pools, and $k \in \mathbb{N}_0$ to denote positions of threads.

Scheduler Model. We present a parametric scheduler model that can be instantiated for a wide range of schedulers. For modeling the behavior of schedulers, we use labeled transition systems as described below.

We assume a set of *scheduler states* $\mathcal{S}$ and a set of possible *scheduler inputs* In. Scheduler states model the memory of a scheduler and scheduler inputs model the input to the scheduler by the environment. We leave the set In underspecified, but require that any $in \in \mathit{In}$ reveals at least the number of active threads in the current thread pool and denote this number by $\#(in)$.

We define the set of *scheduler decisions* by $\mathcal{D}ec = \mathit{In} \times \mathbb{N}_0 \times [0;1]$. Intuitively, a scheduler decision $(in, k, p) \in \mathcal{D}ec$ models that the scheduler selects the kth thread with the probability p given the scheduler input in. The special case $p = 1$ models a deterministic decision.

Definition 1. *A* scheduler model $\mathfrak{s}$ *is a labeled transition system* $(\mathcal{S}, s_0, \mathcal{D}ec, \rightarrow)$*, where* $\mathcal{S}$ *is a set of scheduler states,* $s_0 \in \mathcal{S}$ *is an initial state,* $\mathcal{D}ec$ *is the set of scheduler decisions, and* $\rightarrow \subseteq \mathcal{S} \times \mathcal{D}ec \times \mathcal{S}$ *is a transition relation such that:*

1. $\forall (s_1, (in, k, p), s_2) \in \rightarrow .\ (k < \#(in) \wedge p \neq 0)$
2. $\forall s_1 \in \mathcal{S}.\ \forall in \in \mathit{In}.\ \Big(\#(in) > 0 \Longrightarrow \Big(\sum_{(s_1,(in,k,p),s_2) \in \rightarrow} p\Big) = 1\Big)$
3. $\forall s_1, s_2, s_2' \in \mathcal{S}.\ \forall in \in \mathit{In}.\ \forall k \in \mathbb{N}_0.\ \forall p, p' \in]0;1].$
   $(((s_1, (in, k, p), s_2) \in \rightarrow) \wedge ((s_1, (in, k, p'), s_2') \in \rightarrow) \Longrightarrow p = p' \wedge s_2 = s_2')$

For a scheduler model $\mathfrak{s}$*, we write* $(s_1, in) \overset{k}{\rightsquigarrow}{}^{\mathfrak{s}}_{p}\ s_2$ *iff* $(s_1, (in, k, p), s_2) \in \rightarrow$.

Conditions 1 and 2 ensure that a scheduler model definitely selects some thread from the current thread pool. Condition 3 ensures that the probability of a scheduler decision and the resulting scheduler state are uniquely determined by the original scheduler state, the scheduler input, and the selected thread.

Our notion of scheduler models is suitable for expressing a wide range of schedulers, including Round-Robin schedulers as well as uniform schedulers.

For simplicity of presentation we consider only scheduler models without redundant states. Formally, we define the bisimilarity of scheduler states coinductively by a symmetric relation $\sim = \mathcal{S} \times \mathcal{S}$ that is the largest relation such that for all $dec \in \mathcal{D}ec$ and for all $s_1, s_1', s_2 \in \mathcal{S}$, if $s_1 \sim s_1'$ and $(s_1, dec, s_2) \in \rightarrow$ then there exists a scheduler state $s_2' \in \mathcal{S}$ with $(s_1', dec, s_2') \in \rightarrow$ and $s_2 \sim s_2'$. We require that the equivalence classes of $\sim$ are singleton sets, i.e. $\forall s, s' \in \mathcal{S}.\ (s \sim s' \Longrightarrow s = s')$, which means that there are no redundant states. Note that any given scheduler model can be transformed into one that satisfies this constraint by using the equivalence classes of $\sim$ as scheduler states.

As a notational convention, we use $in \in \mathit{In}$ to denote scheduler inputs, $p \in [0;1]$ to denote probabilities, and $s \in \mathcal{S}$ to denote scheduler states. For brevity, we often write *scheduler* instead of scheduler model.

Integration into a System Model. We now present the system model which defines the interaction between threads and a scheduler.

We define the set of *observation functions* by the function space $\mathcal{O}bs = (\mathcal{C}^* \times \mathcal{M}em) \rightarrow \mathit{In}$. A function $obs \in \mathcal{O}bs$ models the input to a scheduler for a given thread pool and memory state. We define the set of *system configurations* by $\mathcal{C}nf = \mathcal{C}^* \times \mathcal{M}em \times \mathcal{S}$. Intuitively, a system configuration $\langle\!| thr, m, s |\!\rangle \in \mathcal{C}nf$ models the current state of a multi-threaded program in a run-time environment.

We model *system steps* by judgments of the form $cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2$, where $cnf_1, cnf_2 \in \mathcal{C}nf$ and $(k, p) \in \mathbb{N}_0 \times]0; 1]$. Intuitively, this judgment models that, in system configuration cnf_1, the scheduler selects the kth thread with probability p and that this results in cnf_2. We define the rule for deriving this judgment by:

$$[\text{SysStep}]\frac{(s_1, in) \overset{k}{\rightsquigarrow}^{\mathfrak{s}}_{p} s_2 \qquad \langle\!|thr_1[k], m_1|\!\rangle \overset{\alpha}{\twoheadrightarrow} \langle\!|c_2, m_2|\!\rangle \\ in = obs(thr_1, m_1) \qquad thr_2 = update_k(thr_1, c_2, \alpha)}{\langle\!|thr_1, m_1, s_1|\!\rangle \Rightarrow^{\mathfrak{s}}_{k,p} \langle\!|thr_2, m_2, s_2|\!\rangle}$$

The two premises on the left hand side require the selection of the kth thread with probability p by scheduler $\mathfrak{s}$ given the scheduler input $obs(thr_1, m_1)$. The third premise requires that the execution step of thread $thr_1[k]$ spawns new threads α and results in program state c_2 and memory state m_2. The fourth premise requires that the resulting thread pool thr_2 is obtained by $update_k(thr_1, c_2, \alpha)$.

Intuitively, $update_k$ replaces the program state at a position k by a program state c_2 and inserts newly created threads (i.e. α) after c_2. Formally, we define $update_k(thr, c, \alpha)$ by $sub(thr, 0, k-1) :: \langle c\rangle :: \alpha :: sub(thr, k+1, \#(thr)-1)$ if $c \neq \epsilon$, and otherwise by $sub(thr, 0, k-1) :: \alpha :: sub(thr, k+1, \#(thr)-1)$, where $::$ is the append operator that has the empty list $\langle\rangle$ as neutral element and $sub(thr, i, j)$ equals the list of threads i to j, i.e. $sub(thr, i, j) = \langle thr[i]\rangle :: sub(thr, i+1, j)$ if $i \leq j < \#(thr)$, and $sub(thr, i, j) = \langle\rangle$ otherwise.

We define the auxiliary function $stepsTo^{\mathfrak{s}} : (\mathcal{C}nf \times \mathfrak{P}(\mathcal{C}nf)) \to \mathfrak{P}(\mathbb{N}_0 \times]0; 1])$ by $stepsTo^{\mathfrak{s}}(cnf_1, Cnf) = \{(k, p) \mid \exists cnf_2 \in Cnf.\ cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2\}$.

That is, applying the function $stepsTo^{\mathfrak{s}}$ to cnf_1 and Cnf returns the labels of all possible system steps from $cnf_1 \in \mathcal{C}nf$ to some configuration in Cnf.

We call a property $P : \mathcal{C}nf \to Bool$ an *invariant* under $\mathfrak{s}$ if $P(cnf_1)$ and $cnf_1 \Rightarrow^{\mathfrak{s}}_{k,p} cnf_2$ imply $P(cnf_2)$ for all $cnf_1, cnf_2 \in \mathcal{C}nf$ and $(k, p) \in \mathbb{N}_0 \times]0; 1]$.

As a notational convention, we use $cnf \in \mathcal{C}nf$ to denote system configurations. Moreover, we introduce the selectors $pool(cnf) = thr$, $mem(cnf) = m$, and $sst(cnf) = s$ for decomposing a system configuration $cnf = \langle\!|thr, m, s|\!\rangle$.

2.2 Exemplary Programming Language

We define security on a semantic level. However, to give concrete examples we introduce a simple multi-threaded while language with dynamic thread creation. We define $\mathcal{E}$ and $\mathcal{C}$ of our example language by:

$$\begin{aligned} e &::= v \mid x \mid op(e, \ldots, e) \\ c &::= \mathsf{skip}_\iota \mid x{:=}_\iota e \mid c;c \\ &\quad \mid \mathsf{spawn}_\iota(c, \ldots, c) \mid \mathsf{if}_\iota\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}_\iota\ e\ \mathsf{do}\ c\ \mathsf{od} \end{aligned}$$

Some commands carry a label $\iota \in \mathbb{N}_0$ that we will use to identify program points.

The operational semantics for our language defines which instances of the judgment $\langle\!|c_1, m_1|\!\rangle \overset{\alpha}{\twoheadrightarrow} \langle\!|c_2, m_2|\!\rangle$ are derivable. The only notable aspect of the semantics is the label α. If the top-level command is $\mathsf{spawn}_\iota(c_0, \ldots, c_{n-1})$, then we have $\alpha = \langle c_0, \ldots, c_{n-1}\rangle$ while, otherwise, $\alpha = \langle\rangle$ holds.

For readability, we also use infix instead of prefix notation for expressions.

2.3 Attacker Model and Security Policies

A security policy describes what information a user is allowed to know based on a classification of information according to its confidentiality. We use sets of *security domains* to model different degrees of confidentiality. *Domain assignments* associate each program variable with a security domain.

Definition 2. *A* multi-level security policy *(brief: mls-policy) is a triple* $(\mathcal{D}, \leq, dom)$*, where* $\mathcal{D}$ *is a finite set of security domains,* $\leq$ *is a partial order on* $\mathcal{D}$*, and* $dom : \mathcal{V}ar \rightarrow \mathcal{D}$ *is a domain assignment.*

Intuitively, $d \not\leq d'$ with $d, d' \in \mathcal{D}$ models that no information must flow from the security domain d to the security domain d'.

A *d-observer* is a user who is allowed to observe a variable $x \in \mathcal{V}ar$, only if $dom(x) \leq d$. Hence, he can distinguish two memory states only if they differ in the value of at least one variable x with $dom(x) \leq d$. Dual to the ability to distinguish memory states is the following *d-indistinguishability.*

Definition 3. *Two memory states* $m \in \mathcal{M}em$ *and* $m' \in \mathcal{M}em$ *are* d-equal *for* $d \in \mathcal{D}$ *(denoted:* $m =_d m'$*), iff* $\forall x \in \mathcal{V}ar.\ (dom(x) \leq d \Longrightarrow m(x) = m'(x))$*.*

An *attacker* is a d-observer who tries to get information that he must not know. In terms of d-indistinguishability, this means that an attacker tries to distinguish initially d-equal memory states by running programs. Conversely, a program is intuitively secure, if running this program does not enable a d-observer to distinguish any two initial memory states that are d-equal. This intuition will be formalized by security properties in Section 3.

For the rest of the article, we assume that $(\mathcal{D}, \leq, dom)$ is an mls-policy.

2.4 Auxiliary Concepts for Relations

For any relation $R \subseteq A \times A$, there is at least one subset A' of A (namely $A' = \emptyset$) such that the restricted relation $R_{|A'} = R \cap (A' \times A')$ is an equivalence relation on A'. We characterize the subsets $A' \subseteq A$ for which $R_{|A'}$ constitutes an equivalence relation by a predicate $EquivOn_A \subseteq \mathfrak{P}(A \times A) \times \mathfrak{P}(A)$ that we define by $EquivOn_A(R, A')$ if and only if $R_{|A'}$ is an equivalence relation on A'.

In our definitions of security, we will use *partial equivalence relations* (brief: *pers*), i.e. binary relations that are symmetric and transitive but that need not be reflexive (see Sections 3 and 4.2). For each per $R \subseteq A \times A$, there is a unique maximal set $A' \subseteq A$ such that $EquivOn_A(R_{|A'}, A')$ holds. This maximal set is the set $A_{R,refl} = \{e \in A \mid e\ R\ e\}$, i.e. the subset of A on which R is reflexive.

Theorem 1. *If* $R \subseteq A \times A$ *is a per on a set* A *then* $EquivOn_A(R_{|A_{R,refl}}, A_{R,refl})$ *holds and* $\forall A' \subseteq A.\ (EquivOn_A(R_{|A'}, A') \Longrightarrow A' \subseteq A_{R,refl})$*.*

For brevity, we will use the symbol R instead of $R_{|A'}$ when this does not lead to ambiguities. In particular, we will write $EquivOn_A(R, A')$ meaning that $EquivOn_A(R_{|A'}, A')$ holds. Moreover, if $R \subseteq A \times A$ is a per, we will use $[e]_R$ to refer to the equivalence classes of an element $e \in A_{R,refl}$ under $R_{|A_{R,refl}}$.

Finally, we define a partial function $classes_A : \mathfrak{P}(A \times A) \rightharpoonup \mathfrak{P}(\mathfrak{P}(A))$ by $classes_A(R) = \{[e]_R \mid e \in A_{R,refl}\}$ if R is a per, while $classes_A(R)$ is undefined if R is not a per. That is, if R is a per, then $classes_A(R)$ equals the set of all equivalence classes of R (meaning the equivalence classes of $R_{|A_{R,refl}}$).

If the set A is clear from the context we write *classes* instead of $classes_A$.

3 Declassification in the Presence of Scheduling

A declassification is the deliberate release of secrets or, in other words, an intentional violation of an mls-policy. Naturally, such a release of secrets must be rigorously constrained to prevent unintended information leakage.

Example 1. Online music shops rely on not giving out songs for free. Hence, songs are only delivered to a user after he has paid. However, often downsampled previews are offered without payment to any user for promotion. The following example program shall implement this functionality.

$$P_1 = \mathsf{if}_1\ \mathsf{paid\ then\ out}{:=}_2\mathsf{song\ else\ out}{:=}_3\mathsf{downsample(song,\ bitrate)\ fi}$$

Consider an mls-policy with two domains *low* and *high*, and the total order $\leq$ with $high \not\leq low$. The domain assignment *dom* is defined such that $dom(\mathsf{song}) = high$ and $dom(\mathsf{out}) = low$ hold. Intuitively, this mls-policy means that song is confidential with respect to out. The program P_1 intuitively satisfies the requirement that any user may receive a downsampled preview, while only a user who has paid may receive the full song. Note that some information about the confidential song is released in both branches of P_1, i.e. a declassification occurs. However, what information is released differs for the two branches. $\Diamond$

As this example shows, an adequate control of declassification needs to respect what information (the full song or the preview) is released and where this release occurs (e.g., after payment has been checked by the program). This corresponds to the W-aspects *What* and *Where* that we address in this article. The W-aspects of declassification were first introduced in [21] and form the basis for a taxonomy of approaches to controlling declassification [33].

Before presenting our schema WHAT&WHERE$^{\mathfrak{s}}$ for scheduler-specific security properties that control what is declassified where (see Section 3.3), we introduce the simpler schema WHAT$^{\mathfrak{s}}$ (see Section 3.2) for controlling what is declassified. We show in Section 3.4 that WHAT&WHERE$^{\mathfrak{s}}$ implies WHAT$^{\mathfrak{s}}$ and also satisfies the so called prudent principles of declassification from [33].

3.1 Escape Hatches and Immediate Declassification Steps

As usual, we use pairs $(d, e) \in \mathcal{D} \times \mathcal{E}$, so called *escape hatches* [29], to specify what information may be declassified. Intuitively, (d, e) allows a d-observer to peek at the value of e, even if in e occurs a variable x with $dom(x) \not\leq d$. Hence, an escape hatch might enable a d-observer to distinguish memory states although they are d-equal. Dual to this ability is the following notion of (d, H)-*equality.*

Definition 4. *Two memory states m and m' are (d, H)-*equal *for $d \in \mathcal{D}$ and a set of escape hatches $H \subseteq \mathcal{D} \times \mathcal{E}$ (denoted: $m \sim_d^H m'$), iff $m =_d m'$ and $\forall (d', e) \in H.\ (d' \leq d \Longrightarrow (eval(e, m) = eval(e, m')))$ hold.*

We employ program points to restrict where declassification may occur. For each program, we assume a set of *program points* $\mathcal{PP} \subseteq \mathbb{N}_0$ and a function $pp : \mathcal{C} \rightarrow \mathcal{PP}$ that returns a program point for each sub-command of the program. Moreover, we assume that program points are unique within a program.

For our example language, we use the labels ι to define the function pp. For instance, $pp(\mathsf{out}{:=}_2\mathsf{song}) = 2$ and $pp(\mathsf{if}_1\ \mathsf{paid\ then}\ \ldots\ \mathsf{else}\ \ldots\ \mathsf{fi}) = 1$ hold. As sequential composition does not carry a label ι, we define $pp(c_1;c_2) = pp(c_1)$. Note that, after unwinding a loop, multiple sub-commands in a program state might be associated with the same program point. This results from copying the body of a while loop in the operational semantics if the guard evaluates to true.

We augment escape hatches with program points from $\mathcal{PP}$ and call the resulting triples *local escape hatches*. Like an escape hatch $(d, e) \in \mathcal{D} \times \mathcal{E}$, a local escape hatch $(d, e, \iota) \in \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ intuitively allows a d-observer to peek at the value of e. However, (d, e, ι) allows this only while the command at program point ι is executed. We use a set $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ to specify at which program points a d-observer may peek at which values. For Example 1, a natural set of local escape hatches would be $\{(low, \mathsf{downsample(song,\ bitrate)}, 3), (low, \mathsf{song}, 2)\}$.

Definition 5. *A* local escape hatch *is a triple $(d, e, \iota) \in \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$. We call a set of local escape hatches $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$* global *(denoted: $Global(lH)$) if $(d, e, \iota) \in lH$ implies $(d, e, \iota') \in lH$ for all $d \in \mathcal{D}$, $e \in \mathcal{E}$, and $\iota, \iota' \in \mathcal{PP}$.*

To aggregate the information that may be declassified at a given program point, we define the filter function $htchLoc : \mathfrak{P}(\mathcal{D} \times \mathcal{E} \times \mathcal{PP}) \times \mathcal{PP} \rightarrow \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ by $htchLoc(lH, \iota) = \{(d, e) \in \mathcal{D} \times \mathcal{E} \mid (d, e, \iota) \in lH\}$. Given a set of points $PP \subseteq \mathcal{PP}$, we use $htchLoc(lH, PP)$ as a shorthand notation for $\bigcup\{htchLoc(lH, \iota) \mid \iota \in PP\}$. Note that if lH is global then $\forall \iota, \iota' \in \mathcal{PP}.\ (htchLoc(lH, \iota) = htchLoc(lH, \iota'))$.

We call a command an *immediate d-declassification command* for a set of escape hatches $H \subseteq \mathcal{D} \times \mathcal{E}$ if its next execution step might reveal information to a d-observer that he should not learn according to the mls-policy, but that may permissibly be released to him due to some escape hatch in H.

Definition 6. *The predicate IDC_d on $\mathcal{C} \times \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ is defined by*

$$IDC_d(c, H) \Longleftrightarrow \begin{bmatrix} (\exists m, m' \in \mathcal{M}em.\ m =_d m' \wedge [\![c]\!](m) \neq_d [\![c]\!](m')) \\ \wedge\ (\forall m, m' \in \mathcal{M}em.\ m \sim_d^H m' \Longrightarrow [\![c]\!](m) =_d [\![c]\!](m')) \end{bmatrix}$$

The predicate IDC_d characterizes the immediate d-declassification commands for each set of escape hatches H. The predicate requires, firstly, that a release of secrets could, in principle, occur (i.e. for some pair of d-equal memories, the next step results in memories that are not d-equal) and, secondly, that no more information is released than allowed by the escape hatches (i.e. for all pairs of (d, H)-equal memories, the next step must result in d-equal memories).

Remark 1. If $IDC_d(c, htchLoc(lH, \iota))$ and $c \in \mathcal{C}$ is the command at program point $\iota \in \mathcal{PP}$ then c either has the form $x{:=}_\iota e$ or the form $x{:=}_\iota e; c'$. ♢

All concepts defined in this section are monotonic in the set of escape hatches, and the empty set of escape hatches is equivalent to forbidding declassification.

Theorem 2. *For all $d \in \mathcal{D}$ and $H, H' \subseteq \mathcal{D} \times \mathcal{E}$ the following propositions hold:*

1. $\forall m, m' \in \mathcal{M}em.\ ((\neg(m \sim_d^{H'} m') \wedge H' \subseteq H) \Longrightarrow \neg(m \sim_d^{H} m'))$ *;*
2. $\forall m, m' \in \mathcal{M}em.\ (m \sim_d^{\emptyset} m' \Longleftrightarrow m =_d m')$ *;*
3. $\forall c \in \mathcal{C}.\ ((IDC_d(c, H') \wedge H' \subseteq H) \Longrightarrow IDC_d(c, H))$ *; and*
4. $\forall c \in \mathcal{C}.\ \neg(IDC_d(c, \emptyset))$ *.*

A command is not a *d-declassification command* if its next execution step does not reveal any information to a d-observer that he cannot observe directly.

Definition 7. *The predicate NDC_d on $\mathcal{C}$ is defined by*

$$NDC_d(c) \Longleftrightarrow (\forall m, m' \in \mathcal{M}em.\ m =_d m' \implies [\![c]\!](m) =_d [\![c]\!](m'))$$

Note that $NDC_d(c)$ cannot hold if $IDC_d(c, H)$ holds for some $H \subseteq \mathcal{D} \times \mathcal{E}$. If c leaks beyond what H permits then neither $IDC_d(c, H)$ nor $NDC_d(c)$ holds.

We use $\iota \in \mathcal{PP}$ to denote program points, $H \subseteq \mathcal{D} \times \mathcal{E}$ to denote sets of escape hatches, and $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ to denote sets of local escape hatches.

3.2 The Security Conditions WHAT[5]

Security can be characterized based on pers (brief for partial equivalence relations, see Section 2.4). Following this approach, one defines a program to be secure if it is related to itself by a suitable per [30]. Consequently, the set of secure programs for a per $R \subseteq A \times A$ is $\bigcup classes_A(R)$. We will characterize confidentiality by pers that relate two thread pools only if they yield indistinguishable observations for any two initial configurations that must remain indistinguishable. Which configurations must remain indistinguishable depends on the observer's security domain d and on the set H of available escape hatches. We make this explicit by annotating pers with d and H (as, e.g., in $R_{d,H}$).

Definition 8. *Let $d \in \mathcal{D}$ and $H \subseteq \mathcal{D} \times \mathcal{E}$. The* lifting *of a relation $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ to a relation $R^{\uparrow}_{d,H} \subseteq \mathcal{C}nf \times \mathcal{C}nf$ is $R^{\uparrow}_{d,H} = (R_{d,H} \times \sim_d^H \times \sim)$.*

Note that, if two configurations cnf and cnf' are related by $R^{\uparrow}_{d,H}$ then they look the same to a d-observer because $mem(cnf) \sim_d^H mem(cnf')$ implies $mem(cnf) =_d mem(cnf')$. Moreover, the lifting of a per to the set $\mathcal{C}nf$ results, again, in a per.

Proposition 1. *If $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ is a per, then $R^{\uparrow}_{d,H} \subseteq \mathcal{C}nf \times \mathcal{C}nf$ is a per.*

Towards a Scheduler-specific Security Condition. Even if two configurations cnf and cnf' look the same to a d-observer, he might be able to infer in which of the configurations a program run must have started based on the observations that he makes during the run. For instance, he can exclude the possibility that the run started in cnf' if he makes an observation that is incompatible with all configurations that are reachable from cnf'. In this case, he

obtains information about the actual initial configuration from the fact that certain observations are impossible if the program is run under a given scheduler. In addition, an attacker might obtain information about the initial configuration from the probability of observations. For instance, if he makes certain observations quite often, when running the program in some initial configuration (which remains fixed and is initially unknown to the attacker), but the likelihood of this observation would be rather low if cnf' were the initial configuration, then the attacker can infer that cnf' is probably not the unknown initial configuration.[1]

We aim at defining a security property that rules out deductions of information about secrets based on the possibility as well as the probability of observations. We will focus on the latter aspect in the following because deductions based on possibilities are just a special case of deductions based on probabilities.

The probability of moving from a configuration cnf to some configuration in a set Cnf depends not only on the program, but also on the scheduler $\mathfrak{s}$.

Definition 9. *The function* $prob^{\mathfrak{s}} : \mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf}) \to [0;1]$ *is defined by:*

$$prob^{\mathfrak{s}}(cnf, Cnf) = \sum_{(k,p) \in stepsTo^{\mathfrak{s}}(cnf, Cnf)} p \ .$$

We will use the function $prob^{\mathfrak{s}}$ in our definition of WHAT$^{\mathfrak{s}}$ to capture that the likelihood of certain observations is the same in two given configurations.

If strict multi-level security were our goal then we could define security based on a per that relates two thread pools thr and thr' only if any two configurations $\langle\!| thr, m, s |\!\rangle$ and $\langle\!| thr', m', s' |\!\rangle$ with $m =_d m'$ and $s \sim s'$ cause indistinguishable observations. As we aim at permitting declassification, the situation is more involved. After a declassification occurred, a d-observer might be allowed to obtain information about the initial configuration that he cannot infer without running the program. However, such inferences should be strictly limited by the exceptions to multi-level security specified by a given set of escape hatches.

WHAT$^{\mathfrak{s}}$. We are now ready to define information-flow security. For each scheduler model $\mathfrak{s}$, we propose a security condition WHAT$^{\mathfrak{s}}$ that restricts declassification according to the constraints specified by a set of escape hatches. Following the per-approach, we define a multi-threaded program as WHAT$^{\mathfrak{s}}$-secure if it is related to itself by some relation $R_{d,H}$ that satisfies the following property.

Definition 10. *Let* $d \in \mathcal{D}$ *be a security domain and* $H \subseteq \mathcal{D} \times \mathcal{E}$ *be a set of escape hatches. An* $\mathfrak{s}$-specific strong (d, H)-bisimulation *is a per* $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *that fulfills the following two conditions:*

1. $\forall (cnf, cnf') \in R^{\uparrow}_{d,H}.\ \forall Cls \in classes(R^{\uparrow}_{d,H}).$
   $prob^{\mathfrak{s}}(cnf, Cls) = prob^{\mathfrak{s}}(cnf', Cls)$
2. *the property* $\lambda cnf \in \mathcal{Cnf}.\ (cnf \in \bigcup classes(R^{\uparrow}_{d,H}))$ *is an invariant under* $\mathfrak{s}$.

Condition 1 in Definition 10 ensures that if a single computation step is performed in two related configurations cnf and cnf' under a scheduler $\mathfrak{s}$ then each

[1] By increasing the number of runs such inferences are possible with high confidence, even if the difference between observed frequency and expected frequency is small.

equivalence class of $R^{\uparrow}_{d,H}$ is reached with the same probability from the two configurations. Condition 2 ensures that all configurations that can result after a computation step are again contained in some equivalence class of $R^{\uparrow}_{d,H}$. This lifts Condition 1 from individual steps to entire runs. The two conditions ensure that if two configurations are related by $R^{\uparrow}_{d,H}$ (which means they must remain indistinguishable for a d-observer who may use the escape hatches in H) then they, indeed, remain indistinguishable when the program is run.

Definition 11. *A thread pool $thr \in \mathcal{C}^*$* has secure information flow *for $(\mathcal{D}, \leq, dom)$ and $H \subseteq \mathcal{D} \times \mathcal{E}$ under $\mathfrak{s}$ (brief: $thr \in \mathrm{WHAT}^{\mathfrak{s}}$) iff for each $d \in \mathcal{D}$ there is a set $H' \subseteq H$ and a relation $R_{d,H'} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ such that $(thr\ R_{d,H'}\ thr)$ holds, and such that $R_{d,H'}$ is an* $\mathfrak{s}$-specific strong (d, H')-bisimulation.

Definition 11 ensures that if $thr \in \mathrm{WHAT}^{\mathfrak{s}}$ and $m \sim^H_d m'$ and $s \sim s'$ then the configurations $\langle\!| thr, m, s |\!\rangle$ and $\langle\!| thr, m', s' |\!\rangle$ yield indistinguishable observations for d while the multi-threaded program thr is executed under $\mathfrak{s}$.

$\mathrm{WHAT}^{\mathfrak{s}}$ will serve as the basis of our first scheduler-independence result in Section 4. More concretely, we will show that our previously proposed security condition WHAT_1 [20] implies $\mathrm{WHAT}^{\mathfrak{s}}$ for a wide range of schedulers. Moreover, we will use $\mathrm{WHAT}^{\mathfrak{s}}$ when arguing that our second security condition $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$ adequately controls what is declassified (see Section 3.4).

3.3 The Security Conditions $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$

We employ local escape hatches to specify where a particular secret may be declassified. The annotations of pers are adapted accordingly by replacing H with a set lH of local escape hatches. Moreover a set of program points $PP \subseteq \mathcal{PP}$ is added as third annotation (resulting in $R_{d,lH,PP}$). The set PP will be used to constrain local escape hatches in the definition of $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$.

Definition 12. *Let $d \in \mathcal{D}$, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$, and $PP \subseteq \mathcal{PP}$. The lifting of a relation $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ to a relation $R^{\uparrow}_{d,lH,PP} \subseteq Cnf \times Cnf$ is defined by $R^{\uparrow}_{d,lH,PP} = (R_{d,lH,PP} \times \sim^H_d \times \sim)$, where $H = htchLoc(lH, PP)$.*

Proposition 2. *If $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ is a per then $R^{\uparrow}_{d,lH,PP} \subseteq Cnf \times Cnf$ also is a per.*

Note that $\langle\!| thr, m, s |\!\rangle\ R^{\uparrow}_{d,lH,PP}\ \langle\!| thr', m', s' |\!\rangle$ implies that $m \sim^{htchLoc(lH,PP)}_d m'$ holds. This means that each variable $x \in \mathcal{Var}$ has the same value in m as in m' if x is visible for a d-observer (i.e. $m =_d m'$). Moreover, an expression $e \in \mathcal{E}$ has the same value in m as in m' if it may be declassified to d according to lH for at least one of the program points in PP (i.e. if $\exists (d', e, \iota) \in lH.\ (d' \leq d \wedge \iota \in PP)$).

Towards Controlling Where Declassification Occurs. If $NDC_d(c)$ holds then the next step of the command c respects strict multi-level security (i.e. no declassification to security domain d occurs in this step). If $IDC_d(c, H)$ holds then the next step of c might declassify information to d, and any such declassification is authorized by the escape hatches in H. However, if neither $NDC_d(c)$

nor $IDC_d(c, H)$ is true then there are memory states $m, m' \in \mathcal{Mem}$ such that $m \sim_d^H m'$ holds while $[\![c]\!](m) =_d [\![c]\!](m')$ does not hold. This means that information might be leaked whose declassification is not permitted by H.

In our definition of the security condition, we need to rule out this third possibility, i.e. $\neg IDC_d(c, H) \wedge \neg NDC_d(c)$ where H is the set of escape hatches that are enabled. Which escape hatches are enabled in a given computation step depends on the set of local escape hatches and on the set of program points that might cause the computation step.

The set of program points that might cause a transition from a configuration cnf to some configuration in a set Cnf depends on the scheduler.

Definition 13. *The function* $pps^{\mathfrak{s}} : (\mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf})) \to \mathfrak{P}(\mathcal{PP})$ *is defined by:*

$$pps^{\mathfrak{s}}(cnf, Cnf) = \{pp(cnf[k]) \mid (k, p) \in stepsTo^{\mathfrak{s}}(cnf, Cnf)\} \quad .$$

Using $pps^{\mathfrak{s}}$, we define which hatches might be relevant for a computation step.

Definition 14. *The function* $htchs^{\mathfrak{s}} : (\mathfrak{P}(\mathcal{D} \times \mathcal{E} \times \mathcal{PP}) \times \mathcal{Cnf} \times \mathfrak{P}(\mathcal{Cnf})) \to \mathfrak{P}(\mathcal{D} \times \mathcal{E})$ *is defined by* $htchs^{\mathfrak{s}}(lH, cnf, Cnf) = htchLoc(lH, pps^{\mathfrak{s}}(cnf, Cnf))$.

WHAT&WHERE$^{\mathfrak{s}}$. We are now ready to introduce our second schema for scheduler-specific security conditions. Unlike WHAT$^{\mathfrak{s}}$, WHAT&WHERE$^{\mathfrak{s}}$ allows one to control where a particular declassification can occur. This combined control of the W-aspects *What* and *Where* is needed, for instance, in Example 1.

Like in Section 3.2, we define a class of pers on thread pools to characterize indistinguishability from the perspective of a d-observer. A program is then defined to be secure under a scheduler $\mathfrak{s}$ if it is related to itself. Which configurations must remain indistinguishable differs from Section 3.2 because information may only be declassified in a computation step if this is permitted by the set of local escape hatches that are enabled at this step. That is, declassification is more constrained than in Section 3.2.

Definition 15. *Let $d \in \mathcal{D}$ be a security domain, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ be a set of local escape hatches, and $PP \subseteq \mathcal{PP}$ be a set of program points. An* $\mathfrak{s}$-specific strong (d, lH, PP)-bisimulation *is a per* $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ *that fulfills the following three conditions:*

1. $\forall (thr, thr') \in R_{d,lH,PP} \,.\, \forall k \in \mathbb{N}_0.$
   $k < \#(thr) \implies (NDC_d(thr[k]) \vee IDC_d(thr[k], htchLoc(lH, pp(thr[k]))))$
2. $\forall (cnf, cnf') \in R^{\uparrow}_{d,lH,PP}.\, \forall Cls \in classes(R^{\uparrow}_{d,lH,PP}).$
   $(htchs^{\mathfrak{s}}(lH, cnf, Cls) \cup htchs^{\mathfrak{s}}(lH, cnf', Cls)) \subseteq htchLoc(lH, PP)$
   $\implies prob^{\mathfrak{s}}(cnf, Cls) = prob^{\mathfrak{s}}(cnf', Cls)$
3. $\lambda cnf \in \mathcal{Cnf}.\, (cnf \in \bigcup classes(R^{\uparrow}_{d,lH,PP}))$ *is an invariant under* $\mathfrak{s}$

Condition 1 in Definition 15 ensures that each thread $thr[k]$ either causes no declassification to the security domain d or is an immediate declassification command for the set of locally available escape hatches. Condition 2 ensures that

if a single computation step is performed in two related configurations cnf and cnf' then each equivalence class of $R^{\uparrow}_{d,lH,PP}$ is reached with the same probability from the two configurations. In contrast to Condition 1 in Definition 10, this is only required under the condition that each escape hatch (d', e) with $d' \leq d$, that is available at some program point ι that might cause the next computation step, is also contained in $htchLoc(lH, PP)$. Note that this precondition (i.e. $(htchs^{\mathfrak{s}}(lH, cnf, Cls) \cup htchs^{\mathfrak{s}}(lH, cnf', Cls)) \subseteq htchLoc(lH, PP)$) is trivially fulfilled if $PP = \mathcal{PP}$ holds. However, if PP is a proper subset of $\mathcal{PP}$ then the precondition might be violated. That is, choosing a set PP that is too small might lead to missing possibilities for information laundering. We will avoid this pitfall by universally quantifying over all subsets $PP \subseteq \mathcal{PP}$ in the definition of WHAT&WHERE$^{\mathfrak{s}}$. Finally, Condition 3 ensures that all configurations that can result after a computation step are again contained in some equivalence class of $R^{\uparrow}_{d,lH,PP}$. This lifts Condition 1 and 2 from individual steps to entire runs.

Definition 16. *A thread pool $thr \in \mathcal{C}^*$* has secure information flow *for $(\mathcal{D}, \leq, dom)$ and $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ under $\mathfrak{s}$ (brief: $thr \in$ WHAT&WHERE$^{\mathfrak{s}}$) iff for each $d \in \mathcal{D}$ and for each $PP \subseteq \mathcal{PP}$ there are a set $lH' \subseteq lH$ and a relation $R_{d,lH',PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ such that $(thr\ R_{d,lH',PP}\ thr)$ holds, and such that $R_{d,lH',PP}$ is an $\mathfrak{s}$-specific strong (d, lH', PP)-bisimulation.*

The structure of Definition 16 is similar to the one of Definition 11. The main differences are, firstly, that a set lH of local escape hatches is used instead of a set H of escape hatches and, secondly, that the escape hatches, that are available to a d-observer, are further constrained by a set $PP \subseteq \mathcal{PP}$. The universal quantification over all subsets PP of $\mathcal{PP}$ is crucial for achieving the desired control of where a declassification can occur. It were not enough to require Condition 2 in Definition 15 just for $PP = \mathcal{PP}$ because the resulting security guarantee would control what is declassified without restricting where declassification can occur.

Example 2. Let $P_2 = \mathsf{if}_1\ \mathsf{h}\ \mathsf{then}\ \mathsf{spawn}_2(\mathsf{l}{:=}_3 0, \mathsf{l}{:=}_4 1)\ \mathsf{else}\ \mathsf{spawn}_5(\mathsf{l}{:=}_6 1, \mathsf{l}{:=}_7 0)\ \mathsf{fi}$ and $lH = \emptyset$. We consider a biased scheduler $\mathfrak{s}$ that selects the second of two threads with lower, but non-zero probability. Independent of the value of h, P_2 might terminate with a memory state in which $\mathsf{l} = 0$ holds as well as with a memory state in which $\mathsf{l} = 1$ holds. Nevertheless, a good guess about the initial value of h is possible after observing several runs with the same initial memory. If $\mathsf{l} = 0$ is observed significantly more often than $\mathsf{l} = 1$, then it is likely that $\mathsf{h} = \mathsf{False}$ holds in the initial state. Hence, the program is intuitively insecure.

Running P_2 with two memories that differ in h deterministically results in two different thread pools, namely in $\langle \mathsf{l}{:=}_3 0, \mathsf{l}{:=}_4 1 \rangle$ and $\langle \mathsf{l}{:=}_6 1, \mathsf{l}{:=}_7 0 \rangle$. These two thread pools must be related by $R_{low,lH,\mathcal{PP}}$ according to Condition 2 in Definition 15. However, the probability of moving from these two configurations into the same equivalence class differs as our biased scheduler chooses the first thread with a higher probability than the second. Therefore, Condition 2 is violated by the second computation step and, hence, $P_2 \notin$ WHAT&WHERE$^{\mathfrak{s}}$. ◊

Example 3. Let $P_3 = \mathsf{h2}{:=}_1\mathsf{absolute}(\mathsf{h2});\ \mathsf{if}_2\ \mathsf{h1}\ \mathsf{then}\ \mathsf{l1}{:=}_3\mathsf{h2}\ \mathsf{else}\ \mathsf{l1}{:=}_4\text{-}\mathsf{h2}\ \mathsf{fi}$ and $lH = \{(low, \mathsf{h2}, 3), (low, \mathsf{h2}, 4)\}$. The assignments in both branches do not reveal more

information than permitted by the respective local escape hatches. However, the sign of the value stored in l1 after a run reveals information about the initial value of h1 in addition. Hence, the program is intuitively insecure.

Two consecutive computation steps of P_3 in two memories that differ in h1 result in two different thread pools, namely in $\langle \mathsf{l1}{:=}_3\mathsf{h2}\rangle$ and $\langle \mathsf{l1}{:=}_4\text{-}\mathsf{h2}\rangle$. According to Condition 2 in Definition 15, these two thread pools must be related by $R_{low,lH,\mathcal{PP}}$. However, a third computation step in each of them results in two memories that are *low*-distinguishable and, hence, $P_3 \notin$ WHAT&WHERE$^{\mathfrak{s}}$. $\Diamond$

3.4 Meta-properties of the Scheduler-Specific Security Properties

The security conditions WHAT&WHERE$^{\mathfrak{s}}$ restrict declassification according to a set of local escape hatches. This allows one a more fine-grained control of declassification by restricting what information can be declassified where. In comparison to WHAT$^{\mathfrak{s}}$, declassification shall be controlled more rigorously, and WHAT&WHERE$^{\mathfrak{s}}$ is indeed at least as restrictive as WHAT$^{\mathfrak{s}}$.

Theorem 3. *Let $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ and $thr \in \mathcal{C}^*$. If $thr \in$ WHAT&WHERE$^{\mathfrak{s}}$ with lH then $thr \in$ WHAT$^{\mathfrak{s}}$ with $H = htchLoc(lH, \mathcal{PP})$.*

In [32], various so called prudent principles were proposed as sanity checks for definitions of information-flow security that are compatible with declassification. In order to convince ourselves about the adequacy of our novel security condition, we have checked WHAT&WHERE$^{\mathfrak{s}}$ against these principles, and we have shown that it satisfies the following prudent principles (based on the formalization of a slightly augmented set of prudent principles in [16]):

Semantic consistency [32]. The (in)security of a program is invariant under semantic-preserving transformations of declassification-free subprograms.

Monotonicity of release [32] . Allowing further declassifications for a program that is WHAT&WHERE$^{\mathfrak{s}}$-secure cannot render it insecure.

Persistence [16] . For every program that satisfies WHAT&WHERE$^{\mathfrak{s}}$, all programs that are reachable also satisfy this security condition.

Relaxation [16] . Every program that satisfies noninterference also satisfies WHAT&WHERE$^{\mathfrak{s}}$.

Noninterference up-to [16] . Every WHAT&WHERE$^{\mathfrak{s}}$-secure program also satisfies noninterference if it were executed in an environment that terminates the program when it is about to perform a declassification.

Another prudent principle proposed in [32] is Non-occlusion. This principle requires that the presence of a declassifying operation cannot mask other covert information leaks. Unfortunately, a bootstrapping problem occurs. Any adequate formal characterization of non-occlusion itself is an adequate definition of information-flow security with controlled declassification. If such an adequate characterization existed then there would be no need to propose a definition of information-flow security.

4 Secure Declassification for Multi-threaded Programs

When developing a multi-threaded program, usually a specification of the scheduler's interface is available, but the concrete scheduler is not known. An interface might reveal to a scheduler information about the current configuration such as the number of active threads and the values of special program variables (e.g., for setting scheduling priorities). However, the scheduler should not have direct access to secrets via the interface because the scheduling of threads might have an effect on the probability of an attacker's observations. Hence, one should treat all elements of the scheduler's interface like public sinks in a security analysis.

We specify interfaces to schedulers by observation functions (see Section 2.1) and assume that interfaces do not give a scheduler access to the value of program counters as well as of variables that might contain secrets. This is captured by the following restriction on observation functions.

Definition 17. *An observation function $obs \in \mathcal{O}bs$ is* confined *wrt. an mls-policy $(\mathcal{D}, \leq, dom)$, iff for all thr_1, $thr_1' \in \mathcal{C}^*$ and all $m_1, m_1' \in \mathcal{M}em$:*
$$(\sharp(thr_1) = \sharp(thr_1') \wedge \exists d \in \mathcal{D}.\ m_1 =_d m_1') \implies obs(thr_1, m_1) = obs(thr_1', m_1')\ .$$

If the interface to the scheduler is confined, then the scheduling behavior is identical for any two configurations that have the same number of active threads and assign the same value to each variable that is visible for all security domains.

Remark 2. Note that our restriction to confined observation functions does not eliminate the refinement problem for schedulers. As already pointed out in [35], a program might have secure information flow if executed with the fictitious possibilistic scheduler, but be insecure if executed with a uniform scheduler. Since a uniform scheduler bases its decisions only on the number of active threads, its interface can be captured by a confined observation function. Another example of a scheduler with a confined observation function is the biased scheduler described in Example 2. The program P_2 in this example is insecure if run with the biased scheduler, but it would be secure if run with the possibilistic scheduler. ◊

As the concrete scheduler is usually not known when developing a program, properties are needed that allow one to reason about security independently of the concrete scheduler. In this section, we recall the security property WHAT$_1$ from [20] and propose the novel security property WHAT&WHERE. We show that these properties imply WHAT$^{\mathfrak{s}}$ and WHAT&WHERE$^{\mathfrak{s}}$, respectively, for all schedulers $\mathfrak{s}$ and confined observation functions. These scheduler-independence results provide the theoretical basis for reasoning in a sound way about the security of multi-threaded programs without knowing the concrete scheduler.

4.1 Scheduler-Independent WHAT-Security

The following definition of strong (d, H)-bisimulations is an adaptation of the corresponding notion from [20] to the formal exposition used in this article.

$$\forall thr, thr' \in \mathcal{C}^*. \forall m_1, m_1' \in \mathcal{Mem}. \forall k \in \mathbb{N}_0. \forall \alpha \in \mathcal{C}^*. \forall c \in \mathcal{C}_\epsilon. \forall m_2 \in \mathcal{Mem}.$$
$$\left[\begin{array}{l} thr\ R_{d,H}\ thr' \wedge m_1 \sim_d^H m_1' \wedge \boxed{\langle\!| thr[k], m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c, m_2 |\!\rangle} \\ \Longrightarrow \exists \alpha' \in \mathcal{C}^*. \exists c' \in \mathcal{C}_\epsilon. \exists m_2' \in \mathcal{Mem}. \\ \quad \left[\boxed{\langle\!| thr'[k], m_1' |\!\rangle \xrightarrow{\alpha'} \langle\!| c', m_2' |\!\rangle} \wedge \boxed{\langle c \rangle\ R_{d,H}\ \langle c' \rangle \wedge \alpha\ R_{d,H}\ \alpha'} \wedge \boxed{m_2 \sim_d^H m_2'} \right] \end{array} \right]$$

Fig. 1. Condition 2 in the definition of strong (d, H)-bisimulations

Definition 18. *Let $d \in \mathcal{D}$ be a security domain and $H \subseteq \mathcal{D} \times \mathcal{E}$ be a set of escape hatches. A* strong (d, H)-bisimulation *is a per $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ that fulfills the following two conditions:*

1. *$\forall (thr, thr') \in R_{d,H}\ .\ \#(thr) = \#(thr')$ and*
2. *$R_{d,H}$ satisfies the formula in Figure 1.*

If two thread pools $thr, thr' \in \mathcal{C}^*$ are strongly (d, H)-bisimilar, and the scheduler chooses in some memory state m the k'th thread of the first thread pool thr for a step, then the thread at position k in the second thread pool thr' can also perform a computation step in any memory state m' that is (d, H)-equal to m (see dark-gray boxes in Figure 1). Moreover, the program states as well as the lists of spawned threads resulting after these two steps are, again, strongly (d, H)-bisimilar (see medium-gray box in Figure 1). Finally, the resulting memory states are, again (d, H)-equal (see light-gray box in Figure 1).

Definition 19. *A thread pool thr* has secure information flow *for $(\mathcal{D}, \leq, dom)$ and $H \subseteq \mathcal{D} \times \mathcal{E}$ (brief: $thr \in WHAT_1$) iff for each $d \in \mathcal{D}$ there is a strong (d, H)-bisimulation $R_{d,H} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ such that $(thr\ R_{d,H}\ thr)$ holds.*

We are now ready to present our scheduler-independence result for WHAT-security. The theorem states that WHAT_1 implies $\text{WHAT}^{\mathfrak{s}}$ for each scheduler model $\mathfrak{s}$. Hence, WHAT_1 is suitable for reasoning about WHAT-security in a sound manner without having to explicitly consider scheduling.

Theorem 4. *Let $(\mathcal{D}, \leq, dom)$ be an mls-policy, $H \subseteq \mathcal{D} \times \mathcal{E}$ be a set of escape hatches, $obs \in \mathcal{Obs}$ be an observation function that is confined wrt. $(\mathcal{D}, \leq, dom)$, and $thr \in \mathcal{C}^*$ be a thread pool. If $thr \in WHAT_1$ holds, then $thr \in \text{WHAT}^{\mathfrak{s}}$ holds for each scheduler model $\mathfrak{s}$.*

4.2 Scheduler-Independent WHAT&WHERE-Security

Like in Section 3.3, we use pers that are annotated with a security domain d, a set lH of local escape hatches, and a set PP of program points. Unlike in Section 3.3, we constrain pers without referring to system steps, because system steps depend on the concrete scheduler's behavior. Our novel security property WHAT&WHERE shall provide adequate control over what information is declassified where, independently of the scheduler under that a program is run.

$$\forall thr, thr' \in \mathcal{C}^*.\ \forall m_1, m_1' \in \mathcal{Mem}.\ \forall k \in \mathbb{N}_0.\ \forall \alpha \in \mathcal{C}^*.\ \forall c \in \mathcal{C}_\epsilon.\ \forall m_2 \in \mathcal{Mem}.$$
$$\left[\begin{array}{l} thr\ R_{d,lH,PP}\ thr' \wedge m_1 \sim_d^{htchLoc(lH,PP)} m_1' \wedge \boxed{\langle\!| thr[k], m_1 |\!\rangle \xrightarrow{\alpha} \langle\!| c, m_2 |\!\rangle} \\ \Longrightarrow \exists \alpha' \in \mathcal{C}^*.\ \exists c \in \mathcal{C}_\epsilon.\ \exists m_2' \in \mathcal{Mem}. \\ \quad \left[\begin{array}{l} \boxed{\langle\!| thr'[k], m_1' |\!\rangle \xrightarrow{\alpha'} \langle\!| c', m_2' |\!\rangle \wedge \langle c\rangle\ R_{d,lH,PP}\ \langle c'\rangle \wedge \alpha\ R_{d,lH,PP}\ \alpha'} \\ \wedge \left(\boxed{m_2 \sim_d^{htchLoc(lH,PP)} m_2'} \vee \boxed{htchLoc(lH, pp(thr[k])) \not\subseteq htchLoc(lH, PP)} \right) \end{array}\right] \end{array}\right]$$

Fig. 2. Condition 3 in the definition of strong (d, lH, PP)-bisimulations

Definition 20. *Let $d \in \mathcal{D}$ be a security domain, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ be a set of local escape hatches, and $PP \subseteq \mathcal{PP}$ be a set of program points. A* strong (d, lH, PP)-bisimulation *is a per $R_{d,lH,PP} \subseteq \mathcal{C}^* \times \mathcal{C}^*$ that fulfills the following three conditions:*

1. $\forall (thr, thr') \in R_{d,lH,PP}\ .\ \#(thr) = \#(thr')$,
2. $\forall (thr, thr') \in R_{d,lH,PP}\ .\ \forall k \in \mathbb{N}_0.$
   $k < \#(thr) \Longrightarrow (NDC_d(thr[k]) \vee IDC_d(thr[k], htchLoc(lH, pp(thr[k]))))$,
3. *$R_{d,lH,PP}$ satisfies the formula in Figure 2.*

Condition 1 in Definition 20 ensures that related thread pools have equal size (like Condition 1 in Definition 18). Condition 2 ensures that each thread either causes no declassification to d or is an immediate declassification command for the set of locally available escape hatches (like Condition 1 in Definition 15).

Condition 3 bears similarities with Condition 2 in Definition 18 (see Figure 1). If two thread pools $thr, thr' \in \mathcal{C}^*$ are strongly (d, lH, PP)-bisimilar, and the scheduler chooses in some memory state m the k'th thread of thr for a step, then the k'th thread of thr' can also perform a computation step in any memory state m' that is (d, H)-equal to m (where $H = htchLoc(lH, PP)$), and the resulting program states as well as lists of spawned threads are, again, strongly (d, lH, PP)-bisimilar (see dark-gray boxes in Figure 2). Note that an expression e that occurs in a local escape hatch $(d', e, \iota) \in lH$ need not have the same value in m and m' if $\iota \notin PP$. Consequently, Condition 3 only requires the resulting memory states to be (d, H)-equal (see medium-gray box in Figure 2), if no such local escape hatch might affect the computation step under consideration (see light-gray box in Figure 2). Like in Section 3.3, choosing a set PP that is too small might lead to missing possibilities for information laundering and, again, we will avoid this pitfall by universally quantifying over all subsets $PP \subseteq \mathcal{PP}$.

Definition 21. *A thread pool $thr \in \mathcal{C}^*$ has* secure information flow *for an mls-policy $(\mathcal{D}, \leq, dom)$ and a set of local escape hatches $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ (brief: $thr \in$ WHAT&WHERE) iff for each $d \in \mathcal{D}$ and for each $PP \subseteq \mathcal{PP}$ there is a strong (d, lH, PP)-bisimulation $R_{d,lH,PP}$ such that $(thr\ R_{d,lH,PP}\ thr)$ holds.*

We are now ready to present our second scheduler-independence result.

$$[\text{tconstd}]\frac{}{H \vdash v : d} \qquad [\text{tvard}]\frac{dom(x) = d}{H \vdash x : d} \qquad [\text{thatchd}]\frac{(d,e) \in H}{H \vdash e : d}$$

$$[\text{topd}]\frac{H \vdash e_1 : d_1 \;\ldots\; H \vdash e_m : d_m \qquad \forall i \in \{1,\ldots,m\}.\; d_i \leq d}{H \vdash op(e_1,\ldots,e_m) : d}$$

Fig. 3. Security type system for expressions

Theorem 5. *Let $(\mathcal{D}, \leq, dom)$ be an mls-policy, $lH \subseteq \mathcal{D} \times \mathcal{E} \times \mathcal{PP}$ be a set of local escape hatches, $obs \in \mathcal{Obs}$ be an observation function that is confined wrt. $(\mathcal{D}, \leq, dom)$, and $thr \in \mathcal{C}^*$ be a thread pool. If $thr \in$ WHAT&WHERE holds, then $thr \in$ WHAT&WHERE$^{\mathfrak{s}}$ holds for each scheduler model $\mathfrak{s}$.*

The scheduler-independence theorem shows that WHAT&WHERE provides as much control of what information is declassified where as WHAT&WHERE$^{\mathfrak{s}}$, but without referring to specific schedulers. Hence, WHAT&WHERE is adequate for reasoning about the security of programs when the scheduler is unknown.

5 Security Type System

Our security property WHAT&WHERE is compositional in the following sense:

Theorem 6. *Let $c_0, \ldots, c_{n-1} \in \mathcal{C}$ be commands and $e \in \mathcal{E}$ be an expression. If $\langle c_0 \rangle, \ldots, \langle c_{n-1} \rangle \in$ WHAT&WHERE and if $(m =_d m' \Longrightarrow eval(e,m) = eval(e,m'))$ holds for all $m, m' \in \mathcal{Mem}$ and all $d \in \mathcal{D}$, then we have:*

1. $\langle c_0; c_1 \rangle \in$ WHAT&WHERE,
2. $\langle \mathsf{spawn}_\iota(c_0, \ldots, c_{n-1}) \rangle \in$ WHAT&WHERE,
3. $\langle \mathsf{while}_\iota\ e\ \mathsf{do}\ c_0\ \mathsf{od} \rangle \in$ WHAT&WHERE, *and*
4. $\langle \mathsf{if}_\iota\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \rangle \in$ WHAT&WHERE.

We will now define a syntactic approximation of WHAT&WHERE for our example language in Section 2.2 in the form of a type system. Before we present the typing rules for the commands, we present typing rules for expressions. The judgment $H \vdash e : d$ (where $H \subseteq \mathcal{D} \times \mathcal{E}$, $e \in \mathcal{E}$ and $d \in \mathcal{D}$) can be derived with the typing rules in Figure 3. Intuitively, the judgment $H \vdash e : d$ shall model that the value of e only depends on information that a d-observer is permitted to obtain (for a given mls-policy and the set H of escape hatches). That the typing rules capture this intuition is ensured by the following theorem:

Theorem 7. *Let $H \subseteq \mathcal{D} \times \mathcal{E}$, $e \in \mathcal{E}$, and $d \in \mathcal{D}$. If $H \vdash e : d$ is derivable then*

$$\forall m, m' \in \mathcal{Mem}.\ \left[m \sim_d^H m' \Longrightarrow eval(e,m) = eval(e,m') \right] .$$

For verifying the security of programs we use judgments of the form $\vdash c$ (where $c \in \mathcal{C}$). Intuitively, $\vdash c$ shall express that c satisfies our novel security condition WHAT&WHERE from Section 4.2. The typing rules for this judgment are presented in Figure 4. The typing rules tseq, tspawn, twhile and tif correspond

$$[\text{tassign}]\frac{htchLoc(lH,\iota)\vdash e:d \quad d\leq dom(x) \quad SubstClosure(lH,x,e)}{\vdash x{:=}_\iota e}$$

$$[\text{tseq}]\frac{\vdash c_1 \quad \vdash c_2}{\vdash c_1\,;c_2} \qquad [\text{tif}]\frac{\emptyset\vdash e:d' \quad \forall d''.\,d'\leq d'' \quad \vdash c_1 \quad \vdash c_2}{\vdash \mathsf{if}_\iota\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} \qquad [\text{tskip}]\frac{}{\vdash \mathsf{skip}_\iota}$$

$$[\text{tspawn}]\frac{\vdash c_0\ \ldots \vdash c_{n-1}}{\vdash \mathsf{spawn}_\iota(c_0,\ldots,c_{n-1})} \qquad [\text{twhile}]\frac{\emptyset\vdash e:d' \quad \forall d''.\,d'\leq d'' \quad \vdash c}{\vdash \mathsf{while}_\iota\ e\ \mathsf{do}\ c\ \mathsf{od}}$$

Fig. 4. Security type system for commands

to the four cases of the compositionality theorem (i.e., Theorem 6). Note that the first two preconditions of twhile and tif indeed ensure that $(m =_d m' \Longrightarrow eval(e,m) = eval(e,m'))$ holds for all $m, m' \in \mathcal{Mem}$ and all $d \in \mathcal{D}$. The first two preconditions of the rule for assignments (i.e., tassign) ensure that information only flows into a variable $x \in \mathcal{Var}$ if this is permissible according to the mls-policy and to the set of locally available escape hatches. The third precondition of rule tassign prevents information laundering like in the following example.

Example 4. Let $P_4 = \mathsf{h2}{:=}_1 0; \mathsf{l}{:=}_2 \mathsf{h1}{+}\mathsf{h2}$ and $lH = \{(low, \mathsf{h1}{+}\mathsf{h2}, \iota) \mid \iota \in \mathcal{PP}\}$. If the third precondition of rule tassign were not present, then P_4 would be accepted by the type system. However, the program reveals the value of h1 to a *low*-observer, which is not permitted by lH under the two-level mls-policy. $\Diamond$

In order to avoid such possibilities for information laundering via escape hatches, we use the predicate *SubstClosure* in the third precondition of rule tassign:

Definition 22. *We define* $SubstClosure \subseteq \mathfrak{P}(\mathcal{D}\times\mathcal{E}\times\mathcal{PP})\times\mathcal{Var}\times\mathcal{E}$ *by*

$$SubstClosure(lH,x,e) \Longleftrightarrow \forall (d',e',\iota')\in lH.\ (d',e'[x\backslash e],\iota')\in lH$$

where $e'[x\backslash e]$ *is the expression that results from substituting all occurrences of variable* x *in expression* e' *by the expression* e.

The third precondition of rule tassign (i.e., $SubstClosure(lH,x,e)$) requires that, if the target x of an assignment occurs in the expression e' of some $(d',e',\iota')\in lH$ then $(d',e'[x\backslash e],\iota')\in lH$ must also hold. This ensures that the local escape hatch $(d',e',\iota')\in lH$ may still be used legitimately, after assigning e to x.

The following soundness theorem shows that the judgment $\vdash c$ indeed captures WHAT&WHERE:

Theorem 8. *Let* $c \in \mathcal{C}$. *If* $\vdash c$ *is derivable then* $c \in$ WHAT&WHERE *holds.*

If a program is typable with our security type system, then it adequately controls what information is declassified where, no matter under which scheduler the program is run. This follows from the soundness theorem above in combination with our scheduler-independence result for WHAT&WHERE (i.e., Theorem 5).

Example 5. We reconsider the program P_1 from Example 1 and the set $lH = \{(low, \mathsf{downsample(song,\ bitrate)}, 3), (low, \mathsf{song}, 2)\}$. The judgment $\vdash P_1$ can be derived by applying the rules tif, tvard (for paid and $d = low$), tassign, thatchd

(for song), tassign, thatchd (for downsample(song, bitrate)). From Theorem 8 and Theorem 5 we obtain $P_1 \in \text{WHAT\&WHERE}^{\mathfrak{s}}$ regardless of the scheduler $\mathfrak{s}$. ◊

Remark 3. The type system presented in this section is suitable for verifying WHAT&WHERE-security in a sound way. In the definition of the typing rules, we aimed for conceptual simplicity rather than for maximizing the precision of the analysis. For instance, a more fine-grained treatment of conditionals could be developed by using safe approximation relations (like in [21]). ◊

6 Related Work

Research on information-flow security has addressed scheduler independence as well as declassification, but not yet the combination of these two aspects.

To achieve scheduler-independent information-flow security, three main directions have been explored. *Observational determinism* [36,13] requires that all observations of an attacker are deterministically determined by information that this attacker may obtain. This ensures that security is not affected by how non-determinism is resolved (including the selection of threads by a scheduler). An alternative approach to achieving scheduler independence requires a non-standard interface to schedulers. Schedulers can be asked to "hide" or "unhide" threads via this interface, where threads classified as "unhidden" may only be scheduled if no "hidden" threads are active [7,28]. *Strong security* [31] achieves scheduler independence by defining security based on stepwise bisimulation relations that match steps of threads at the same position, like in this article. *FSI-security* [22] is also a scheduler-independent security condition, although it is less restrictive than strong security. None of these approaches supports declassification.

Scheduler-independence results can be viewed as solutions to the refinement paradox [14] in a particular domain. In fact, the approach to define security based on observational determinism was originally developed as a general solution to avoid the refinement paradox [27]. Unfortunately, this approach also forbids intended non-determinism. An alternative is to identify notions of refinement that preserve information-flow security. For event-based specifications, such refinement operators are proposed in [18]. For sequential programs, refinements that preserve the property "ignorance of secrets" are characterized in [24].

The challenge of certifying information-flow security while permitting declassification is addressed in various publications (see [33] for an overview). In order to make differences in the goals of different approaches to controlling declassification explicit, three aspects of declassification were distinguished in [21]: *What* information may be declassified, *Where* information may be declassified, and *Who* may declassify information. Four dimensions of declassification, which are similar to these W-aspects, are used in [33] to classify existing approaches to declassification. Our novel security condition WHAT&WHERE for multi-threaded programs addresses the aspects *What* and *Where* in an integrated fashion.

For sequential programs, there are solutions addressing the aspects *What* (e.g., [29,15,16]), *Where* (e.g., [10,2,11]), and *Who* (e.g., [25,26,17]) in isolation. There

are also approaches that control *What* information is declassified *Where*. *Localized delimited release* [3] and the security conditions in [4] permit to specify from which program point on the value of a given expression may be declassified. *Delimited non-disclosure* [6] and *delimited gradual release* [5] permit to specify exactly at which position a given expression may be declassified. For the latter two, the value that may be declassified is the value to which the expression evaluates when the declassification is performed. In all other approaches (including the approach in this article), the value that may be declassified is the initial value of the expression. The relation between these two interpretations of escape hatches is clarified in [16]. All previously proposed approaches to control *What* is declassified *Where* were developed for sequential programs.

In a multi-threaded setting, several approaches adopt the ideas underlying *strong security* [31]. *Intransitive noninterference* [21] and WHERE [20] permit declassification by dedicated declassification commands that comply with a flow relation, which may be an intransitive relation. The properties WHAT_1 and WHAT_2 in [20] control that what is declassified complies with a given set of escape hatches. The conditions SIMP_D^* [9] and *non-disclosure* [1] are also based on step-wise bisimulations. However, they do not require that matching steps are executed by threads at the same position, which seems necessary for achieving scheduler independence. While some of these approaches strive for scheduler independence, no scheduler-independence result has been published for them.

7 Conclusion

The scheduler-independence results presented in this article constitute the first two such results for definitions of information-flow security that are compatible with declassification. We showed that our previously proposed security condition WHAT_1 [20] provides adequate control of what can be declassified, for all schedulers that can be expressed in our scheduler model. When proposing WHAT_1, we had hoped that this condition is scheduler independent, but had no proof for this so far. Our novel security condition WHAT&WHERE provides adequate control of what can be declassified where, independent of the scheduler. Our two scheduler-independence results provide the theoretical basis for reasoning about the security of multi-threaded programs in a sound way, without having to explicitly consider the scheduler under which a program runs.

The security guarantees provided by WHAT&WHERE go far beyond a mere conjunction of the previously proposed conditions WHAT_1 and WHERE because a fine-grained, integrated control of what is declassified where is made possible.

The scheduler model (cf. Definition 1) that we used as basis in this article is sufficiently expressive to capture a wide range of schedulers, including uniform and Round-Robin schedulers. Moreover, to our knowledge, $\mathrm{WHAT}^{\mathfrak{s}}$ and $\mathrm{WHAT\&WHERE}^{\mathfrak{s}}$ offer the first scheduler-specific definitions of information-flow security that are compatible with declassification. We used these schemas as reference points for our two scheduler-independence results, and they might serve as role models for other scheduler-specific security conditions in the future.

With this article, we hope to contribute foundations that lead to a better applicability and a more wide-spread use of information-flow analysis in practice.

Acknowledgments. We thank Carroll Morgan, Jeremy Gibbons and the anonymous reviewers for their helpful comments. This work was funded by the DFG under the project RSCP (MA 3326/4-1) in the priority program RS^3 (SPP 1496).

References

1. Almeida Matos, A., Boudol, G.: On Declassification and the Non-Disclosure Policy. Journal of Computer Security 17(5), 549–597 (2009)
2. Askarov, A., Sabelfeld, A.: Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In: IEEE Symposium on Security and Privacy, pp. 207–221 (2007)
3. Askarov, A., Sabelfeld, A.: Localized Delimited Release: Combining the What and Where Dimensions of Information Release. In: Workshop on Programming Languages and Analysis for Security, pp. 53–60 (2007)
4. Askarov, A., Sabelfeld, A.: Tight Enforcement of Information-Release Policies for Dynamic Languages. In: IEEE Computer Security Foundations Symposium, pp. 43–59 (2009)
5. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive Declassification Policies and Modular Static Enforcement. In: IEEE Symposium on Security and Privacy, pp. 339–353 (2008)
6. Barthe, G., Cavadini, S., Rezk, T.: Tractable Enforcement of Declassification Policies. In: IEEE Computer Security Foundations Symposium, pp. 83–97 (2008)
7. Barthe, G., Rezk, T., Russo, A., Sabelfeld, A.: Security of Multithreaded Programs by Compilation. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 2–18. Springer, Heidelberg (2007)
8. Bell, D.E., LaPadula, L.: Secure Computer Systems: Unified Exposition and Multics Interpretation. Tech. Rep. MTR-2997, MITRE (1976)
9. Bossi, A., Piazza, C., Rossi, S.: Compositional Information Flow Security for Concurrent Programs. Journal of Computer Security 15(3), 373–416 (2007)
10. Broberg, N., Sands, D.: Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 180–196. Springer, Heidelberg (2006)
11. Broberg, N., Sands, D.: Paralocks: Role-based Information Flow Control and Beyond. In: ACM Symposium on Principles of Programming Languages, pp. 431–444 (2010)
12. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
13. Huisman, M., Worah, P., Sunesen, K.: A Temporal Logic Characterisation of Observational Determinism. In: IEEE Computer Security Foundations Workshop, pp. 3–15 (2006)
14. Jacob, J.: On the Derivation of Secure Components. In: IEEE Symposium on Security and Privacy, pp. 242–247 (1989)
15. Li, P., Zdancewic, S.: Downgrading Policies and Relaxed Noninterference. In: ACM Symposium on Principles of Programming Languages, pp. 158–170 (2005)
16. Lux, A., Mantel, H.: Declassification with Explicit Reference Points. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 69–85. Springer, Heidelberg (2009)

17. Lux, A., Mantel, H.: Who Can Declassify? In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 35–49. Springer, Heidelberg (2009)
18. Mantel, H.: Preserving Information Flow Properties under Refinement. In: IEEE Symposium on Security and Privacy, pp. 78–91 (2001)
19. Mantel, H.: Information Flow and Noninterference. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd edn., pp. 605–607. Springer (2011)
20. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 141–156. Springer, Heidelberg (2007)
21. Mantel, H., Sands, D.: Controlled Declassification based on Intransitive Noninterference. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 129–145. Springer, Heidelberg (2004)
22. Mantel, H., Sudbrock, H.: Flexible Scheduler-Independent Security. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 116–133. Springer, Heidelberg (2010)
23. McCullough, D.: Specifications for Multi-Level Security and a Hook-Up Property. In: IEEE Symposium on Security and Privacy, pp. 161–166 (1987)
24. Morgan, C.: *The Shadow Knows:* Refinement of Ignorance in Sequential Programs. In: Yu, H.-J. (ed.) MPC 2006. LNCS, vol. 4014, pp. 359–378. Springer, Heidelberg (2006)
25. Myers, A.C., Liskov, B.: Protecting Privacy using the Decentralized Label Model. ACM Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
26. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing Robust Declassification and Qualified Robustness. Journal of Computer Security 14, 157–196 (2006)
27. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. In: Gollmann, D. (ed.) ESORICS 1994. LNCS, vol. 875, pp. 33–53. Springer, Heidelberg (1994)
28. Russo, A., Sabelfeld, A.: Securing Interaction between Threads and the Scheduler in the Presence of Synchronization. Journal of Logic and Algebraic Programming 78(7), 593–618 (2009)
29. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
30. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 40–59. Springer, Heidelberg (1999)
31. Sabelfeld, A., Sands, D.: Probabilistic Noninterference for Multi-threaded Programs. In: IEEE Computer Security Foundations Workshop, pp. 200–215 (2000)
32. Sabelfeld, A., Sands, D.: Dimensions and Principles of Declassification. In: IEEE Computer Security Foundations Workshop, pp. 255–269 (2005)
33. Sabelfeld, A., Sands, D.: Declassification: Dimensions and Principles. Journal of Computer Security 17(5), 517–548 (2009)
34. Sutherland, D.: A Model of Information. In: National Computer Security Conference (1986)
35. Volpano, D., Smith, G.: Probabilistic Noninterference in a Concurrent Language. In: IEEE Computer Security Foundations Workshop, pp. 34–43 (1998)
36. Zdancewic, S., Myers, A.C.: Observational Determinism for Concurrent Program Security. In: IEEE Computer Security Foundations Workshop, pp. 29–43 (2003)

Elementary Probability Theory in the Eindhoven Style

Carroll Morgan⋆

University of New South Wales, NSW 2052 Australia
carrollm@cse.unsw.edu.au

Abstract. We extend the Eindhoven quantifier notation to elementary probability theory by adding "distribution comprehensions" to it.

Even elementary theories can be used in complicated ways, and this occurs especially when reasoning about computer programs: an instance of this is the multi-level probabilistic structures that arise in probabilistic semantics for security.

Our exemplary case study in this article is therefore the probabilistic reasoning associated with a quantitative noninterference semantics based on Hidden Markov Models of computation. But we believe the proposal here will be more generally applicable than that, and so we also revisit a number of popular puzzles, to illustrate the new notation's wider utility.

1 Context and Motivation

Conventional notations for elementary probability theory are more descriptive than calculational. They communicate ideas, but they are not algebraic (as a rule) in the sense of helping to proceed reliably from one idea to the next one: and truly effective notations are those that we can reason *with* rather than simply *about*. In our recent work on security, the conventional notations for probability became so burdensome that we felt that it was worth investigating alternative, more systematic notations for their own sake.

The Eindhoven notation was designed in the 1970's to control complexity in reasoning about programs and their associated logics: the forty years since then have shown how effective it is. But as far as we know it has not been used for probability. We have done so by working "backwards," from an application in computer security (Sec. 9.2), with the Eindhoven style as a target (Sec. 2). That is the opposite, incidentally, of reconstructing elementary probability "forwards" from first principles — also a worthwhile goal, but a different one.

We judge our proposal's success by whether it simplifies reasoning about intricate probabilistic structures in computer science and elsewhere. For that we give a small case study, based on noninterference-security semantics, both in the novel notation and in the conventional notation; and we compare them with each other (Sec. 9). We have also used the new notation more extensively [15].

⋆ We are grateful for the support of the Dutch NWO (Grant 040.11.303) and the Australian ARC (Grant DP1092464).

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 48–73, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Although the notation was developed retroactively, the account we give here is forwards, that is from the basics towards more advanced constructions. Along the way we use a number of popular puzzles as more general examples.

2 The Eindhoven Quantifier Notation, and Our Extension

In the 1970's, researchers at *THE* in Eindhoven led by EW Dijkstra proposed a uniform notation for quantifications in first-order logic, elementary set theory and related areas [3]. By $(\mathcal{Q}\,x{:}\,T \mid rng \bullet exp)$ [1] they meant that *quantifier* $\mathcal{Q}$ *binds variable* x *of type* T *within textual scope* $(\cdots)$*, that* x *is constrained to satisfy formula rng, that expression exp is evaluated for each such* x *and that those values then are combined via an associative and commutative operator related to quantifier* $\mathcal{Q}$. These examples make the uniformity evident:

$(\forall x{:}\,T \mid rng \bullet exp)$	means	for all x in T satisfying *rng* we have *exp*,
$(\exists x{:}\,T \mid rng \bullet exp)$	means	for some x in T satisfying *rng* we have *exp*,
$(\sum x{:}\,T \mid rng \bullet exp)$	means	the sum of all *exp* for x in T satisfying *rng*,
$\{x{:}\,T \mid rng \bullet exp\}$	means	the set of all *exp* for x in T satisfying *rng*.

A general shorthand applying to them all is that an omitted range *rng* defaults to *true*, and an omitted *exp* defaults to the bound variable x itself.

These (once) novel notations are not very different from the conventional ones: they contain the same ingredients because they must. Mainly they are a reordering, an imposition of consistency, and finally a making explicit of what is often implicit: bound variables, and their scope. Instead of writing $\{n{\in}\mathbb{N} \mid n{>}0\}$ for the positive natural numbers we write $\{n{:}\,\mathbb{N} \mid n{>}0\}$, omitting the "$\bullet n$" via the shorthand above; the only difference is the explicit declaration of n via a colon (as in a programming language) rather than via $n{\in}\mathbb{N}$ which, properly speaking, is a formula (with both n and $\mathbb{N}$ free) and doesn't declare anything. And instead of $\{n^2 \mid n{\in}\mathbb{N}\}$ for the square numbers, we write $\{n{:}\,\mathbb{N} \bullet n^2\}$, keeping the declaration in first position (always) and avoiding ambiguous use of the vertical bar.

In program semantics one can find general structures such as

sets of distributions for probability and nondeterminism [21, 14],

distributions of distributions,
for probabilistic noninterference security [15, 16], and even

sets of distributions of distributions to combine the two [17].

All of these are impeded by the conventional use of "Pr" to refer to probability with respect to some unnamed distribution "of interest" at the time: we need to refer to the whole distribution itself.

And when we turn to particular instances, the semantics of individual programs, we need to build functions corresponding to specific program components. The conventional "random variables" are inconvenient for this, since we

[1] The original Eindhoven style uses colons as separators; the syntax here with $\mid$ and $\bullet$ is one of many subsequent variations based on their innovation.

must invent a name for every single one: we would rather use the expressions and variables occurring in the programs themselves. In the small –but nontrivial– example of information flow (Sec. 9), borrowed from our probabilistic security work [15–17], we compare the novel notation (Sec. 9.2) to the conventional (Sec. 9.3) in those respects.

Our essential extension of the Eindhoven quantifiers was to postulate a "distribution comprehension" notation $\{\!\{s{:}\,\delta \mid \mathit{rng} \bullet \mathit{exp}\}\!\}$, intending it to mean "for all elements s in the distribution δ, conditioned by rng, make a new distribution based on evaluating exp." Thus we refer to a distribution itself (the whole comprehension), and we access random variables as expressions (the exp within). From there we worked backwards, towards primitives, to arrange that indeed the comprehension would have that meaning.

This report presents our results, but working forwards and giving simple examples as we go. Only at Def. 13 do we finally recover our conceptual starting point, a definition of the comprehension that agrees with the guesswork just above (Sec. 8.5).

3 Discrete Distributions as Enumerations

We begin with distributions written out explicitly: this is by analogy with the enumerations of sets which list their elements. The notation $f.x$ for application of function f to argument x is used from here on, except for type constructors where a distinct font allows us to reduce clutter by omitting the dot.

3.1 Finite Discrete Distributions as a Type

A finite discrete distribution δ on a set S is a function assigning to each element s in S a (non-negative) probability $\delta.s$, where the sum of all such probabilities on S is one. The fair distribution on coin-flip outcomes $\{\mathsf{H},\mathsf{T}\}$ takes both H,T to $1/2$; the distribution on die-roll outcomes $\{1{\cdot\cdot}6\}$ for a fair die gives $1/6$ for each integer n with $1{\leq}n{\leq}6$. In general we have

Definition 1. *The constructor $\mathbb{D}$ for finite discrete distributions*

The set $\mathbb{D}S$ of discrete distributions over a finite set S is the functions from S into $[0,1]$ that sum to one, that is $\{\delta{:}\, S{\rightarrow}[0,1] \mid (\sum s{:}\, S \bullet \delta.s) = 1\}$. The set S is called the *base* (type) of δ. □

In Def. 1 the omitted $|\mathit{rng}$ of $\sum$ is $|\mathit{true}$, and the omitted $\bullet\mathit{exp}$ of $\{\cdots\}$ is $\bullet\delta$. One reason for using distinct symbols $|$ and $\bullet$ is that in the default cases those symbols can be omitted as well, with no risk of ambiguity.

3.2 The Support of a Distribution

The support of a distribution is that subset of its base to each of whose elements it assigns nonzero probability; it is in an informal sense the "relevant" or "interesting" elements in the distribution. We define

Definition 2. *Support of a distribution* For distribution $\delta{:}\,\mathbb{D}S$ with base S, the support is the subset $\lceil\delta\rceil{:=}\{s{:}\,S \mid \delta.s{\neq}0\}$ of S. □

The "ceiling" notation $\lceil\cdot\rceil$ suggests the pointwise ceiling of a distribution which, as a function (Def. 1), is the characteristic function of its support.

3.3 Specialised Notation for Uniform Distributions

By analogy with set enumerations like $\{\mathsf{H},\mathsf{T}\}$, we define uniform-distribution enumerations that assign the same probability to every element in their support:

Definition 3. *Uniform-distribution enumeration* The uniform distribution over an enumerated set $\{a, b, \cdots, z\}$ is written $\{\!\{a, b, \cdots, z\}\!\}$. □

Thus for example the fair-coin distribution is $\{\!\{\mathsf{H},\mathsf{T}\}\!\}$ and the fair-die distribution is $\{\!\{1{\cdot\cdot}6\}\!\}$. (The empty $\{\!\{\}\!\}$ would be a sub-distribution [10, 21], not treated here.)

As a special case of uniform distribution we have the point distribution $\{\!\{a\}\!\}$ on some element a, assigning probability 1 to it: this is analogous to the singleton set $\{a\}$ that contains only a.

3.4 General Notation for Distribution Enumerations

For distributions that are not uniform, we attach a probability explicitly to each element. Thus we have $\{\!\{\mathsf{H}^{@\frac{2}{3}}, \mathsf{T}^{@\frac{1}{3}}\}\!\}$ for the coin that is twice as likely to give heads H as tails T, and $\{\!\{1^{@\frac{2}{9}}, 2^{@\frac{1}{9}}, 3^{@\frac{2}{9}}, 4^{@\frac{1}{9}}, 5^{@\frac{2}{9}}, 6^{@\frac{1}{9}}\}\!\}$ for the die that is twice as likely to roll odd as even (but is uniform otherwise). In general we have

Definition 4. *Distribution enumeration* We write $\{\!\{a^{@p_a}, b^{@p_b}, \cdots, z^{@p_z}\}\!\}$ for the distribution over set $\{a, b, \cdots, z\}$ that assigns probability p_a to element a etc. For well-formedness we require that $p_a{+}p_b{+}\cdots{+}p_z = 1$. □

3.5 The Support of a Distribution is a Subset of Its Base

Strictly speaking one can't tell, just by drawing samples, whether $\{\!\{\mathsf{H},\mathsf{T}\}\!\}$ represents the distribution of a fair two-sided coin, or instead represents the distribution of a three-sided coin with outcomes $\{\mathsf{H},\mathsf{T},\mathsf{E}\}$ that never lands on its edge E. Similarly we might not know whether $\{\!\{6\}\!\}$ describes a die that has the numeral 6 written on every face or a loaded die that always rolls 6.

Saying that δ is uniform *over* S means it is uniform and its support is S.

3.6 Specialised Infix Notations for Making Distributions

For distributions of support no more than two we have the special notation

Definition 5. *Doubleton distribution* For any elements a, b and $0{\leq}p{\leq}1$ we write $a\,{}_p{\oplus}\,b$ for the distribution $\{\!\{a^{@p}, b^{@1-p}\}\!\}$. □

Thus the fair-coin distribution $\{\!\!\{\mathsf{H},\mathsf{T}\}\!\!\}$ can be written $\mathsf{H}\,{}_{1/2}\!\oplus\mathsf{T}$. For the weighted sum of two distributions we have

Definition 6. *Weighted sum* For two numbers x, y and $0{\leq}p{\leq}1$ we define $x\,{}_p\!+ y{:=}\,px + (1{-}p)y$; more generally x, y can be elements of a vector space.

In particular, for two distributions $\delta, \delta'{:}\,\mathbb{D}S$ we define their weighted sum $\delta\,{}_p\!+\delta'$ by $(\delta\,{}_p\!+\delta').s := p(\delta.s) + (1{-}p)(\delta'.s)$ for all s in S. □

Thus the biased die from Sec. 3.4 can be written as $\{\!\!\{1,3,5\}\!\!\}\,{}_{2/3}\!+\{\!\!\{2,4,6\}\!\!\}$, showing at a glance that its odds and evens are uniform on their own, but that collectively the odds are twice as likely as the evens.

As simple examples of algebra we have first $x\,{}_p\!\oplus y = \{\!\!\{x\}\!\!\}\,{}_p\!+\{\!\!\{y\}\!\!\}$, and then

$$\delta\,{}_0\!+\delta' = \delta' \qquad \text{and} \quad \delta\,{}_1\!+\delta' = \delta$$
$$\text{and} \quad \lceil \delta\,{}_p\!+\delta' \rceil = \lceil \delta \rceil \cup \lceil \delta' \rceil \text{ when } 0{<}p{<}1.$$

3.7 Comparison with Conventional Notation[2]

Conventionally a distribution is over a *sample space* S, which we have called the base (Def. 1). Subsets of the sample space are *events*, and a distribution assigns a number to every event, the probability that an observation "sampled" from the sample space will be an occurrence of that event. That is, a distribution is of type $\mathbb{P}S{\to}[0,1]$ from subsets of S rather than from its elements.

With our odd-biased die in Sec. 3.4 the sample space is $S{=}\{1{\cdot\cdot}6\}$ and the probability $2/3$ of "rolled odd," that is of the event $\{1,3,5\}{\subset}S$, is twice the probability $1/3$ of "rolled even," that is of the event $\{2,4,6\}{\subset}S$.

There are "additivity" conditions placed on general distributions, among which are that the probability assigned to the union of two disjoint events should be the sum of the probabilities assigned to the events separately, that the probability assigned to all of S should be one, and that the probability assigned to the empty event should be zero.

When S is finite, the general approach specialises so that a *discrete* distribution δ acts on separate points, instead of on sets of them. The probability of any event $S'{\subset}S$ is then just $\sum_{s\in S'} \delta(s)$ from additivity.

4 Expected Values over Discrete Distributions

4.1 Definition of Expected Value as Average

If the base S of a distribution $\delta{:}\,\mathbb{D}S$ comprises numbers or, more generally, is a vector space, then the "weighted average" of the distribution is the sum of the values in S multiplied by the probability that δ assigns to each, that is $(\sum s{:}\,S \bullet \delta.s{\times}s)$. For the fair die that becomes $(1{+}2{+}3{+}4{+}5{+}6)/6 = 3^1\!/_2$; for the odd-biased die the average is $4^2\!/_3$.

[2] In these comparison sections we will use conventional notation throughout, for example writing $f(x)$ instead of $f.x$ and $\{exp \mid x{\in}S\}$ instead of $\{x{:}\,S \bullet exp\}$.

For the fair coin $\{\!\{\mathsf{H},\mathsf{T}\}\!\}$ however we have no average, since $\{\mathsf{H},\mathsf{T}\}$ has no arithmetic. We must work indirectly via a function on the base, using

Definition 7. *Expected value* By $(\mathcal{E}s{:}\,\delta \bullet exp)$ we mean the expected value of function $(\boldsymbol{\lambda}\, s \bullet exp)$ over distribution δ; it is

$$(\mathcal{E}s{:}\,\delta \bullet exp) \quad := \quad (\textstyle\sum s{:}\,\lceil\delta\rceil \bullet \delta.s{\times}exp)\ .$$ [3]

Note that *exp* is an expression in which bound variable s probably appears (though it need not). We call *exp* the *constructor*. □

For example, the expected value of the *square* of the value rolled on a fair die is $(\mathcal{E}s{:}\,\{\!\{1{\cdot\cdot}6\}\!\} \bullet s^2) = (1^2 + \cdots + 6^2)/6 = 15^1\!/_6$.

For further examples, we name a particular distribution $\Delta{:=}\,\{\!\{0,1,2\}\!\}$ and describe a notation for converting Booleans to numbers:

Definition 8. *Booleans converted to numbers* The function $[\cdot]$ takes Booleans T,F to numbers 0,1 so that $[\mathsf{T}]{:=}\,1$ and $[\mathsf{F}]{:=}\,0$.[4] □

Then we have

$$\begin{array}{rrcccl} & (\mathcal{E}s{:}\,\Delta \bullet s \,\mathbf{mod}\, 2) & = & 1/3{\times}0 + 1/3{\times}1 + 1/3{\times}0 & = & 1/3 \\ \text{and} & (\mathcal{E}s{:}\,\Delta \bullet [s{\neq}0]) & = & 1/3{\times}0 + 1/3{\times}1 + 1/3{\times}1 & = & 2/3\ , \end{array}$$

where in the second case we have used Def. 8 to convert the Boolean $s{\neq}0$ to a number. Now we can formulate the average proportion of heads shown by a fair coin as $(\mathcal{E}s{:}\,\{\!\{\mathsf{H},\mathsf{T}\}\!\} \bullet [s{=}\mathsf{H}]) = {}^1\!/_2$.

4.2 The Probability of a Subset Rather Than of a Single Element

We can use the expected value quantifier to give the aggregate probability assigned to a (sub)set of outcomes, provided we have a formula describing that set.[5] When *exp* is Boolean, we have that $(\mathcal{E}s{:}\,\delta \bullet [exp])$ is the probability assigned by δ to the whole of the set $\{s{:}\,\lceil\delta\rceil \mid exp\}$. This is because the expected value of the characteristic function of a set is equal to the probability of that set as a whole. An example of this is given at 4.3(e) below.

4.3 Abbreviation Conventions

The following are five abbreviations that we use in the sequel.

(a) If several bound variables are drawn from the same distribution, we assume they are drawn independently from separate instances of it. Thus $(\mathcal{E}x,y{:}\,\delta \bullet \cdots)$ means $(\mathcal{E}x{:}\,\delta, y{:}\,\delta \bullet \cdots)$ or equivalently $(\mathcal{E}(x,y){:}\,\delta^2 \bullet \cdots)$.

[3] Here is an example of not needing to know the base type: we simply sum over the support of δ, since the other summands will be zero anyway.

[4] We disambiguate T for *true* and T for *tails* by context.

[5] Note that those aggregate probabilities do not sum to one over all subsets of the base, since the individual elements would be counted many times.

(b) If in an expected-value quantification the *exp* is omitted, it is taken to be the bound variable standing alone (or a tuple of them, if there are several). Thus $(\mathcal{E}s{:}\,\delta)$ means $(\mathcal{E}s{:}\,\delta \bullet s)$, and more generally $(\mathcal{E}x,y{:}\,\delta)$ means $(\mathcal{E}x,y{:}\,\delta \bullet (x,y))$ with appropriate arithmetic induced on $\lceil\delta\rceil{\times}\lceil\delta\rceil$.
(c) By analogy with summation, where for a set S we abbreviate $(\sum s{:}\,S)$ by $\sum S$, we abbreviate $(\mathcal{E}s{:}\,\delta)$ by $\mathcal{E}\delta$. Thus $\mathcal{E}\Delta = \mathcal{E}\{\!\{0,1,2\}\!\} = (0{+}1{+}2)/3 = 1$.
(d) If a set is written where a distribution is expected, we assume implicitly that it is the uniform distribution over that set. Thus $\mathcal{E}\{0,1,2\} = \mathcal{E}\Delta = 1$.
(e) If a Boolean expression occurs where a number is expected, then we assume an implicit application of the conversion function $[\cdot]$ from Def. 8. Thus $(\mathcal{E}s{:}\,\{0,1,2\} \bullet s{\neq}0) = 2/3$ is the probability that a number chosen uniformly from $0,1,2$ will not be zero.

4.4 Example of Expected Value: Dice at the Fairground

Define the set D to be $\{1{\cdot\cdot}6\}$, the possible outcomes of a die roll.

At the fairground there is a tumbling cage with three fair dice inside, and a grid of six squares marked by numbers from D. You place \$1 on a square, and watch the dice tumble until they stop.

If your number appears exactly once among the dice, then you get your \$1 back, plus \$1 more; if it appears twice, you get \$2 more; if it appears thrice you get \$3 more. If it's not there at all, you lose your \$1.

Using our notation so far, your expected profit is written

$$-1 \;+\; (\mathcal{E}s_1,s_2,s_3{:}\,D \bullet (\bigvee i \bullet s_i{=}s) + (\sum i \bullet s_i{=}s)) \;, \tag{1}$$

where the initial -1 accounts for the dollar you paid to play, and the free variable s is the number of the square on which you placed it. The disjunction describes the event that you get your dollar back; and the summation describes the extra dollars you (might) get as well.

The D is converted to a uniform distribution by 4.3(d), then replicated three times by 4.3(a), independently for $s_{\{1,2,3\}}$; and the missing conversions from Boolean to 0,1 are supplied by 4.3(e).

Finally we abuse notation by writing s_i even though i is itself a (bound) variable: e.g. by $(\bigvee i \bullet s_i{=}s)$ we mean in fact $s_1{=}s \vee s_2{=}s \vee s_3{=}s$.[6]

[6] It is an abuse because in the scope of i we are using it as if it were an argument to some function $s_{(\cdot)}$ — but the name s is already used for something else. Moreover s_1,s_2,s_3 must themselves be names(not function applications) since we quantify over them with $\mathcal{E}$. Also we gave no type for i.

Although our purpose is to show how we achieve a concise presentation with precise notation, we are at the same time arguing that "to abuse, or not to abuse" should be decided on individual merits. There are times when a bit of flexibility is helpful: arguably the abuse here gains more in readability than it loses in informality.

A similar use is $(\exists i \bullet \cdots H_i \cdots)$ for the weakest precondition of a loop: this finesse avoided swamping a concise first-order presentation with (mostly unnecessary) higher-order logic throughout [2].

While the point of this example is the way in which (1) is written, it's worth pointing out that its value is approximately $-.08$, independent of s, thus an expected loss of about eight cents in the dollar every time you play and no matter which square you choose.

4.5 Comparison with Conventional Notation

Conventionally, expected values are taken over *random variables* that are functions from the sample space into a set with arithmetic, usually the reals (but more generally a vector space). Standard usage is first to define the sample space, then to define a distribution over it, and finally to define a random variable over the sample space and give it a name, say X. Then one writes $\Pr(X{=}x)$ for the probability assigned by that distribution to the event that the (real-valued) random variable X takes some (real) value x; and $\mathbf{E}(X)$ is the notation for the expected value of random variable X over the same (implicit) distribution.

In Def. 7 our random variable is $(\boldsymbol{\lambda}\, s \bullet \mathit{exp})$, and we can write it without a name since its bound variable s is already declared. Furthermore, because we give the distribution δ explicitly, we can write expressions in which the distributions are themselves expressions. As examples, we have

$$\begin{array}{rcll} (\mathcal{E}s\colon \{\!\{e\}\!\} \bullet \mathit{exp}) &=& \mathit{exp}[s\backslash e] & \text{– one-point rule} \\ (\mathcal{E}s\colon (\delta \;{}_p{+}\; \delta') \bullet \mathit{exp}) &=& (\mathcal{E}s\colon \delta \bullet \mathit{exp}) \;{}_p{+}\; (\mathcal{E}s\colon \delta' \bullet \mathit{exp}) & \text{– using Def. 6} \\ (\mathcal{E}s\colon (x \,{}_p{\oplus}\, y) \bullet \mathit{exp}) &=& \mathit{exp}[s\backslash x] \;{}_p{+}\; \mathit{exp}[s\backslash y] & \text{– from the two above,} \end{array}$$

where $\mathit{exp}[s\backslash e]$ is bound-variable-respecting replacement of s by e in exp.

5 Discrete Distributions as Comprehensions

5.1 Definition of Distribution Comprehensions

With a comprehension, a distribution is defined by properties rather than by enumeration. Just as the set comprehension $\{s\colon \lceil \Delta \rceil \bullet s^2\}$ gives the set $\{0,1,4\}$ having the property that its elements are precisely the squares of the elements of $\lceil \Delta \rceil = \{0,1,2\}$, we would expect $\{\!\{s\colon \Delta \bullet s^2\}\!\}$ to be $\{\!\{0,1,4\}\!\}$ where in this case the uniformity of the source Δ has induced uniformity in the target.

If however some of the target values "collide," because exp is not injective, then their probabilities add together: thus we have $\{\!\{s\colon \Delta \bullet s \,\mathbf{mod}\, 2\}\!\} = \{\!\{0^{@\frac{2}{3}}, 1^{@\frac{1}{3}}\}\!\} = 0 \,{}_{2/3}{\oplus}\, 1$, where target element 0 has received $1/3$ probability as $0 \,\mathbf{mod}\, 2$ and another $1/3$ as $2 \,\mathbf{mod}\, 2$.

We define distribution comprehensions by giving the probability they assign to an arbitrary element; thus

Definition 9. *Distribution comprehension.* For distribution δ and arbitrary value e of the type of exp we define

$$\{\!\!\{s\colon\delta \bullet exp\}\!\!\}.e \quad := \quad (\mathcal{E}s\colon\delta \bullet [exp{=}e])\ .$$ [7]

□

The construction is indeed a distribution on $\{s\colon\lceil\delta\rceil \bullet exp\}$ (Lem. 1 in App. C), and assigns to element e the probability that $exp{=}e$ as s ranges over $\lceil\delta\rceil$.[8]

5.2 Examples of Distribution Comprehensions

We have from Def. 9 that the probability $\{\!\!\{s\colon\Delta \bullet s \,\mathbf{mod}\, 2\}\!\!\}$ assigns to 0 is

$$\begin{array}{ll} & \{\!\!\{s\colon\Delta \bullet s \,\mathbf{mod}\, 2\}\!\!\}.0 \\ = & (\mathcal{E}s\colon\Delta \bullet [s \,\mathbf{mod}\, 2 = 0]) \\ = & 1/3{\times}[0{=}0] + 1/3{\times}[1{=}0] + 1/3{\times}[0{=}0] \\ = & 1/3{\times}1 + 1/3{\times}0 + 1/3{\times}1 \\ = & 2/3\ , \end{array}$$

and the probability $\{\!\!\{s\colon\Delta \bullet s \,\mathbf{mod}\, 2\}\!\!\}$ assigns to 1 is

$$\begin{array}{ll} & \{\!\!\{s\colon\Delta \bullet s \,\mathbf{mod}\, 2\}\!\!\}.1 \\ = & 1/3{\times}[0{=}1] + 1/3{\times}[1{=}1] + 1/3{\times}[0{=}1] \\ = & 1/3\ . \end{array}$$

Thus we have verified that $\{\!\!\{s\colon\Delta \bullet s \,\mathbf{mod}\, 2\}\!\!\} = 0\ {}_{2/3}\!\oplus 1$ as stated in Sec. 5.1.

5.3 Comparison with Conventional Notation

Conventionally one makes a target distribution from a source distribution by "lifting" some function that takes the source sample space into a target. We explain that here using the more general view of distributions as functions of *subsets* of the sample space (Sec. 3.7), rather than as functions of single elements.

If δ_X is a distribution over sample space X, and we have a function $f\colon X{\to}Y$, then distribution δ_Y over Y is defined $\delta_Y(Y'){:=}\,\delta_X(f^{-1}(Y'))$ for any subset Y' of Y. We then write $\delta_Y = f_*(\delta_X)$, and function $f_*\colon \mathbb{D}X{\to}\mathbb{D}Y$ is called the *push-forward*; it makes the *image measure* wrt. $f\colon X{\to}Y$ [5, index].

In the distribution comprehension $\{\!\!\{s\colon\delta \bullet exp\}\!\!\}$ for $\delta\colon\mathbb{D}S$, the source distribution is δ and the function f between the sample spaces is $(\boldsymbol{\lambda}\, s\colon S \bullet exp)$. The induced push-forward f_* is then the function $(\boldsymbol{\lambda}\,\delta\colon\mathbb{D}S \bullet \{\!\!\{s\colon\delta \bullet exp\}\!\!\})$.

[7] Compare $\{x\colon X \bullet exp\}{\ni}e$ defined to be $(\exists x\colon X \bullet exp{=}e)$.

[8] A similar comprehension notation is used in cryptography, for example the

$$\{s \xleftarrow{R} S; s' \xleftarrow{R} S' : exp\}$$

that in this case takes bound variables (s, s') uniformly $(\xleftarrow{R})$ from sample spaces (S, S') and, with them, makes a new distribution via a constructor expression (exp) containing those variables. We would write that as $\{\!\!\{s\colon S; s'\colon S' \bullet exp\}\!\!\}$ with the S, S' converted to uniform distributions by 4.3(d).

6 Conditional Distributions

6.1 Definition of Conditional Distributions

Given a distribution and an event, the latter a subset of possible outcomes, a conditioning of that distribution by the event is a new distribution formed by restricting attention to that event and ignoring all other outcomes. For that we have

Definition 10. *Conditional distribution.* Given a distribution δ and a "range" predicate *rng* in variable s ranging over the base of δ, the *conditional distribution of* δ *given rng* is determined by

$$\{\!\{s{:}\,\delta \mid rng\}\!\}.s' \quad := \quad \frac{(\mathcal{E}s{:}\,\delta \bullet rng \times [s{=}s'])}{(\mathcal{E}s{:}\,\delta \bullet rng)} \ ,$$

for any s' in the base of δ. We appeal to the abbreviation 4.3(e) to suppress the explicit conversion $[rng]$ on the right.[9]

The denominator must not be zero (Lem. 2 in App. C). □

In Def. 6.1 the distribution δ is initially restricted to the subset of the sample space defined by *rng* (in the numerator), potentially making a subdistribution because it no longer sums to one. It it restored to a full distribution by normalisation, the effect of dividing by its weight (the denominator).

6.2 Example of Conditional Distributions

A simple case of conditional distribution is illustrated by the uniform distribution $\varDelta = \{\!\{0,1,2\}\!\}$ we defined earlier. If we condition on the event "is not zero" we find that $\{\!\{s{:}\,\varDelta \mid s{\neq}0\}\!\} = \{\!\{1,2\}\!\}$, that when s is not zero it is equally likely to be 1 or 2. We verify this via Def. 10 and the calculation

$$\begin{array}{ll} & \{\!\{s{:}\,\varDelta \mid s{\neq}0\}\!\}.1 \\ = & (\mathcal{E}s{:}\,\{\!\{0,1,2\}\!\} \bullet [s{\neq}0] \times [s{=}1]) \ / \ (\mathcal{E}s{:}\,\{\!\{0,1,2\}\!\} \bullet [s{\neq}0]) \\ = & \frac{1}{3}/\frac{2}{3} \\ = & 1/2 \ . \end{array}$$

6.3 Comparison with Conventional Notation

Conventionally one refers to the conditional probability of an event A given some (other) event B, writing $\Pr(A|B)$ whose meaning is given by the Bayes formula $\Pr(A{\wedge}B)/\Pr(B)$. Both A, B are names (not expressions) referring to events defined in the surrounding text, and Pr refers, in the usual implicit way, to the probability distribution under consideration. Well-definedness requires that $\Pr(B)$ be nonzero.

Def. 10 with its conversions 4.3(e) explicit becomes

$$(\mathcal{E}s{:}\,\delta \bullet [s{=}s' \wedge rng]) \ / \ (\mathcal{E}s{:}\,\delta \bullet [rng]) \ ,$$

with Event A corresponding to "is equal to s'" and Event B to "satisfies *rng*."

[9] Leaving the $[\cdot]$ out enables a striking notational economy in Sec. 8.2.

7 Conditional Expectations

7.1 Definition of Conditional Expectations

We now put constructors *exp* and ranges *rng* together in a single definition of conditional expectation, generalising conditional distributions:

Definition 11. *Conditional expectation.* Given a distribution δ, predicate *rng* and expression *exp* both in variable *s* ranging over the base of δ, the *conditional expectation of exp over δ given rng* is

$$(\mathcal{E}s\colon\delta \mid rng \bullet exp) \quad := \quad \frac{(\mathcal{E}s\colon\delta \bullet rng \times exp)}{(\mathcal{E}s\colon\delta \bullet rng)} \;, \qquad [10]$$

in which the expected values on the right are in the simpler form to which Def. 7 applies, and *rng*,*exp* are converted if necessary according to 4.3(e).

The denominator must not be zero. □

7.2 Conventions for Default Range

If *rng* is omitted in $(\mathcal{E}s\colon\delta \mid rng \bullet exp)$ then it defaults to T, that is *true* as a Boolean or 1 as a number: and this agrees with Def. 7. To show that, in this section only we use $\underline{\boldsymbol{\mathcal{E}}}$ for Def. 11 and reason

$$
\begin{array}{lll}
 & (\underline{\boldsymbol{\mathcal{E}}}s\colon\delta \bullet exp) & \text{“as interpreted in Def. 11”} \\
= & (\underline{\boldsymbol{\mathcal{E}}}s\colon\delta \mid \mathsf{T} \bullet exp) & \text{“default } rng \text{ is } \mathsf{T}\text{”} \\
= & (\mathcal{E}s\colon\delta \bullet [\mathsf{T}] \times exp) \;/\; (\mathcal{E}s\colon\delta \bullet [\mathsf{T}]) & \text{“Def. 11 and 4.3(e)”} \\
= & (\mathcal{E}s\colon\delta \bullet exp) \;/\; (\mathcal{E}s\colon\delta \bullet 1) & \text{“}[\mathsf{T}]{=}1\text{”} \\
= & (\mathcal{E}s\colon\delta \bullet exp)\;. & \text{“}(\mathcal{E}s\colon\delta \bullet 1) = (\sum s\colon S \bullet \delta.s) = 1\text{”}
\end{array}
$$

More generally we observe that a nonzero range *rng* can be omitted whenever it contains no free *s*, of which “being equal to the default value T” is a special case. That is because it can be distributed out through the $(\mathcal{E}s)$ and then cancelled.

7.3 Examples of Conditional Expectations

In our first example we ask for the probability that a value chosen according to distribution $\varDelta$ will be less than two, given that it is not zero.

Using the technique of Sec. 4.2 we write $(\mathcal{E}s\colon\varDelta \mid s{\neq}0 \bullet s{<}2)$ which, via Def. 11, is equal to 1/2. Our earlier example at Sec. 6.2 also gives 1/2, the probability of being less than two in the uniform distribution $\{\!\{1, 2\}\!\}$.

Our second example is the expected value of a fair die roll, given that the outcome is odd. That is written $(\mathcal{E}s\colon D \mid s \;\mathbf{mod}\; 2 = 1)$, using the abbreviation of 4.3(b) to omit the constructor *s*. Via Def. 11 it evaluates to $(1{+}3{+}5)/3 = 3$.

[10] From (9) in Sec. 11 we will see this equivalently as $(\mathcal{E}s\colon \{\!\{s\colon\delta \mid rng\}\!\} \bullet exp)$.

7.4 Comparison with Conventional Notation

Conventionally one refers to the expected value of some random variable X given that some other random variable Y has a particular value y, writing $\mathbf{E}(X|Y{=}y)$. With X, Y and the distribution referred to by $\mathbf{E}$ having been fixed in the surrounding text, the expression's value is a function of y.

Our first example in Sec. 7.3 is more of conditional probability than of conditional expectation: we would state in the surrounding text that our distribution is Δ, that event A is "is nonzero" and event B is "is less than two." Then we would have $\Pr(A|B) = 1/2$.

In our second example, the random variable X is the identity on D, the random variable Y is the $\mathbf{mod}\,2$ function, the distribution is uniform on D and the particular value y is 1. Then we have $\mathbf{E}(X|Y{=}1) = 3$.

8 Belief Revision: *A Priori* and *A Posteriori* Reasoning

8.1 A-Priori and A-Posteriori Distributions in Conventional Style: Introduction and First Example

A priori, i.e. "before" and *a posteriori*, i.e. "after" distributions refer to situations in which a distribution is known (or believed) and then an observation is made that changes one's knowledge (or belief) in retrospect. This is sometimes known as *Bayesian belief revision.* A typical real-life example is the following.

> In a given population the incidence of a disease is believed to be one person in a thousand. There is a test for the disease that is 99% accurate. A patient who arrives at the doctor is therefore *a priori* believed to have only a 1/1,000 chance of having the disease; but then his test returns positive. What is his *a posteriori* belief that he has the disease?

The patient probably thinks the chance is now 99%. But the accepted Bayesian analysis is that one compares the probability of having the disease, and testing positive, with the probability of testing positive on its own (i.e. including false positives). That gives for the *a posteriori* belief

$$
\begin{aligned}
& \Pr(\text{has disease} \wedge \text{test positive}) \;/\; \Pr(\text{test positive}) \\
=\; & (1/1000) \times (99/100) \;/\; ((1/1000) \times (99/100) + (999/1000) \times (1/100)) \\
=\; & 99 \;/\; (99 + 999) \\
\approx\; & 9\% \;,
\end{aligned}
$$

that is less than one chance in ten, and not 99% at all. Although he is believed one hundred times more likely than before to have the disease, still it is ten times less likely than he feared.

8.2 Definition of *a Posteriori* Expectation

We begin with expectation rather than distribution, and define

Definition 12. A posteriori *expectation.* Given a distribution δ, an experimental outcome *rng* and expression *exp* both possibly containing variable s ranging over the base set of δ, the a posteriori *conditional expectation of exp over δ given rng* is $(\mathcal{E}s\colon \delta \mid rng \bullet exp)$, as in Def. 11 but without requiring *rng* to be Boolean. □

This economical reuse of the earlier definition, hinted at in Sec. 6.1, comes from interpreting *rng* not as a predicate but rather as the probability, depending on s, of observing some result. Note that since it varies with s it is not (necessarily) based on any *single* probability distribution, as we now illustrate.

8.3 Second Example of Belief Revision: Bertrand's Boxes

Suppose we have three boxes, identical in appearance and named Box 0, Box 1 and Box 2. Each one has two balls inside: Box 0 has two black balls, Box 1 has one white- and one black ball; and Box 2 has two white balls.

A box is chosen at random, and a ball is drawn randomly from it. *Given that the ball was white*, what is the chance the other ball is white as well?

Using Def. 12 we describe this probability as $(\mathcal{E}b\colon \Delta \mid b/2 \bullet b{=}2)$, exploiting the box-numbering convention to write $b/2$ for the probability of observing the event "ball is white" if drawing randomly from Box b. Since $(\sum b\colon \{0,1,2\} \bullet b/2)$ is $3/2 \neq 1$, it's clear that $b/2$ is not based on some single distribution, even though it is a probability. Direct calculation based on Def. 12 gives

$$\begin{array}{ll} & (\mathcal{E}b\colon \Delta \mid b/2 \bullet b{=}2) \\ = & (\mathcal{E}b\colon \{\!\{0,1,2\}\!\} \bullet b/2 \times [b{=}2]) \;/\; (\mathcal{E}b\colon \{\!\{0,1,2\}\!\} \bullet b/2) \\ = & \frac{1}{3}\times\frac{2}{2} \;/\; (\frac{1}{3}\times\frac{0}{2} + \frac{1}{3}\times\frac{1}{2} + \frac{1}{3}\times\frac{2}{2}) \\ = & \frac{1}{3} \;/\; \frac{1}{2} \\ = & 2/3\ . \end{array}$$

The other ball is white with probability 2/3.

8.4 Third Example of Belief Revision: The Reign in Spain

In Spain the rule of succession is currently that the next monarch is the eldest son of the current monarch, if there is a son at all: thus an elder daughter is passed over in favour of a younger son. We suppose that the current king had one sibling at the time he succeeded to the throne. What is the probability that his sibling was a brother?[11]

[11] We see this as belief revision if we start by assuming the monarch's only sibling is as likely to be male as female; when we learn that the monarch is a Spanish king, we revise our belief.

The answer to this puzzle will be of the form

$$(\mathcal{E}\,\textsf{two siblings} \mid \textsf{one is king} \bullet \textsf{the other is male})\ ,$$

and we deal with the three phrases one by one.

For two siblings we introduce two Boolean variables $c_{\{0,1\}}$, that is c for "child" and with the larger subscript 1 denoting the child with the larger age (i.e. the older one). Value T means "is male," and each Boolean will be chosen uniformly, reflecting the an assumption that births are fairly distributed between the two genders.

For the other is male we write $c_0 \wedge c_1$ since the king himself is male, and therefore his sibling is male just when they both are. We have now reached

$$(\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid \textsf{one is king} \bullet c_0 \wedge c_1)\ . \tag{2}$$

In the Spanish system, there will be a king (as opposed to a queen) just when *either* sibling is male: we conclude our "requirements analysis" with the formula

$$(\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid c_0 \vee c_1 \bullet c_0 \wedge c_1)\ . \tag{3}$$

It evaluates to 1/3 via Def. 12: in Spain, kings are more likely to have sisters.

Proceeding step-by-step as we did above allows us easily to investigate alternative situations. What would the answer be in Britain, where the eldest sibling becomes monarch regardless of gender? In that case we would start from (2) but reach the final formulation $(\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid c_1 \bullet c_0 \wedge c_1)$ instead of the Spanish formulation (3) we had before. We could evaluate this directly from Def. 12; but more interesting is to illustrate the algebraic possibilities for simplifying it:

$$\begin{array}{ll}
 & (\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid c_1 \bullet c_0 \wedge c_1) \qquad \text{“British succession”} \\
= & (\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid c_1 \bullet c_0 \wedge \mathsf{T}) \qquad \text{“c_1 is T, from the range”} \\
= & (\mathcal{E}c_0, c_1\colon \mathsf{Bool} \mid c_1 \bullet c_0) \qquad \text{“Boolean identity”} \\
= & (\mathcal{E}c_0\colon \mathsf{Bool} \mid (\mathcal{E}c_1\colon \mathsf{Bool} \bullet c_1) \bullet c_0) \qquad \text{“c_1 not free in constructor ($\bullet c_0$): see below”} \\
= & (\mathcal{E}c_0\colon \mathsf{Bool} \mid 1/2 \bullet c_0) \qquad \text{“Def. 7”} \\
= & (\mathcal{E}c_0\colon \mathsf{Bool} \bullet c_0) \qquad \text{“remove constant range: recall Sec. 7.2”} \\
= & 1/2\ . \qquad \text{“Def. 7”}
\end{array}$$

We set the above out in unusually small steps simply in order to illustrate its (intentional) similarity with normal quantifier-based calculations. The only non-trivial step was the one labelled "see below": it is by analogy with the set equality $\{s\colon S; s'\colon S' \mid rng \bullet exp\} = \{s\colon S \mid (\exists s'\colon S' \bullet rng) \bullet exp\}$ that applies when s' is not free in *exp*. We return to it in Sec. 11.

8.5 General Distribution Comprehensions

Comparison of Def. 10 and Def. 12 suggests a general form for distribution comprehensions, comprising both a range and a constructor. It is

Definition 13. *General distribution comprehensions.* Given a distribution δ, an experimental outcome *rng* in variable s that ranges over the base set of δ and a constructor *exp*, the general *a posteriori* distribution formed via that constructor is determined by

$$\{\!\{s{:}\,\delta \mid rng \bullet exp\}\!\}.e \quad := \quad (\mathcal{E}s{:}\,\delta \mid rng \bullet [exp{=}e])\ ,$$

for arbitrary e of the type of *exp*. □

Thus $\{\!\{c_0, c_1{:}\,\mathsf{Bool} \mid c_0 \vee c_1 \bullet c_0 \wedge c_1\}\!\} = \mathsf{T}\ _{1/3}\!\oplus \mathsf{F}$, giving the distribution of kings' siblings in Spain.

8.6 Comparison with Conventional Notation

Conventional notation for belief revision is similar to the conventional notation for conditional reasoning once we take the step of introducing the *joint distribution*.

In the first example, from Sec. 8.1, we would consider the joint distribution over the product space, that is

Joint sample space (Cartesian product)	Joint distribution×100,000		
		⊞	⊟
{has disease ☹, doesn't have disease ☺}	☹	1×99	1×1
× {test positive ⊞, test negative ⊟}	☺	999×1	999×99

and then the column corresponding to ⊞, i.e. test positive, assigns weights 99 and 999 to ☹ and ☺ respectively. Normalising those weights gives the distribution ☹ $_{9\%}\!\oplus$ ☺ for the *a posteriori* health of the patient.

Thus we would establish that joint distribution, in the surrounding text, as the referent of Pr, then define as random variables the two projection functions H (health) and T (test), and finally write for example Pr(H=☹|T=⊞) = 9% for the *a posteriori* probability that a positive-testing patient has the disease.

9 Use in Computer Science for Program Semantics

9.1 "Elementary" Can Still Be Intricate

By *elementary* probability theory we mean discrete distributions, usually over finite sets. Non-elementary would then include measures, and the subtle issues of measurability as they apply to infinite sets. In Sec. 9.2 we illustrate how simple computer programs can require intricate probabilistic reasoning even when restricted to discrete distributions on small finite sets.

The same intricate-though-elementary effect led to the Eindhoven notation in the first place.

A particular example is assignment statements, which are mathematically elementary: functions from state to state. Yet for *specific* program texts those functions are determined by expressions in the program variables, and they leave most of those variables unchanged: working with syntactic substitutions is a better approach [2, 3], but that can lead to complex formulae in the program logic.

Careful control of variable binding, and quantifiers, reduces the risk of reasoning errors in the logic, and can lead to striking simplifications because of the algebra that a systematic notation induces. That is what we illustrate in the following probabilistic example.

9.2 Case Study: Quantitative Noninterference Security

In this example we treat noninterference security for a program fragment, based on the mathematical structure of Hidden Markov Models [8, 11, 16].

Suppose we have a "secret" program variable h of type H whose value could be partly revealed by an assignment statement $v{:=}\,\mathit{exp}$ to a visible variable v of type V, if expression exp contains h. Although an attacker cannot see h, he can see v's final value, and he knows the program code (i.e. he knows the text of exp).

Given some known initial distribution δ in $\mathbb{D}H$ of h, how do we express what the attacker learns by executing the assignment, and how might we quantify the resulting security vulnerability? As an example we define $\delta{=}\{\!\{0,1,2\}\!\}$ to be a distribution on h in $H{=}\{0,1,2\}$, with $v{:=}\,h\ \mathbf{mod}\ 2$ assigning its parity to v of type $V{=}\{0,1\}$.

The output distribution over V that the attacker observes in variable v is

$$\{\!\{h\colon \delta \bullet \mathit{exp}\}\!\}\ , \tag{4}$$

thus in our example $\{\!\{h\colon \{\!\{0,1,2\}\!\} \bullet h\ \mathbf{mod}\ 2\}\!\}$. It equals $0\ {}_{2/3}\!\oplus 1$, showing that the attacker will observe $v{=}0$ twice as often as $v{=}1$.

The attacker is however not interested in v itself: he is interested in h. When he observes $v{=}1$ what he learns, and remembers, is that definitely $h{=}1$. But when $v{=}0$ he learns "less" because the (*a posteriori*) distribution of h in that case is $\{\!\{0,2\}\!\}$. In that case he is still not completely sure of h's value.

In our style, for the first case $v{=}1$ the *a posteriori* distribution of h is given by the conditional distribution $\{\!\{h\colon \{\!\{0,1,2\}\!\} \mid h\ \mathbf{mod}\ 2 = 1\}\!\} = \{\!\{1\}\!\}$; in the second case it is however $\{\!\{h\colon \{\!\{0,1,2\}\!\} \mid h\,\mathbf{mod}\,2 = 0\}\!\} = \{\!\{0,2\}\!\}$; and in general it would be $\{\!\{h\colon \{\!\{0,1,2\}\!\} \mid h\ \mathbf{mod}\ 2 = v\}\!\}$ where v is the observed value, either 0 or 1.

If in the example the attacker forgets v but remembers what he learned about h, then 2/3 of the time he remembers that h has distribution $\{\!\{0,2\}\!\}$, i.e. is equally likely to be 0 or 2; and 1/3 of the time he remembers that h has distribution $\{\!\{1\}\!\}$, i.e. is certainly 1. Thus what he remembers about h is

$$\{\!\{0,2\}\!\} \quad {}_{2/3}\!\oplus \quad \{\!\{1\}\!\}\ , \tag{5}$$

which is a distribution of distributions.[12] In general, what he remembers about h is the distribution of distributions Δ given by

$$\Delta \quad := \quad \{\!\{v\colon \{\!\{h\colon \delta \bullet \mathit{exp}\}\!\} \bullet \{\!\{h\colon \delta \mid \mathit{exp}{=}v\}\!\}\}\!\}\ , \tag{6}$$

because v itself has a distribution, as we noted at (4) above; and then the *a posteriori* distribution $\{\!\{h\colon \delta \mid \mathit{exp}{=}v\}\!\}$ of h is determined by that v. The attacker's lack of interest in v's actual value is reflected in v's not being free in (6).

We now show what the attacker can do with (6), his analysis Δ of the program's meaning: if he guesses optimally for h's value, with what probability

[12] In other work, we call this a *hyperdistribution* [15–17].

will he be right? For $v{=}0$ he will be right only half the time; but for $v{=}1$ he will be certain. So overall his attack will succeed with probability $\frac{1}{2}\,{}_{2/3}{\oplus}\,1 = \frac{2}{3}{\times}\frac{1}{2}+\frac{1}{3}{\times}1 = 2/3$, obtained from (5) by replacing the two distributions with the attacker's "best guess probability" for each, the maximum of the probabilities in those distributions. We say that the "vulnerability" in this example is 2/3.

For *vulnerability* in general take (6), apply the "best guess" strategy and then average over the cases: it becomes $(\mathcal{E}\eta{:}\,\Delta \bullet (\mathbf{max}\,h{:}\,H \bullet \eta.h))$, that is the maximum probability in each of the "inner" distributions η of Δ, averaged according to the "outer" probability Δ itself assigns to each.[13]

It is true that (6) appears complex if all you want is the information-theoretic vulnerability of a single assignment statement. But a more direct expression for that vulnerability is not compositional for programs generally; we need Δ-like semantics from which the vulnerability can subsequently be calculated, because they contain enough additional information for composition of meanings. We show elsewhere that (6) is necessary and sufficient for compositionality [15].

9.3 Comparison with Conventional Notation

Given the assignment statement $v{:=}\,\mathit{exp}$ as above, define random variables F for the function exp in terms of h, and I for h itself (again as a function of h, i.e. the identity).

Then we determine the observed output distribution of v from the input distribution δ of h by the push-forward of $F_*(\delta)$, from Sec. 5.3, of F over δ.

Then define function g^δ, depending on h's initial distribution δ, that gives for any value of v the conditioning of δ by the event $F{=}v$. That is $g^\delta(v){:=}\ \Pr(I|F{=}v)$ where the Pr on the right refers to δ.

Finally, the output hyperdistribution (6) of the attacker's resulting knowledge of h is given by the push-forward $g^\delta_*(F_*(\delta))$ of g^δ over $F_*(\delta)$ which, because composition distributes through push-forward, we can rewrite as $(g^\delta \circ F)_*(\delta)$.

An analogous treatment of (6) is given at (8) below, where superscript δ in g^δ here reflects the fact that δ is free in the inner comprehension there.

9.4 Comparison with *Qualitative* Noninterference Security

In a qualitative approach [19, 20] we would suppose a *set* $H{:=}\,\{0,1,2\}$ of hidden initial possibilities for h, not a distribution of them; and then we would execute the assignment $v{:=}\,h\ \mathbf{mod}\ 2$ as before. An observer's deductions are described by the set of sets $\{\{0,2\},\{1\}\}$, a demonic choice between knowing $h{\in}\{0,2\}$ and knowing $h{=}1$. The general $v{:=}\,\mathit{exp}$ gives $\{v{:}\,\{h{:}\,H \bullet \mathit{exp}\} \bullet \{h{:}\,H \mid \mathit{exp}{=}v\}\}$, which is a qualitative analogue of (6).[14]

---

[13] This is the *Bayes Vulnerability* of Δ [23].

[14] Written conventionally that becomes $\{\{h{\in}H \mid \mathit{exp}{=}v\} \mid v{\in}\{\mathit{exp} \mid h{\in}H\}\}$, where the left- and right occurrences of "|" now have different meanings. And then what does the middle one mean?

With the (extant) Eindhoven algebra of *set* comprehensions, and some calculation, that can be rewritten

$$\{h{:}\, H \bullet \{h'{:}\, H \mid exp{=}exp'\}\}\ , \tag{7}$$

where exp' is $exp[h\backslash h']$. It is the partition of H by the function $(\boldsymbol{\lambda} h{:}\, H \bullet exp)$. Analogously, with the algebra of *distribution* comprehensions (see (9) below) we can rewrite (6) to

$$\{\!\{h{:}\, \delta \bullet \{\!\{h'{:}\, H \mid exp{=}exp'\}\!\}\}\!\} \tag{8}$$

The occurrence of (7) and others similar, in our earlier qualitative security work [18, App. A], convinced us that there should be a *probabilistic* notational analogue (8) reflecting those analogies of meaning. This report has described how that was made to happen.

10 Monadic Structures and Other Related Work

The structure of the Eindhoven notation is monadic: for distributions it is the Giry monad Π on a category Mes of measurable spaces, with measurable maps as its morphisms [7]; for sets, it is the powerset monad $\mathbb{P}$ on Set. That accounts for many similarities, among which is the resemblance between (7) and (8).

The functor $\mathbb{D}$ takes a base set (actually measure space) to distributions (actually, measures) on it; and $\mathbb{D}$ applied to an arrow is the push-forward $()_*$. The unit transformation $\eta(x){:=}\,\{\!\{x\}\!\}$ forms the point distribution, and the multiply transformation $\mu.\Delta{:=}\,(\mathcal{E}\delta{:}\, \Delta \bullet \delta) = \mathcal{E}\Delta$ forms a weighted average of the distributions δ found within a distribution of distributions Δ.

Similarly, functor $\mathbb{P}$ takes a set to the set of its subsets; and $\mathbb{P}$ applied to an arrow is the relational image. The unit transformation takes x to singleton $\{x\}$, and multiply makes distributed union $(\bigcup x{:}\, X \bullet x) = \bigcup X$ from set of sets X.

There are also correspondences with monads in functional programming; and a number of functional-programming packages have been put together on that basis [22, 4, 12, 6]. The goal of those is mainly to enable probabilistic functional programming, except for the last one where the emphasis is also on a notation for reasoning.

There is an obvious connection with multisets, where the value associated with elements is a nonnegative integer, rather than a fraction (a probability) as here, and there is no one-summing requirement. There might thus be a more general notational treatment applying to sets, multisets and distributions all at once, if a unifying principle for conditioning can be found.

A notable example of other related work, but with a different background, is Hehner's *Probabilistic Perspective* [9]. A distribution there is an expression whose free variables range over a separately declared sample space: for each assignment of values to the free variables, the expression gives the probability of that assignment as an observation: thus for $n{:}\, \mathbb{N}^+$ the expression 2^{-n} is an example of a geometric distribution on the positive integers.

With a single new operator $\updownarrow$, for normalisation, and existing programming-like notations, Hehner reconstructs many familiar artefacts of probability theory

(including conditional distributions and *a posteriori* analyses), and convincingly demystifies a number of probability puzzles, including some of those treated here.

A strategic difference between our two approaches is (we believe) that Hehner's aim is in part to put elementary probability theory on a simpler, more rational footing; we believe he succeeds. In the sense of our comments in Sec. 1, he is working "forwards." As we hope Sec. 9 demonstrated, we started instead with existing probabilistic constructions (essentially Hidden Markov Models as we explain elsewhere [16]), as a program semantics for noninterference, and then worked backwards towards the Eindhoven quantifier notation. One of the senses in which we met Hehner "in the middle" is that we both identify discrete distributions as first-class objects, for Hehner a real-valued expression over free variables of a type and for us a function from that type into the reals.

In conventional probability theory that explicit treatment of distributions, i.e. giving them names and manipulating them, does not occur until one reaches either proper measures or Markov chains. For us it is (in spirit) the former; we believe part of Hehner's approach can be explained in terms of the latter.

A technical difference is our explicit treatment of free- and bound variables, a principal feature of the Eindhoven notation and one reason we chose it.

11 Summary and Prospects

We have argued that Eindhoven-style quantifier notation simplifies many of the constructions appearing in elementary probability. As evidence for this we invite comparison of the single expression (8) with the paragraphs of Sec. 9.3.

There is no space here to give a comprehensive list of calculational identities; but we mention two of them as examples of how the underlying structure mentioned above (Sec. 10) generates equalities similar to those already known in the Eindhoven notation applied to sets.

One identity is the trading rule

$$\begin{array}{ll} & (\mathcal{E} s\colon \{\!\{ s\colon \delta \mid rng' \bullet exp' \}\!\} \mid rng \bullet exp) \\ = & (\mathcal{E} s\colon \delta \mid rng' \times rng[s\backslash exp'] \bullet exp[s\backslash exp']) \ , \end{array} \qquad (9)$$

so-called because it "trades" components of an inner quantification into an outer one. Specialised to defaults for *true* for *rng* and s for exp', it gives an alternative to Def. 11. An identity similar to this took us from (6) to (8).

A second identity is the one used in Sec. 8.4, that $(\mathcal{E} s\colon \delta; s'\colon \delta' \mid rng \bullet exp)$ equals $(\mathcal{E} s\colon \delta \mid (\mathcal{E} s'\colon \delta' \bullet rng) \bullet exp)$ when s' is not free in *exp*. As noted there, this corresponds to a similar trading rule between set comprehension and existential quantification: both are notationally possible only because variable bindings are explicitly given *even when those variables are not used*. This is just what the Eindhoven style mandates.

The notations here generalise to (non-discrete) probability measures, i.e. even to non-elementary probability theory, again because of the monadic structure. For example the integral of a measurable function given as expression *exp* in a variable s on a sample space S, with respect to a measure μ, could conventionally

be written $\int exp\,\mu(\mathrm{d}s)$.[15] We write it however as$(\mathcal{E}s\colon \mu \bullet exp)$, and have access to (9)-like identities such as

$$(\mathcal{E}s\colon \{\!\{s'\colon \mu \bullet exp'\}\!\} \bullet exp) \quad = \quad (\mathcal{E}s'\colon \mu \bullet exp[s\backslash exp]) \ .$$

(See App. A for how this would be written conventionally for measures.)

We ended in Sec. 9 with an example of how the notation improves the treatment of probabilistic computer programs, particularly those presented in a denotational-semantic style and based on *Hidden Markov Models* for quantitative noninterference security [11, 16]. Although the example concludes this report, it was the starting point for the work.

Acknowledgements. Jeremy Gibbons identified functional-programming activity in this area, and shared his own recent work with us. Frits Vaandrager generously hosted our six-month stay at Radboud University in Nijmegen during 2011. The use of this notation for security (Sec. 9.2) was in collaboration with Annabelle McIver and Larissa Meinicke [15, 16, and others]. Roland Backhouse, Eric Hehner, Bart Jacobs and David Jansen made extensive and helpful suggestions; in particular Jansen suggested looking at continuous distributions (i.e. those given as a density function). Annabelle McIver gave strategic advice on the presentation. Finally, we thank the referees for their careful reading and useful comments.

References

1. Cheng, S.: A crash course on the Lebesgue integral and measure theory (December 2011), `www.gold-saucer.org/math/lebesgue/lebesgue.pdf`
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
3. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer (1990)
4. Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. Journal of Functional Programming 16, 21–34 (2006)
5. Fremlin, D.H.: Measure Theory. Torres Fremlin (2000)
6. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP, pp. 2–14. ACM (2011)
7. Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer (1981)
8. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proc. IEEE Symp. on Security and Privacy, pp. 75–86. IEEE Computer Society (1984)
9. Hehner, E.C.R.: A probability perspective. Form. Asp. Comput. 23, 391–419 (2011)
10. Jones, C., Plotkin, G.: A probabilistic powerdomain of evaluations. In: Proceedings of the IEEE 4th Annual Symposium on Logic in Computer Science, pp. 186–195. Computer Society Press, Los Alamitos (1989)

[15] Or not? We say "could" because "[there] are a number of different notations for the integral in the literature; for instance, one may find any of the following: $\int_Y s\,\mathrm{d}\mu$, $\int_Y s(x)\,\mathrm{d}\mu$, $\int_Y s(x)\mu$, $\int_Y s(x)\mu(\,\mathrm{d}x)$, or even $\int_Y s(x)\,\mathrm{d}x\ldots$" [1].

11. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Prentice Hall International (2000)
12. Kiselyov, O., Shan, C.-C.: Embedded probabilistic programming. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 360–384. Springer, Heidelberg (2009)
13. E Kowalski. Measure and integral (December 2011), www.math.ethz.ch/~kowalski/measure-integral.pdf
14. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Tech. Mono. Comp. Sci. Springer, New York (2005)
15. McIver, A., Meinicke, L., Morgan, C.: Compositional closure for bayes risk in probabilistic noninterference. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 223–235. Springer, Heidelberg (2010)
16. McIver, A., Meinicke, L., Morgan, C.: Hidden-Markov program algebra with iteration. At arXiv:1102.0333v1; to appear in Mathematical Structures in Computer Science in 2012 (2011)
17. McIver, A., Meinicke, L., Morgan, C.: A Kantorovich-monadic powerdomain for information hiding, with probability and nondeterminism. In: Proc. Logic in Computer Science, LiCS (2012)
18. Morgan, C.: Compositional noninterference from first principles. Formal Aspects of Computing, pp. 1–24 (2010), dx.doi.org/10.1007/s00165-010-0167-y
19. Morgan, C.: *The shadow knows:* refinement of ignorance in sequential programs. In: Yu, H.-J. (ed.) MPC 2006. LNCS, vol. 4014, pp. 359–378. Springer, Heidelberg (2006)
20. Morgan, C.C.: *The shadow knows:* refinement of ignorance in sequential programs. Science of Computer Programming 74(8), 629–653 (2009)
21. Morgan, C.C., McIver, A.K., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Prog. Lang. Sys. 18(3), 325–353 (1996), doi.acm.org/10.1145/229542.229547
22. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. SIGPLAN Not. 37, 154–165 (2002)
23. Smith, G.: Adversaries and Information Leaks (Tutorial). In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 383–400. Springer, Heidelberg (2008)

Appendices

A Measure Spaces

More general than discrete distributions, *measures* are used for probability over infinite sample spaces, where expected value becomes integration [5]. Here we sketch how "measure comprehensions" might appear; continuous distributions would be a special case of those.

In Riemann integration we write $\int_a^b x^2\,\mathrm{d}x$ for the integral of the real-valued squaring-function $sqr := (\boldsymbol{\lambda} x \bullet x^2)$ over the interval $[a, b]$, and in that notation the x in x^2 is bound by the quantifier $\mathrm{d}x$. The scope of the binding is from $\int$ to $\mathrm{d}x$.

In Lebesgue integration however we write $\int sqr\,\mathrm{d}\mu$ for the integral of that same function over a measure μ.

The startling difference between those two notations is the use of of the concrete syntax "d" that in Riemann integration's $\mathrm{d}x$ binds x, while for measures the μ in $\mathrm{d}\mu$ is free. To integrate the expression form of the squaring-function over μ we have to bind its x in another way: two typical approaches are $\int x^2\,\mu(\mathrm{d}x)$ and $\int x^2\,\mathrm{d}\mu(x)$ [1].[16]

An alternative is to achieve some uniformity by using $\mathrm{d}(\cdot)$ in the same way for both kinds of integrals [14]. We use $\int exp\,\mathrm{d}x$ for $\int (\boldsymbol{\lambda} x \bullet exp)$ in all cases; and the measure, or the bounds, are always found *outside* the expression, next to the integral sign $\int$. Thus we write $\int_\mu(\cdot)$ for integration over general measure μ, and then the familiar $\int_a^b(\cdot)$ is simply a typographically more attractive presentation of the special case $\int_{[a,b]}(\cdot)$ over the uniform measure on the real interval $[a, b]$.[17]

Then with $f := (\boldsymbol{\lambda} x \bullet F)$ we would have the equalities

$$\int_{\{\!\{c\}\!\}} F\,\mathrm{d}x \quad = \quad \int_{\{\!\{c\}\!\}} f \quad = \quad f.c \qquad \text{one-point rule}$$

and

$$\int_{g_* . \mu} F\,\mathrm{d}x \quad = \quad \int_{g_* . \mu} f \quad = \quad \int_\mu f \circ g \qquad \text{recall push-forward from Sec. 5.3.}$$

In the second case we equate the integral of f, over an (unnamed) measure formed by pushing function g forward over measure μ, with the integral of the functional composition $f \circ g$ over measure μ directly.

For complicated measures, unsuitable as subscripts, an alternative for the integral notation $\int_\mu exp\,\mathrm{d}x$ is the expected value $(\mathcal{E}x{:}\, \mu \bullet exp)$. The one-point rule is then written $(\mathcal{E}x{:}\, \{\!\{ exp \}\!\} \bullet F) = F[x \backslash exp]$. In the second case we have

[16] And there are more, since "[if] we want to display the argument of the integrand function, alternate notations for the integral include $\int_{x \in X} f(x)\,\mathrm{d}\mu \ldots$" [13].

[17] This is more general than probabiity measures, since the (e.g. Lebesgue) measure $b-a$ of the whole interval $[a, b]$ can exceed one.

$$(\mathcal{E}x\colon \{\!\{y\colon \mu \bullet G\}\!\} \bullet F) \quad = \quad (\mathcal{E}y\colon \mu \bullet F[x \backslash G]) \;, \tag{10}$$

where function g has become the lambda abstraction $(\boldsymbol{\lambda} y \bullet G)$. In Lem. 3 below we prove (10) for the discrete case.

B Exploiting Non-freeness in the Constructor

Here we prove the nontrivial step referred forward from Sec. 8.4: the main assumption is that s' is not free in *exp*. But should δ itself be an expression, we require that s' not be free there either.

$$
\begin{array}{ll}
 & (\mathcal{E}s\colon \delta; s'\colon \delta' \mid rng \bullet exp) \\
= & (\mathcal{E}s\colon \delta; s'\colon \delta' \bullet rng \times exp) \;/\; (\mathcal{E}s\colon \delta; s'\colon \delta' \bullet rng) \qquad \text{“Def. 11”} \\[1ex]
= & \dfrac{(\sum s\colon S; s'\colon S' \bullet \delta.s \times \delta'.s' \times rng \times exp)}{(\sum s\colon S; s'\colon S' \bullet \delta.s \times \delta'.s' \times rng)} \qquad \text{“Def. 7”} \\[2ex]
= & \dfrac{(\sum s\colon S \bullet \delta.s \times (\sum s'\colon S' \bullet \delta'.s' \times rng) \times exp)}{(\sum s\colon S \bullet \delta.s \times (\sum s'\colon S' \bullet \delta'.s' \times rng))} \qquad \text{“}s'\text{ not free in }exp\text{”} \\[2ex]
= & (\mathcal{E}s\colon \delta \bullet (\mathcal{E}s'\colon \delta' \bullet rng) \times exp) \;/\; (\mathcal{E}s\colon \delta \bullet (\mathcal{E}s'\colon \delta' \bullet rng)) \qquad \text{“Def. 7”} \\
= & (\mathcal{E}s\colon \delta \mid (\mathcal{E}s'\colon \delta' \bullet rng) \bullet exp) \;. \qquad \text{“Def. 11”}
\end{array}
$$

C Assorted Proofs Related to Definitions[18]

Lemma 1. *$\{\!\{s\colon \delta \bullet exp\}\!\}$ is a distribution on $\{s\colon \lceil \delta \rceil \bullet exp\}$*

Proof: We omit the simple proof that $0 \le \{\!\{s\colon \delta \bullet exp\}\!\}$; for the one-summing property, we write S for $\lceil \delta \rceil$ and calculate

$$
\begin{array}{lll}
 & (\sum e\colon \{s\colon S \bullet exp\} \bullet \{\!\{s\colon \delta \bullet exp\}\!\}.e) & \text{“let } e \text{ be fresh”} \\
= & (\sum e\colon \{s\colon S \bullet exp\} \bullet (\mathcal{E}s\colon \delta \bullet [exp{=}e])) & \text{“Def. 9”} \\
= & (\sum e\colon \{s\colon S \bullet exp\} \bullet (\sum s\colon S \bullet \delta.s {\times} [exp{=}e])) & \text{“Def. 7”} \\
= & (\sum s\colon S; e\colon \{s\colon S \bullet exp\} \bullet \delta.s {\times} [exp{=}e]) & \text{“merge and swap summations”} \\
= & (\sum s\colon S; e\colon \{s\colon S \bullet exp\} \mid exp{=}e \bullet \delta.s) & \text{“trading”} \\
= & (\sum s\colon S \bullet \delta.s) & \text{“one-point rule”} \\
= & 1 \;. & \text{“}\delta\text{ is a distribution”}
\end{array}
$$

□

Lemma 2. *$\{\!\{s\colon \delta \mid rng\}\!\}$ is a distribution on $\lceil \delta \rceil$ if $(\mathcal{E}s\colon \delta \bullet rng) \neq 0$*

[18] We thank Roland Backhouse for the suggestion to include the first two of these.

Proof: We omit the simple proof that $0 \le \{\!\{ s\colon \delta \mid rng \}\!\}$; for the one-summing property, we write S for $\lceil \delta \rceil$ and calculate

$$
\begin{array}{ll}
 & (\sum s'\colon S \bullet \{\!\{ s\colon \delta \mid rng \}\!\}.s') \qquad \text{“let } s' \text{ be fresh”} \\
= & (\sum s'\colon S \bullet (\mathcal{E} s\colon \delta \bullet rng \times [s{=}s']) \,/\, (\mathcal{E} s\colon \delta \bullet rng)) \qquad \text{“Def. 10”} \\
= & (\sum s'\colon S \bullet (\mathcal{E} s\colon \delta \bullet rng \times [s{=}s'])) \,/\, (\mathcal{E} s\colon \delta \bullet rng) \qquad \text{“} s' \text{ not free in denominator”} \\
= & (\mathcal{E} s\colon \delta \bullet rng) \,/\, (\mathcal{E} s\colon \delta \bullet rng) \qquad \text{“one-point rule; Def. 7”} \\
= & 1\ . \qquad \text{“}(\mathcal{E} s\colon \delta \bullet rng) {\ne} 0\text{”}
\end{array}
$$

□

Lemma 3. $(\mathcal{E} x\colon \{\!\{ y\colon \delta \bullet G \}\!\} \bullet F) = (\mathcal{E} y\colon \delta \bullet F[x \backslash G])$

This is Equation (10) in the discrete case.

Proof: Let X be the support of $\{\!\{ y\colon \delta \bullet G \}\!\}$, for which a more concise notation is given in App. D below, and let Y be the support of δ; we calculate

$$
\begin{array}{ll}
 & (\mathcal{E} x\colon \{\!\{ y\colon \delta \bullet G \}\!\} \bullet F) \\
= & (\sum x\colon X \bullet \{\!\{ y\colon \delta \bullet G \}\!\}.x \times F) \qquad \text{“Def. 7”} \\
= & (\sum x\colon X \bullet (\mathcal{E} y\colon \delta \bullet [G{=}x]) \times F) \qquad \text{“Def. 9”} \\
= & (\sum x\colon X \bullet (\sum y\colon Y \bullet \delta.y \times [G{=}x]) \times F) \qquad \text{“Def. 7”} \\
= & (\sum y\colon Y; x\colon X \bullet \delta.y \times [G{=}x] \times F) \qquad \text{“distribution of summations”} \\
= & (\sum y\colon Y \bullet \delta.y \times F[x \backslash G]) \qquad \text{“one-point rule for summation”} \\
= & (\mathcal{E} y\colon \delta \bullet F[x \backslash G])\ . \qquad \text{“Def. 7”}
\end{array}
$$

□

From Lem. 3 we have immediately an analogous equality for distributions, since distribution comprehensions are a special case of expected values: a more succinct, point-free alternative to Def. 9 and Def. 13 is given by the equality

$$
\{\!\{ s\colon \delta \mid rng \bullet exp \}\!\} \quad = \quad (\mathcal{E} s\colon \delta \mid rng \bullet \{\!\{ exp \}\!\})\ , \qquad (11)
$$

[19]

where the right-hand expected value is being taken in a vector space (of discrete distributions). This is how we simplified (6) to (8) in Sec. 9.

D Further Identities

The identities below are motivated by the first one, i.e. Sec. D.1, justifying the idea that in a comprehension with both range and constructor one can think in terms of enforcing the range as a first step, and then the constructor to what results. The identities are listed in order of increasing generality.

For conciseness in this section we use E^{new}_{old} for substitution and letters R, E instead of words rng, exp for expressions and $\lceil s\colon \delta \mid rng \bullet exp \rceil$ for the support $\lceil \{\!\{ s\colon \delta \mid rng \bullet exp \}\!\} \rceil$.

[19] from $(\mathcal{E} s\colon \delta \mid rng \bullet \{\!\{ exp \}\!\}).e = (\mathcal{E} s\colon \delta \mid rng \bullet \{\!\{ exp \}\!\}.e) = (\mathcal{E} s\colon \delta \mid rng \bullet [exp{=}e])\ .$

D.1 ————————————

$$
\begin{array}{lll}
 & (\mathcal{E}s\colon \{\!\{s\colon \delta \mid R\}\!\} \bullet E) & \\
= & (\mathcal{E}e\colon \{\!\{s\colon \delta \mid R\}\!\} \bullet E^e_s) & \text{“fresh variable } e\text{”} \\
= & (\sum e\colon \lceil s\colon \delta \mid R\rceil \bullet \{\!\{s\colon \delta \mid R\}\!\}.e \times E^e_s) & \\
= & (\sum e\colon \lceil s\colon \delta \mid R\rceil \bullet (\mathcal{E}s\colon \delta \mid R \bullet [s{=}e]) \times E^e_s) & \\
= & (\sum e\colon \lceil s\colon \delta \mid R\rceil \bullet (\mathcal{E}s\colon \delta \bullet R \times [s{=}e]) \times E^e_s/(\mathcal{E}s\colon \delta \bullet R)) & \\
= & (\sum e\colon \lceil s\colon \delta \mid R\rceil \bullet \delta.e \times R^e_s \times E^e_s/(\mathcal{E}s\colon \delta \bullet R)) & \text{“one-point rule”} \\
= & (\sum e\colon \lceil s\colon \delta \mid R\rceil \bullet \delta.e \times R^e_s \times E^e_s)/(\mathcal{E}s\colon \delta \bullet R) & \text{“}e\text{ not free in } R \text{ or } \delta\text{”} \\
= & (\mathcal{E}e\colon \delta \bullet R^e_s \times E^e_s)/(\mathcal{E}s\colon \delta \bullet R) & \text{“definition } \mathcal{E} \text{ and } \lceil s\colon \delta \mid R\rceil\text{”} \\
= & (\mathcal{E}s\colon \delta \bullet R \times E)/(\mathcal{E}s\colon \delta \bullet R) & \text{“}e\text{ not free in } R, E\text{”} \\
= & (\mathcal{E}s\colon \delta \mid R \bullet E)\ . & \text{“Def. 11”}
\end{array}
$$

D.2 ———————————— **D.1 for distributions**

$$
\begin{array}{lll}
 & \{\!\{s\colon \{\!\{s\colon \delta \mid R\}\!\} \bullet E\}\!\} & \\
= & \{\!\{s\colon \delta \mid R \bullet E\}\!\}\ . & \text{“from Sec. D.1 under the same conditions, using (11)”}
\end{array}
$$

D.3 ————————————

An elaboration of Sec. D.1 with constructor F, generalising Lem. 3.

$$
\begin{array}{lll}
 & (\mathcal{E}s\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \bullet E) & \\
= & & \text{“as for Sec. D.1...”} \\
 & (\sum e\colon \lceil s\colon \delta \mid R \bullet F\rceil \bullet (\mathcal{E}s\colon \delta \bullet R \times [F{=}e]) \times E^e_s/(\mathcal{E}s\colon \delta \bullet R)) & \\
= & & \text{“... but cannot use one-point wrt. } F\text{”} \\
 & (\sum e\colon \lceil s\colon \delta \mid R \bullet F\rceil \bullet (\sum s\colon \lceil \delta\rceil \bullet \delta.s \times R \times [F{=}e]) \times E^e_s/(\mathcal{E}s\colon \delta \bullet R)) & \\
= & (\sum s\colon \lceil \delta\rceil ; e\colon \lceil s\colon \delta \mid R \bullet F\rceil \bullet \delta.s \times R \times [F{=}e] \times E^e_s/(\mathcal{E}s\colon \delta \bullet R)) & \\
= & (\sum s\colon \lceil \delta\rceil \bullet \delta.s \times R \times E^F_s/(\mathcal{E}s\colon \delta \bullet R)) & \text{“if } \delta.s \text{ and } R \text{ both nonzero,} \\
 & & \text{then } F{\in}\lceil s\colon \delta \mid R \bullet F\rceil; \\
 & & e \text{ not free in } R\text{”} \\
= & (\sum s\colon \lceil \delta\rceil \bullet \delta.s \times R \times E^F_s)/(\mathcal{E}s\colon \delta \bullet R) & \\
= & (\mathcal{E}s\colon \delta \bullet R \times E^F_s)/(\mathcal{E}s\colon \delta \bullet R) & \\
= & (\mathcal{E}s\colon \delta \mid R \bullet E^F_s)\ . &
\end{array}
$$

D.4 ———————————— **D.3 for distributions**

$$
\begin{array}{lll}
 & \{\!\{s\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \bullet E\}\!\} & \\
= & \{\!\{s\colon \delta \mid R \bullet E^F_s\}\!\}\ . & \text{“from Sec. D.3, under the same conditions”}
\end{array}
$$

D.5 ————————————

An elaboration of Sec. D.3 with range G.

$$
\begin{array}{lll}
 & (\mathcal{E}s\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \mid G \bullet E) & \\
= & (\mathcal{E}e\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \mid G^e_s \bullet E^e_s) & \\
= & (\mathcal{E}e\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \bullet G^e_s \times E^e_s)/(\mathcal{E}e\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \bullet G^e_s) & \\
= & (\mathcal{E}s\colon \delta \mid R \bullet G^F_s \times E^F_s)/(\mathcal{E}s\colon \delta \mid R \bullet G^F_s) & \text{“Sec. D.3”} \\
= & \dfrac{(\mathcal{E}s\colon \delta \bullet R \times G^F_s \times E^F_s)/(\mathcal{E}s\colon \delta \bullet R)}{(\mathcal{E}s\colon \delta \bullet R \times G^F_s)/(\mathcal{E}s\colon \delta \bullet R)} & \text{“if } (\mathcal{E}s\colon \delta \bullet R) \text{ nonzero”} \\
= & (\mathcal{E}s\colon \delta \bullet R \times G^F_s \times E^F_s) \;/\; (\mathcal{E}s\colon \delta \bullet R \times G^F_s) & \\
= & (\mathcal{E}s\colon \delta \mid R \times G^F_s \bullet E^F_s)\ . &
\end{array}
$$

D.6 ———————————— **D.5 for distributions**

$$
\begin{array}{lll}
 & \{\!\{s\colon \{\!\{s\colon \delta \mid R \bullet F\}\!\} \mid G \bullet E\}\!\} & \\
= & \{\!\{s\colon \delta \mid R \times G^F_s \bullet E^F_s\}\!\}\ . & \text{“from Sec. D.5, under the same conditions”}
\end{array}
$$

E A Special Notation for Kernel

Expression (8) suggests that distribution δ is partitioned into equivalence classes based on equality of elements $s\colon \lceil \delta \rceil$ wrt. the function $(\boldsymbol{\lambda} s \bullet exp)$. For sets (i.e. without probability) this is a well-known construction that partitions a set, converting it into a set of pairwise-disjoint equivalence classes based on equality with respect to a function. Thus we propose

Definition 14. *Distribution kernel* The kernel of a distribution δ with respect to a range *rng* in variable s is

$$
(\mathcal{K}s\colon \delta / rng) \quad := \quad \{\!\{s\colon \Delta \bullet \{\!\{s'\colon \Delta \mid rng{=}rng[s \backslash s']\}\!\}\}\!\}\ .
$$

□

Def. 14 gives a still more compact alternative $(\mathcal{K}h\colon \delta / exp)$ for the effect of the assignment $v{:=}\,exp$ on incoming distribution δ over hidden variable h.

Scheduling and Buffer Sizing of n-Synchronous Systems

Typing of Ultimately Periodic Clocks in Lucy-n

Louis Mandel and Florence Plateau*

Laboratoire de Recherche en Informatique, Université Paris-Sud 11
Laboratoire d'Informatique de l'École Normale Supérieure, INRIA

Abstract. Lucy-n is a language for programming networks of processes communicating through *bounded buffers*. A dedicated type system, termed a clock calculus, automatically computes static schedules of the processes and the sizes of the buffers between them.

In this article, we present a new algorithm which solves the subtyping constraints generated by the clock calculus. The advantage of this algorithm is that it finds schedules for tightly coupled systems. Moreover, it does not overestimate the buffer sizes needed and it provides a way to favor either system throughput or buffer size minimization.

1 Introduction

The *n-synchronous model* [8] is a data-flow programming model. It describes networks of processes that are executed concurrently and communicate through buffers of bounded size. It combines concurrency, determinism and flexible communications. These properties are especially useful for programming multimedia applications.

A language called Lucy-n [17] has been proposed for programming in the n-synchronous model. It is essentially Lustre [5] extended with a buffer operator. Lucy-n provides a static analysis that infers the activation conditions of computation nodes and the related sizes of buffers. This analysis is in the tradition of the *clock calculus* of the synchronous data-flow languages [10]. A clock calculus is a dedicated type system that ensures that a network of processes can be executed in bounded memory. The original clock calculus ensures that a network can be executed without buffering [6]. In the synchronous languages, each flow is associated with a clock that defines the instants where data is present. The clocks are infinite binary words where the occurrence of a `1` indicates the presence of a value on the flow and the occurrence of a `0` indicates the absence of a value. Here is an example of a flow `x` and its clock:

`x`	2	5		3		7	9	4			6 ...
clock(`x`)	1	1	0	1	0	1	1	1	0	0	1 ...

* Presently at Prove & Run.

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 74–101, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The clock calculus forces each expression to satisfy a typing constraint similar to the following ($e_1 + e_2$ is the pointwise application of the addition operator +):

$$\frac{H \vdash e_1 : ct_1 \mid C_1 \qquad H \vdash e_2 : ct_2 \mid C_2}{H \vdash e_1 + e_2 : ct_3 \mid \{ct_1 \ = \ ct_2 \ = \ ct_3\} \cup C_1 \cup C_2}$$

This rule establishes that in the typing environment H, the expression $e_1 + e_2$ has a clock of type ct_3 if e_1 has a clock of type ct_1, e_2 a clock of type ct_2 and if the constraint $ct_1 = ct_2 = ct_3$ is satisfied.[1] Type equality ensures clock equality. Thus two processes producing flows of the same type can be composed without buffers.

The traditional clock calculus of synchronous languages only considers equality constraints on types; adapting the clock calculus to the n-synchronous model requires the introduction of a subtyping rule for the buffer primitive. If a flow whose clock is of type ct can be stored in a buffer of bounded size to be consumed on a clock of type ct', we say that ct is a subtype of ct', denoted $ct \ {<:} \ ct'$:

$$\frac{H \vdash e : ct \mid C}{H \vdash \texttt{buffer}(e) : ct' \mid \{ct \ {<:} \ ct'\} \cup C}$$

The clock calculus of Lucy-n considers both equality and subtyping constraints.

To solve such constraints, we have to be able to unify types ($ct_1 = ct_2$) and to verify the subtyping relation ($ct_1 \ {<:} \ ct_2$). These two operations depend very much on the clock language. One especially interesting and useful clock language can be built from *ultimately periodic binary words* which comprise a finite prefix followed by an infinite repetition of a finite pattern. An algorithm to solve constraints on the types of ultimately periodic clocks is proposed in [17]. The algorithm exploits clock abstraction [9] where the exact "shape" of clocks is forgotten in favor of simpler specifications of the presence instants of the flows: their asymptotic rate and two offsets bounding the potential delay with respect to this rate.

Type constraints on abstract clocks can be solved efficiently. But, the loss of precise information leads to over-approximations of buffer sizes. Moreover, even if a constraint system has a solution, the resolution algorithm can fail to find it because of the abstraction. Therefore, when clocks are simple, we prefer to find buffer sizes precisely, rather than quickly.

In this article, we present an algorithm to solve the constraints without clock abstraction. This problem is difficult for two reasons. First, such an algorithm must consider all the information present in the clocks. If the prefixes and periodic patterns of the words that describe the clocks are long, there may be combinatorial explosions. Second, the handling of the initial behaviors (described by the prefixes of the words) is always delicate [2] and not always addressed [1]. Dealing with the initial and periodic behaviors simultaneously is a source of complexity but, to the best of our knowledge, there is no approach that manages to treat them in separate phases.

[1] The sets C_1 and C_2 contain the constraints collected during the typing of the expressions e_1 and e_2.

A program (d) is a sequence of node and clock definitions.

d ::=	`let node` $f(pat) = e$	node definition
	| `let clock` $c = ce$	clock definition
	| $d\ d$	sequence of definitions

A pattern (pat) can be a variable or a tuple.

pat ::=	x | $(pat, \ldots, pat)$	pattern

The body of a node is defined by an expression (e).

e ::=	i	constant flow
	| x	flow variable
	| $(e, \ldots, e)$	tuple
	| $e\ op\ e$	imported operator
	| `if` e `then` e `else` e	mux operator
	| $f\ e$	node application
	| e `where rec` eqs	local definitions
	| e `fby` e	initialized delay
	| e `when` ce | e `whenot` ce	sampling
	| `merge` $ce\ e\ e$	merging
	| `buffer`(e)	buffering
eqs ::=	$pat = e$ | eqs `and` eqs	mutually recursive equations

Clock expressions (ce) are either clock names or ultimately periodic words.

$$ce ::= c \mid u(v)$$
$$u ::= \varepsilon \mid 0.u \mid 1.u$$
$$v ::= 0 \mid 1 \mid 0.v \mid 1.v$$

Fig. 1. The Lucy-n kernel

Section 2 and 3 we present the Lucy-n language and its clock calculus by way of an extended example. Section 4 introduces the properties used in Section 5, which presents an algorithm for resolving constraints. Section 6 discusses results obtained on examples and compares them with previous resolution algorithms. Finally, Section 7 concludes the article.

An extended version of the article [16] with additional details and proofs, the code of the examples, a commented implementation of the algorithm and the Lucy-n compiler are available at `http://www.lri.fr/~mandel/mpc12`.[2]

2 The Lucy-n Language

The kernel of the Lucy-n language is summarized in Figure 1. In this section, we present the language through the programming of a GSM voice encoder component. This component is a cyclic encoder. It takes as input a flow of bits

[2] While the present paper is based on [15], it presents some new results. In particular, we generalize the first version of the algorithm, which allows us to define both a semi-decidable and complete algorithm and a decidable algorithm which is complete on a well defined class of systems. Finally, we also explain how to favor either system throughput or buffer size minimization.

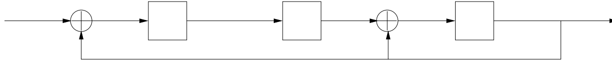


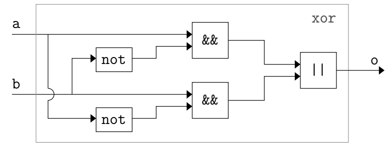
Fig. 2. Circuit for division [19] by $X^3 + X + 1$. The input flow is the sequence of fifty coefficients of the polynomial to divide. After consuming the fiftieth input bit, all the coefficients of the quotient polynomial have been produced at the output and the registers contain the coefficients of the remainder polynomial.

representing voice samples and produces an output flow that contains 3 new redundancy bits after every 50 data bits. The redundancy bits are the coefficients of the remainder of the division of the 50 bits to encode, considered as a polynomial of degree 49, by a polynomial peculiar to the encoder, here $X^3 + X + 1$.

The classical circuit to divide a polynomial is shown in Figure 2; the operator $\oplus$ represents the exclusive-or and boxes represent registers initialized to `false`.

The exclusive-or operator can be programmed as follows in Lucy-n (corresponding block diagrams are shown to the right of code samples):

```
let node xor (a, b) = o where
  rec o = (a && (not b)) || (b && (not a))
val xor : (bool * bool) -> bool
val xor :: forall 'a. ('a * 'a) -> 'a
```

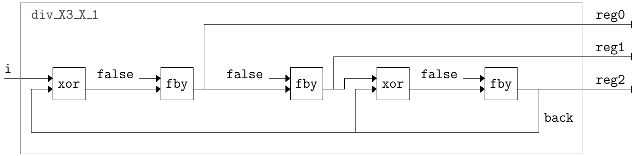


The node `xor` takes as input two flows `a` and `b` and computes the value of the output flow `o`. The value of `o` is defined by the equation `o = (a && (not b)) || (b && (not a))` where the scalar operators `&&`, `||` and `not` are applied pointwise to their input flows. Hence, if we apply the node `xor` to two flows `x` and `y`, we obtain a new flow `xor(x,y)`:

x	true	false	true	false	false	...
y	false	false	true	true	false	...
xor(x,y)	true	false	false	true	false	...

The definition of the `xor` node is followed by two facts automatically inferred by the Lucy-n compiler: the data type (*`val xor : (bool * bool) -> bool`*), and the clock type (*`val xor :: forall 'a.('a * 'a) -> 'a`*). In the clock type, the variable *`'a`* represents the activation condition of the node. The type *`'a * 'a -> 'a`* means that at each activation, the two inputs are consumed (thus, they must be present) and the output is produced instantaneously. Since *`'a`* is a polymorphic variable, this type indicates that the node can be applied to any input flows that have the same clock as each other, whatever that clock is, and that it will have to be activated according to the instants defined by this clock.

Using this new node and the initialized register primitive of Lucy-n, `fby` (followed by), we can program the circuit of Figure 2.




```
let node div_X3_X_1 i = (reg0,reg1,reg2) where
  rec reg0 = false fby (xor(i, back))
  and reg1 = false fby reg0
  and reg2 = false fby (xor(reg1, back))
  and back = reg2
val div_X3_X_1 : bool -> (bool * bool * bool)
val div_X3_X_1 :: forall 'a. 'a -> ('a * 'a * 'a)
```

The equation `reg1 = false fby reg0` means that `reg1` is equal to `false` at the first instant and equal to the preceding value of `reg0` at the following instants. Note that the definitions of flows `reg0`, `reg1`, `reg2` and `back` are mutually recursive.

In order to divide a flow of polynomials, the `div_X3_X_1` node must be modified. After the arrival of the coefficients of each polynomial, that is after every 50 input bits, the three registers must be reset to `false`. Since the content of some registers is the result of an exclusive-or between the feedback edge `back` and the preceding register (or the input flow for the first register), to reset the registers to `false`, we have to introduce three `false` values as input and three `false` values on the feedback wire, every 50 input bits.[3]

The clock type of the node `div_X3_X_1` modified accordingly is:[4]

```
val div_X3_X_1 :: forall 'a. 'a on (1^50 0^3) -> ('a * 'a * 'a)
```

The notation *(1^50 0^3)* represents the infinite repetition of the binary word $1^{50}0^3$ where 1^{50} is the concatenation of fifty 1s and 0^3 the concatenation of three 0s. To understand the type of `div_X3_X_1`, notice that *'a* (the activation rhythm of the node) defines the notion of instants for the equations of the node. The clock type of the input flow is *'a on (1^50 0^3)*. It means that the input flow has to be present during the first 50 instants, then absent for 3 instants (during which the registers are reset). Therefore, this node can compute one division every 53 instants of the rhythm *'a*. Finally, the clock type of the three outputs is *'a*, it means that the values of the registers are produced at each instant.

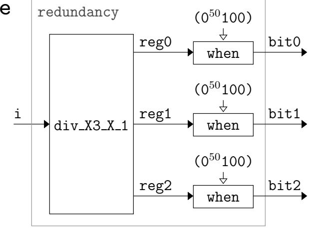
Now, to define a node `redundancy` which computes only the redundancy bits corresponding to a flow of polynomials, we sample the output of the node `div_X3_X_1`. In our implementation of the node `div_X3_X_1`, the remainder of the division is contained in the registers after the 50th input bit and output at the 51st instant. Thus, the `redundancy` node has to sample the output of `div_X3_X_1` at the 51st instant. For this, we use the `when` operator. It is parameterized by a flow and a sampling condition, and it filters the values of the flow following the pattern defined by the sampler: if the flow is absent, the output of the `when` is

[3] It is implicit, here and in the following, that such behaviors iterate repeatedly.

[4] The source code of the modified node is available at `http://www.lri.fr/~mandel/mpc12/gsm.ls.html`.

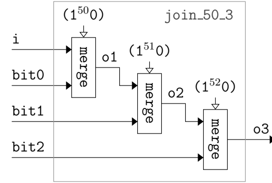
absent; if the input flow is present and the next element of the sampler is 1, the value of the flow is output; if the input flow is present and the next element of the sampler is 0, the output of the `when` is absent. To keep only the 51st element of a sequence of 53 bits, we use the sampling condition (0^{50}100):

```
let node redundancy i = (bit0,bit1,bit2) where
  rec (reg0,reg1,reg2) = div_X3_X_1 i
  and bit0 = reg0  when (0^50 100)
  and bit1 = reg1  when (0^50 100)
  and bit2 = reg2  when (0^50 100)
val redundancy : bool -> (bool * bool * bool)
val redundancy ::
 forall 'a. 'a on (1^50 0^3) ->
  ('a on (0^50 100) * 'a on (0^50 100) * 'a on (0^50 100))
```



To append 3 redundancy bits after 50 data bits, we use the `merge` operator. Its parameters are a merging condition and two flows; `merge` ce e_1 e_2 outputs the value of e_1 when ce is equal to 1 and the value of e_2 when ce is equal to 0. The flows e_1 and e_2 must be present on disjoint instants of the clock of ce: when ce is equal to 1, e_1 must be present and e_2 absent and vice versa when ce is equal to 0. Thus, to incorporate the first redundancy bit (bit0) after 50 input bits, we use the merging condition (1^{50}0) and obtain a flow of 51 bits. Then, we use the condition (1^{51}0) to incorporate the second redundancy bit, and finally the condition (1^{52}0) for the third redundancy bit.

```
let node join_50_3 (i, bit0, bit1, bit2) = o3 where
  rec o1 = merge (1^50 0) i  bit0
  and o2 = merge (1^51 0) o1 bit1
  and o3 = merge (1^52 0) o2 bit2
val join_50_3 : forall 'x.
  ('x * 'x * 'x * 'x) -> 'x
val join_50_3 :: forall 'a.
  ('a on (1^52 0) on (1^51 0) on (1^50 0) *
   'a on (1^52 0) on (1^51 0) on not (1^50 0) *
   'a on (1^52 0) on not (1^51 0) * 'a on not (1^52 0)) -> 'a
```



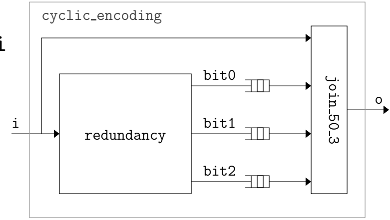
We will see in Section 4 that the clock type of `join_50_3` is equivalent to:

$\forall\alpha.(\alpha\ \texttt{on}\ (1^{50}000) \times \alpha\ \texttt{on}\ (0^{50}100) \times \alpha\ \texttt{on}\ (0^{50}010) \times \alpha\ \texttt{on}\ (0^{50}001)) \rightarrow \alpha$

This type expresses that the flow containing data must be present for the first 50 instants, and then absent for the following 3 instants. The flows containing the first, second and third redundancy bits must arrive at the 51st, 52nd, and 53rd instants respectively.

To complete the cyclic encoder, we must use the node `redundancy` to compute the three redundancy bits and the node `join_50_3` to incorporate them into the input flow. But the redundancy bits are produced at instant 51 which is too early for the `join_50_3` node which expects them successively at instants 51, 52 and 53. They must thus be stored using the `buffer` operator:

```
39 let node cyclic_encoding i = o where
40   rec (bit0, bit1, bit2) = redundancy i
41   and o = join_50_3 (i, buffer bit0,
42                         buffer bit1,
43                         buffer bit2)
```



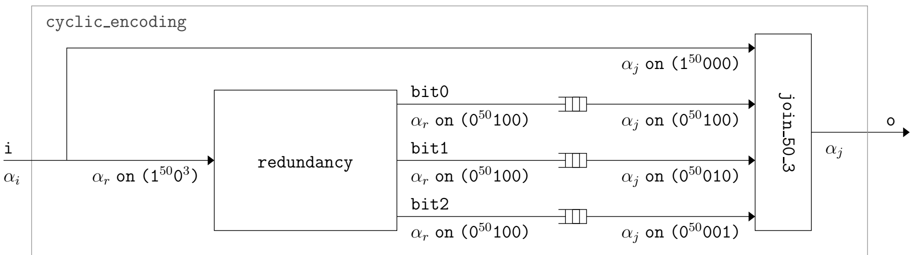

```
val cyclic_encoding : bool -> bool
val cyclic_encoding ::
  forall 'a. 'a on (1^50 0^3) -> 'a
Buffer line 41, characters 24-35: size = 0
Buffer line 42, characters 24-35: size = 1
Buffer line 43, characters 24-35: size = 1
```

The compiler automatically computes the buffer sizes required. We can see that the buffer at line 41 is not really needed, the inferred size is 0. This buffer is used for the communication of the first redundancy bit (`bit0`) between the `redundancy` node and the `join_50_3` node. This bit is produced at the 51st instant and consumed immediately. The two other redundancy bits (`bit1` and `bit2`) are also produced at the 51st instant, but they are consumed later. Thus the second bit has to be stored in a buffer of size 1 for 1 instant and the third bit has to be stored in a buffer of size 1 for 2 instants.

Notice that before calculating the buffer sizes, the compiler must infer the activation rhythm of each node. When the output of one node is consumed directly by another, i.e., when there is no buffer between them, the nodes must be activated such that the outputs of the first node are produced at the very same instants that they are to be consumed as inputs by the second node. When the output of one node is consumed by another through a buffer, the nodes must be activated such that the buffer is not read when it is empty and such that there is no infinite accumulation of data in the buffer.

3 Clock Calculus

We have seen in Section 1 that each expression in a program must satisfy a type constraint (the rules of the clock calculus are detailed in annex A). To illustrate the typing inference algorithm which collects the constraints, we return to the `cyclic_encoding` node of the previous section.



If we associate with the input `i` the clock type variable α_i, the expression `redundancy i` generates the equality constraint $\alpha_i = \alpha_r \texttt{ on } (1^{50}0^3)$. Indeed, once instantiated with a fresh variable α_r, the clock type of the node `redundancy` is $\alpha_r \texttt{ on } (1^{50}0^3) \rightarrow (\alpha_r \texttt{ on } (0^{50}100) \times \alpha_r \texttt{ on } (0^{50}100) \times \alpha_r \texttt{ on } (0^{50}100))$. Hence, the type of its input must be equal to $\alpha_r \texttt{ on } (1^{50}0^3)$. Consequently, the equation `(bit0, bit1, bit2) = redundancy i` adds to the typing environment that `bit0`, `bit1` and `bit2` are of type $\alpha_r \texttt{ on } (0^{50}100)$.

Similarly, the application of `join_50_3` adds some constraints on the types of its inputs. Once instantiated with a fresh type variable α_j, the clock type of the node `join_50_3` is $(\alpha_j \texttt{ on } (1^{50}000) \times \alpha_j \texttt{ on } (0^{50}100) \times \alpha_j \texttt{ on } (0^{50}010) \times \alpha_j \texttt{ on } (0^{50}001)) \rightarrow \alpha_j$. This type imposes the constraint that the type of the first input (here α_i, the type of the data input `i`) has to be equal to $\alpha_j \texttt{ on } (1^{50}000)$ and the types of the other inputs (here $\alpha_r \texttt{ on } (0^{50}100)$) must be, respectively, subtypes of $\alpha_j \texttt{ on } (0^{50}100)$, $\alpha_j \texttt{ on } (0^{50}010)$ and $\alpha_j \texttt{ on } (0^{50}001)$. For these last inputs, we do not impose type equality but rather only subtyping (<:) since they are consumed through buffers. The subtyping relation ensures that there are neither reads in an empty buffer nor writes in a full buffer. Finally, the equation `o = join_50_3 (...)` augments the typing environment with the information that the type of `o` is α_j, the return type of `join_50_3`.

The `cyclic_encoding` node thus has the clock type $\alpha_i \rightarrow \alpha_j$, with the following constraints:

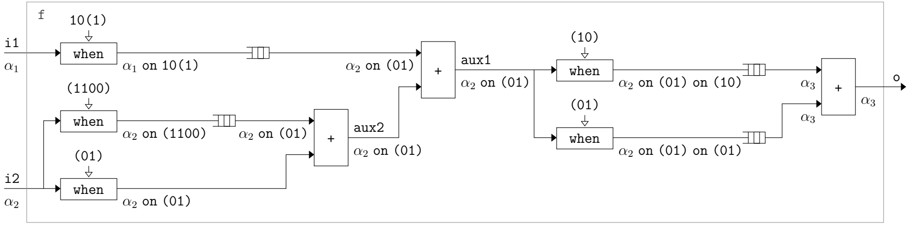
$$C = \left\{ \begin{array}{rcl} \alpha_i & = & \alpha_r \texttt{ on } (1^{50}0^3) \\ \alpha_i & = & \alpha_j \texttt{ on } (1^{50}0^3) \\ \alpha_r \texttt{ on } (0^{50}100) & <: & \alpha_j \texttt{ on } (0^{50}100) \\ \alpha_r \texttt{ on } (0^{50}100) & <: & \alpha_j \texttt{ on } (0^{50}010) \\ \alpha_r \texttt{ on } (0^{50}100) & <: & \alpha_j \texttt{ on } (0^{50}001) \end{array} \right\}$$

To finish the typing of this node and to be able to compute the buffer sizes, we have to find a solution to this constraint system, that is we must find instantiations of the variables α_i, α_r and α_j such that the constraints are always satisfied. These instantiations have to be Lucy-n clock types, i.e., of the shape: $ct ::= \alpha \mid (ct \texttt{ on } p)$ where p is an ultimately periodic binary word (formally defined in Section 4.1).

To solve the constraint system of the example, we start with the equality constraints and choose the following substitution: $\theta = \{\alpha_i \leftarrow \alpha \texttt{ on } (1^{50}0^3)\,;\alpha_r \leftarrow \alpha\,;\alpha_j \leftarrow \alpha\}$. Applying this substitution to C gives:

$$\theta(C) = \left\{ \begin{array}{rcl} \alpha \texttt{ on } (1^{50}0^3) & = & \alpha \texttt{ on } (1^{50}0^3) \\ \alpha \texttt{ on } (1^{50}0^3) & = & \alpha \texttt{ on } (1^{50}0^3) \\ \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}100) \\ \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}010) \\ \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}001) \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}100) \\ \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}010) \\ \alpha \texttt{ on } (0^{50}100) & <: & \alpha \texttt{ on } (0^{50}001) \end{array} \right\}$$

Remark 1. Notice that there is no complete greedy unification algorithm because there is no most general unifier for clock types [21]. Therefore, to be complete, a resolution algorithm must take into account all the constraints globally. As in this example greedy structural unification leads to a solution, we used it




```
let node f (i1, i2) = o where
  rec aux1 = buffer (i1 when 10(1)) + aux2
  and aux2 = buffer (i2 when (1100)) + i2 when (01)
  and o = buffer (aux1 when (10)) + buffer (aux1 when (01))
```

Fig. 3. The node `f` and its block diagram representation. The diagram is annotated with the types obtained after the resolution of equality constraints.

for the sake of conciseness. In the general case, a simple way to handle equality constraints is to consider them as two subtyping constraints ($ct_1 = ct_2 \Leftrightarrow (ct_1 <: ct_2) \wedge (ct_2 <: ct_1)$).

After transforming our constraint system to a system that contains only subtyping constraints, we notice that all the constraints depend on the same type variable. So, we apply a result from [17] to simplify the `on` operators:

$$\theta(C) \Leftrightarrow \left\{ \begin{array}{l} \texttt{(0}^{50}\texttt{100)} <: \texttt{(0}^{50}\texttt{100)} \\ \texttt{(0}^{50}\texttt{100)} <: \texttt{(0}^{50}\texttt{010)} \\ \texttt{(0}^{50}\texttt{100)} <: \texttt{(0}^{50}\texttt{001)} \end{array} \right\}$$

We will see in Section 5.1 that these constraints on words can be checked. Sometimes however, subtyping constraints are not expressed with respect to the same type variable. For example, the program of Figure 3 generates the following set of subtyping constraints where only the second constraint can be simplified:

$$C' = \left\{ \begin{array}{rcl} \alpha_1 \texttt{ on } \texttt{10(1)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(1100)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} \alpha_1 \texttt{ on } \texttt{10(1)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \end{array} \right\}$$

But, in fact, such systems can always be reduced to ones where all the constraints are expressed with respect to a single type variable. To do so, we introduce *word variables* denoted c_n and we replace each type variable α_n with α `on` c_n. Here, the application of the substitution $\theta = \{\alpha_1 \leftarrow \alpha \texttt{ on } c_1;\ \alpha_2 \leftarrow \alpha \texttt{ on } c_2;\ \alpha_3 \leftarrow \alpha \texttt{ on } c_3\}$ to system C' gives:

$$\begin{array}{rcl} \theta(C') & = & \left\{ \begin{array}{rcl} \alpha \texttt{ on } c_1 \texttt{ on } \texttt{10(1)} & <: & \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha \texttt{ on } c_3 \texttt{ on } \texttt{(1)} \\ \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha \texttt{ on } c_3 \texttt{ on } \texttt{(1)} \end{array} \right\} \\ & \Leftrightarrow & \left\{ \begin{array}{rcl} c_1 \textit{ on } \texttt{10(1)} & <: & c_2 \textit{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ c_2 \textit{ on } \texttt{(01)} \textit{ on } \texttt{(10)} & <: & c_3 \textit{ on } \texttt{(1)} \\ c_2 \textit{ on } \texttt{(01)} \textit{ on } \texttt{(01)} & <: & c_3 \textit{ on } \texttt{(1)} \end{array} \right\} \end{array}$$

This succession of operations transforms a system where the unknowns are types into a system where the unknowns are ultimately periodic binary words. The operator *on* and the relation $<:$ on binary words are defined in the following section. The algorithm that infers ultimately periodic binary words c_n to satisfy the $<:$ relation is presented in Section 5.

4 Algebra of Ultimately Periodic Words

In this section, we present the definitions and properties of ultimately periodic binary words that underlie the constraint resolution algorithm presented in Section 5. Proofs are provided in the extended version of the article [16].

4.1 Ultimately Periodic Binary Words

We write w for an infinite binary word ($w ::= \mathtt{0}w \mid \mathtt{1}w$), u or v for finite binary words ($u, v ::= \mathtt{0}u \mid \mathtt{1}u \mid \varepsilon$), $|u|$ for the size of u and $|u|_{\mathtt{1}}$ for the number of 1s it contains. The buffer analysis relies on the instants of presence of data on the flows. Therefore, it mainly manipulates indexes of 1s in the words:

Definition 1 (index of the jth 1 in w: $\mathcal{I}_w(j)$)
Let w be a binary word that contains infinitely many 1s.

$$\begin{array}{rll} \mathcal{I}_w(0) & \stackrel{def}{=} 0 & \\ \mathcal{I}_w(1) & \stackrel{def}{=} 1 & \text{if } w = \mathtt{1}w' \\ \forall j > 1,\ \mathcal{I}_w(j) & \stackrel{def}{=} 1 + \mathcal{I}_{w'}(j-1) & \text{if } w = \mathtt{1}w' \\ \forall j > 0,\ \mathcal{I}_w(j) & \stackrel{def}{=} 1 + \mathcal{I}_{w'}(j) & \text{if } w = \mathtt{0}w' \end{array}$$

For example, the index of the third 1 in $w_1 = \mathtt{11010\,11010}\ldots$ is 4, i.e., $\mathcal{I}_{w_1}(3) = 4$.

Remark 2 (increasing indexes). $\mathcal{I}_w$ is increasing: $\forall j \geq 1,\ \mathcal{I}_w(j) < \mathcal{I}_w(j+1)$.

Remark 3 (sufficient indexes). As a direct consequence of Remark 2, the index of the jth 1 is greater than or equal to j: $\forall j \geq 1, \mathcal{I}_w(j) \geq j$.

A word w can also be characterized by its cumulative function which counts the number of 1s since the beginning of w.

Definition 2 (cumulative function of w: $\mathcal{O}_w$)[5]

$$\mathcal{O}_w(0) \stackrel{def}{=} 0 \qquad \forall i \geq 1,\ \mathcal{O}_w(i) \stackrel{def}{=} \sum_{0 \leq i' \leq i} w[i']$$

In this article, we consider ultimately periodic clocks $u(v)$ which comprise a finite word u as prefix followed by the infinite repetition of a non-empty finite word v:

[5] The notation $w[i]$ represents the ith element of a word w.

Definition 3 (ultimately periodic word). $p = u(v) \overset{def}{\Leftrightarrow} p = uw$ *with* $w = vw$

For example, $p =$ `1101(110)` $=$ `1101 110 110 110` ... We use the notation $p.u$ for the prefix of a word p (e.g. (`1101(110)`)$.u =$ `1101`) and $p.v$ for its periodic pattern (e.g. (`1101(110)`)$.v =$ `110`).

An ultimately periodic binary word has an infinite number of different representations. For example, `(10)` $=$ `(1010)` $=$ `1(01)` $= \ldots$. But, there exists a normal form which is the representation where the prefix and the periodic pattern have the shortest size. For some binary operations, however, it is more convenient to put the two words in a form which is longer than the normal form. For example, for some operations we would prefer that the operands have the same size, or the same number of `1`s, or even that the number of `1`s in the first word is equal to the size of the second word.

Remark 4. We can change the shape of an ultimately periodic binary word with the following manipulations:

- Increase prefix size: $u(vv') = uv(v'v)$. For example, we can add two elements to the prefix of the word $p =$ `1101(110)` to obtain the form `1101 11(0 11)`. Increasing the size of the prefix can be used to increase the number of `1`s it contains.
- Repeat periodic pattern: $u(v) = u(v^k)$ with $k > 0$. For example, we can triple the size of the periodic pattern of $p =$ `1101(110)` (and thus triple its number of `1`s) to obtain `1101(110 110 110)`.

The following two properties ensure that a periodic word is well formed.

Remark 5 (periodicity). Two successive occurrences of the same `1` of a periodic pattern are separated by a distance equal to the size of the pattern:

$$\forall j > |p.u|_1,\ \mathcal{I}_p(j + |p.v|_1) = \mathcal{I}_p(j) + |p.v|$$

As a direct consequence of this property, the distance between any `1` in a repetition of a periodic pattern and the corresponding `1` in the first occurrence of the pattern is a multiple of the size of the pattern.

$$\forall j,\ |p.u|_1 < j \leq |p.u|_1 + |p.v|_1,\ \mathcal{I}_p(j + l \times |p.v|_1) = \mathcal{I}_p(j) + l \times |p.v|$$

For example, if $p =$ `101(10010)`, $\mathcal{I}_p(3 + 2) = \mathcal{I}_p(3) + 5$ and $\mathcal{I}_p(4 + 2 \times 2) = \mathcal{I}_p(3) + 2 \times 5$.

Remark 6 (sufficient size). The size of the periodic pattern of a word p (i.e., $|p.v|$) is greater than or equal to the number of elements between the indexes of the first and last `1` of the periodic pattern of p (for words with at least one `1` in the periodic pattern):

$$|p.v| \geq 1 + \mathcal{I}_p(|p.u|_1 + |p.v|_1) - \mathcal{I}_p(|p.u|_1 + 1)$$

For example, if $p =$ `101(10010)`, $|p.v| \geq 1 + 7 - 4$.

The rate of a word w is the proportion of `1`s in the word w:

Definition 4 (rate of p). $rate(w) = \lim_{i \to +\infty} \dfrac{\mathcal{O}_w(i)}{i}$

For an ultimately periodic binary word p, the rate is the ratio between the number of 1s and the size of its periodic pattern:

Proposition 1 (rate of p). $rate(p) = \frac{|p.v|_1}{|p.v|}$

In the following, we only consider words of non-null rate, i.e., such that the periodic pattern contains at least one 1 (these words have an infinite number of 1s and are the clocks of flows that produce values infinitely often).

4.2 Adaptability Relation

We now define the relation <: on binary words, called the *adaptability* relation. The relation $w_1 <: w_2$ holds if and only if a flow of clock w_1 can be stored in a buffer of bounded size and consumed at the rhythm of the clock w_2. It means that data does not accumulate without finite bound in the buffer, and that reads are not attempted when the buffer is empty. The adaptability relation is the conjunction of *precedence* and *synchronizability* relations. The synchronizability relation between two words w_1 and w_2 (written $w_1 \bowtie w_2$) asserts that there is a finite upper bound on the number of values present in the buffer during an execution. It states that the asymptotic numbers of reads and writes from and to the buffer are equal. The precedence relation between the words w_1 and w_2 (written $w_1 \preceq w_2$) asserts the absence of reads from an empty buffer. It states that the jth write to the buffer always occurs before the jth read.

Two words w_1 and w_2 are synchronizable if the difference between the number of occurrences of 1s in w_1 and the number of occurrences of 1s in w_2 is bounded.

Definition 5 (synchronizability $\bowtie$).

$$w_1 \bowtie w_2 \quad \overset{def}{\Leftrightarrow} \quad \exists b_1, b_2,\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$$

To test this synchronizability relation on ultimately periodic binary words, we have only to check that the periodic patterns of the two words have the same proportion of 1s. For example, $\texttt{1(1100)} \bowtie \texttt{(101001)}$ because $\frac{2}{4} = \frac{3}{6}$.

Proposition 2 (synchronizability test). $p_1 \bowtie p_2 \;\Leftrightarrow\; rate(p_1) = rate(p_2)$

A word w_1 precedes a word w_2 if the jth 1 of w_1 always occurs before or at the same time as the jth 1 of w_2.

Definition 6 (precedence $\preceq$). $w_1 \preceq w_2 \;\overset{def}{\Leftrightarrow}\; \forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_2}(j)$

To check this relation on ultimately periodic words, we only have to consider this relation until a "common" periodic behavior is reached.[6] For example, $\texttt{1(1100)} \preceq \texttt{(110100)}$ because $\mathcal{I}_{\texttt{1(1100)}}(j) \leq \mathcal{I}_{\texttt{(110100)}}(j)$ for all j such that $1 \leq j \leq 7$ and the relative behavior between the two words from the 8th 1 is

[6] A common periodic behavior of two ultimately periodic words p_1 and p_2 is defined by an index h and a size k such that $\forall j > h,\ \mathcal{I}_{p_2}(j) - \mathcal{I}_{p_1}(j) = \mathcal{I}_{p_2}(j-k) - \mathcal{I}_{p_1}(j-k)$.

exactly the same as the one from the 2nd `1`. It can be seen if we rewrite `1(1100)` as `1(110011001100)` and `(110100)` as `1(101001101001)`, the periodic patterns within these two words recommence simultaneously.

Proposition 3 (precedence test). *Consider* p_1 *and* p_2 *such that* $p_1 \bowtie p_2$*. Let* $h = \max(|p_1.u|_1, |p_2.u|_1) + \text{lcm}(|p_1.v|_1, |p_2.v|_1)$*. Then:*

$$p_1 \preceq p_2 \quad \Leftrightarrow \quad \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)$$

The intuition for the value of the bound h is the following. By Remark 4, we can adjust the respective components of p_1 and p_2 to have the same number of `1`s. We obtain two words p'_1 and p'_2 (equivalent to p_1 and p_2) such that the number of `1`s in their prefixes is $\max(|p_1.u|_1, |p_2.u|_1)$ and the number of `1`s in their periodic pattern is $\text{lcm}(|p_1.v|_1, |p_2.v|_1)$. Hence, after the traversal of h `1`s in p'_1 and p'_2 (with $h = |p'_1.u|_1 + |p'_1.v|_1 = |p'_2.u|_1 + |p'_2.v|_1$), the periodic patterns of both words restart simultaneously. And since the two words have the same rate, we are in exactly the same situation as we were at the beginning of the first traversal of the periodic patterns. So, if the condition holds until the hth `1`, it always holds.

The *adaptability* relation is the conjunction of the synchronizability and precedence relations.

Definition 7 (adaptability test). $p_1 <: p_2 \ \Leftrightarrow\ p_1 \bowtie p_2 \ \wedge\ p_1 \preceq p_2$

4.3 Buffer Size

To compute the size of a buffer, we must know the number of values that are written and read during an execution.

Consider a buffer that takes as input a flow with clock w_1, and gives as output the same flow but with clock w_2. The number of elements present at each instant i in the buffer is the difference between the number of values that have been written into it ($\mathcal{O}_{w_1}(i)$) and the number of values that have been read from it ($\mathcal{O}_{w_2}(i)$). The necessary and sufficient buffer size is the maximum number of values present in the buffer during any execution.

Definition 8 (buffer size). $\mathit{size}(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$

To compute this size on adaptable ultimately periodic binary words, we need only to consider the initial patterns of the two words before their "common" periodic behavior is reached.

Proposition 4 (buffer size)

Consider p_1 *and* p_2 *such that* $p_1 <: p_2$*.*
Let $H = \max(|p_1.u|, |p_2.u|) + \text{lcm}(|p_1.v|, |p_2.v|)$*. Then:*

$$\mathit{size}(p_1, p_2) = \max_{1 \leq i \leq H}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$$

Note that the bound H is not the same as the one of Proposition 3, because here we iterate over indexes (not over `1`s).

4.4 Sampled Clocks

The *on* operator computes the rhythm of a sampled flow. It can express the output clock of the `when` operator that keeps or suppresses values of a flow of clock w_1 depending on a condition `w2`:

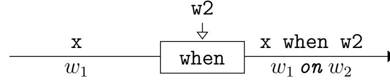


Fig. 4. If `x` has clock w_1, `x when w2` has clock w_1 *on* w_2

Definition 9 (*on* operator).
$$\begin{aligned} 0w_1 \text{ on } w_2 &\stackrel{def}{=} 0(w_1 \text{ on } w_2) \\ 1w_1 \text{ on } 1w_2 &\stackrel{def}{=} 1(w_1 \text{ on } w_2) \\ 1w_1 \text{ on } 0w_2 &\stackrel{def}{=} 0(w_1 \text{ on } w_2) \end{aligned}$$

For example, if $w_1 = 11010111...$ and $w_2 = 101100...$, then w_1 *on* $w_2 = 10010100...$ Consider the sampling of a flow `x` with clock w_1 by a condition `w2`:

x	2	5		3		7	9	4 ...
w2	1	0		1		1	0	0 ...
x when w2	2			3		7		...

w_1	1	1	0	1	0	1	1	1 ...
w_2	1	0		1		1	0	0 ...
w_1 *on* w_2	1	0	0	1	0	1	0	0 ...

At each instant, if `x` is present, that is, the corresponding element of w_1 is equal to `1`, the next element of the sampling condition w_2 is considered. If this element is `1`, then the value of `x` is sampled and the flow `x when w2` is present (w_1 *on* w_2 equals `1`). If the element is `0`, the value of `x` is not sampled and the flow `x when w2` is absent (w_1 *on* w_2 equals `0`). If `x` is absent (w_1 equals `0`), the sampling condition w_2 is not considered, and the flow `x when w2` will be absent (w_1 *on* w_2 equals `0`).

To compute the *on* operator on two ultimately periodic binary words p_1 and p_2, we first compute the size of the expected result, i.e., $|(p_1 \text{ on } p_2).u|$, the size of the prefix, and $|(p_1 \text{ on } p_2).v|$, the size of the periodic part. Then, we compute the value of the elements of the prefix and the periodic part by applying Definition 9.

Proposition 5 (computation of p_1 *on* p_2). *Let* $p_1 = u_1(v_1)$ *and* $p_2 = u_2(v_2)$*. Then* p_1 *on* $p_2 = u_3(v_3)$ *with:* $|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$

$$|v_3| = \frac{\text{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$$

and $\forall i,\ 1 \le i \le |u_3|, \quad u_3[i] = (p_1 \text{ on } p_2)[i]$
$\forall i,\ 1 \le i \le |v_3|, \quad v_3[i] = (p_1 \text{ on } p_2)[|u_3| + i]$

Intuitively, the prefix of p_1 *on* p_2 is obtained after completely processing the prefixes of p_1 and p_2. One element of p_1 is processed to produce one element of p_1 *on* p_2. Thus, the processing of the elements of $p_1.u$ for the computation of the *on* terminates at the index $|p_1.u|$. An element of p_2 is processed only when

there is a 1 in p_1. Thus the processing of the elements of $p_2.u$ terminates at the index $\mathcal{I}_{p_1}(|p_2.u|)$. Therefore, the size of the prefix of p_1 *on* p_2 is the maximum of $|p_1.u|$ and $\mathcal{I}_{p_1}(|p_2.u|)$. The size of the periodic pattern is obtained by the computation of the common period of p_1 and p_2 when p_2 is processed at the rhythm of the 1s of p_1.

The result of the *on* operation can be computed more simply for certain shapes of arguments. The simplest case is the one where the number of 1s in the prefix and in the periodic pattern of the first word are, respectively, equal to the size of the prefix and the size of the periodic pattern of the second word as in the following example:

p_1	1	1	0	1	(	1	1	1	0	0	1	1	0	)
p_2	1	0		1	(	1	0	0			1	0		)
p_1 *on* p_2	1	0	0	1	(	1	0	0	0	0	1	0	0	)

Proposition 6. *Consider p_1 and p_2 such that $|p_1.u|_1 = |p_2.u|$ and $|p_1.v|_1 = |p_2.v|$. Then:*

$$|(p_1 \text{ on } p_2).u| = |p_1.u| \qquad |(p_1 \text{ on } p_2).u|_1 = |p_2.u|_1$$
$$|(p_1 \text{ on } p_2).v| = |p_1.v| \qquad |(p_1 \text{ on } p_2).v|_1 = |p_2.v|_1$$

As explained in Remark 4, it is possible to increase the size and the number of 1s in the prefixes and periodic patterns of words. Therefore, we can always adjust the operands of the *on* such that they satisfy the assumptions of Proposition 6. Proposition 6 can be generalized to the case where the number of 1s of p_1 is increased by any multiple of the size of $p_2.v$.

Proposition 7. *Consider p_1 and p_2 such that $|p_1.u|_1 = |p_2.u| + k \times |p_2.v|$ and $|p_1.v|_1 = k' \times |p_2.v|$ with $k \in \mathbb{N}$ and $k' \in \mathbb{N} - \{0\}$. Then:*

$$|(p_1 \text{ on } p_2).u| = |p_1.u| \qquad |(p_1 \text{ on } p_2).u|_1 = |p_2.u|_1 + k \times |p_2.v|_1$$
$$|(p_1 \text{ on } p_2).v| = |p_1.v| \qquad |(p_1 \text{ on } p_2).v|_1 = k' \times |p_2.v|_1$$

Finally, to compute the index of the jth 1 of w_1 *on* w_2 ($\mathcal{I}_{w_1 \text{ on } w_2}(j)$), there is no need to compute the word w_1 *on* w_2 and then to apply the $\mathcal{I}$ function since it can be computed directly from $\mathcal{I}_{w_1}$ and $\mathcal{I}_{w_2}$.

Proposition 8 (index of the jth 1 of w_1 *on* w_2)

$$\forall j \geq 1,\ \mathcal{I}_{w_1 \text{ on } w_2}(j) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$$

Indeed, in the computation of w_1 *on* w_2, the elements of w_2 are given when there is a 1 in w_1. Therefore the index of the ith element of w_2 is at index $\mathcal{I}_{w_1}(i)$. Since the 1s of w_1 *on* w_2 are the 1s of w_2, the jth 1 of w_1 *on* w_2 is the $\mathcal{I}_{w_2}(j)$th element of w_2 and thus at the index $\mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$.

We now have all the algebraic tools needed to define an algorithm for the resolution of adaptability constraints on ultimately periodic binary words.

5 Adaptability Constraints Resolution Algorithm

We saw in Section 3 that subtyping constraints can be reduced to adaptability constraints where the unknowns are no longer types but rather ultimately

S1. Subtyping constraints: α_x on p_1 on ... **<:** α_y on p_2 on ...
⇔ { introduction of word variables c_n ;
simplification of type variables }
S2. Adaptability constraints: p_1 *on* ... <: p_2 *on* ...
⇔ { computation of *on* ; c_x *on* p'_1 *on* ... <: c_y *on* p'_2 *on* ...
simplification of p_1 <: p_2 constraints }
S3. Simplified adaptability constraints: c_x *on* p_x <: c_y *on* p_y
⇔ { for each c_n, equalization of the size of its samplers }
S4. Adjusted adaptability constraints: c_x *on* p_x <: c_y *on* p_y
⇐ { choice of the number of 1s of the c_ns ;
splitting of the adaptability constraints }
S5. Synchronizability c_x *on* $p_x \bowtie c_y$ *on* p_y
and precedence constraints: c_x *on* $p_x \preceq c_y$ *on* p_y
⇔ { simplification of synchronizability
and precedence constraints ;
introduction of well formedness constraints }
S6. Indexes of 1s and size constraints:
synchronizability: $|p_y.v|_1 \times |c_x.v| = |p_x.v|_1 \times |c_y.v|$
precedence: $\mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$
periodicity: $\mathcal{I}_{c_n}(j + l \times |c_n.v|_1) - \mathcal{I}_{c_n}(j) = l \times |c_n.v|$
sufficient size: $1 + \mathcal{I}_{c_n}(|c_n.u|_1 + |c_n.v|_1) - \mathcal{I}_{c_n}(|c_n.u|_1 + 1) \leq |c_n.v|$
sufficient indexes: $\mathcal{I}_{c_n}(j) \geq j$
increasing indexes: $\mathcal{I}_{c_n}(j') - \mathcal{I}_{c_n}(j) \geq j' - j$

Fig. 5. Summary of the subtyping constraints resolution algorithm. The form of the constraints is given for each system.

periodic binary words. This is the first step of the subtyping constraints resolution algorithm which is summarized in Figure 5.[7] This section details the remaining steps.

In Section 5.1, we explain how to simplify an adaptability constraint system (S2) to obtain a system (S3) where all the adaptability constraints have the form c_x *on* p_x <: c_y *on* p_y. In Section 5.2, we explain the transformation of adaptability constraints (S3) into a system (S6) of linear inequalities where the unknowns are the size and the indexes of 1s of the sought words. This last system can be solved using standard techniques from Integer Linear Programming, and the resulting solutions can be used to reconstruct the unknown words. In Section 5.3, we discuss the choice of the objective function for the resolution of the linear inequalities. Finally, we discuss the correctness, completeness and complexity of the algorithm.

Before the detailed explanation of the algorithm, we note that, as was the case for the equality constraints, there is no greedy algorithm for solving adaptability

[7] A detailed and commented implementation of the algorithm in OCaml is provided at http://www.lri.fr/~mandel/mpc12.

constraints. Indeed, if the words (c_1, c_2) satisfy a constraint $c_1 \ \mathit{on}\ p_1 <: c_2 \ \mathit{on}\ p_2$, then $(1^d c_1, 1^d c_2)$ and $(0^d c_1, 0^d c_2)$ also satisfy it whatever d is. Hence, contrary to a classical subtyping system, we cannot simply take the greatest word for a variable on the left of an adaptability constraint, and the smallest one for a variable on the right. In our case, the inference of values satisfying constraints must necessarily be performed globally to choose words big enough, and/or small enough to satisfy all constraints.

5.1 Constraint System Simplification

We begin by considering adaptability constraint systems. After the computation of *on* operators (Proposition 5), these systems comprise constraints of the form $p_x <: p_y$ and $c_x \ \mathit{on}\ p_x <: c_y \ \mathit{on}\ p_y$, where p_x and p_y are known words of non-null rate and c_x and c_y are unknown words.

Since a constraint of the form $p_x <: p_y$ contains no variables, its truth value cannot be altered. We need only to check that each such constraint is satisfied, which is done by applying Definition 7. If any are false, then the whole system is unsatisfiable. The true constraints can be removed from the system.

Returning to the `cyclic_encoding` example, the adaptability constraint system was:

$$\left\{ \begin{array}{l} (0^{50}100) <: (0^{50}100) \\ (0^{50}100) <: (0^{50}010) \\ (0^{50}100) <: (0^{50}001) \end{array} \right\}$$

Through the application of the adaptability test, we can check that each constraint is always satisfied. Here, after simplification, the system is empty and thus the node `cyclic_encoding` is well typed.

In the general case, after simplification all the remaining constraints contain variables. For example, the subtyping constraint system C' of Section 3 can be rewritten to the adaptability constraint system A':

$$\theta(C') \Leftrightarrow \left\{ \begin{array}{c} c_1 \ \mathit{on}\ 10(1) <: c_2 \ \mathit{on}\ (01) \\ (1100) <: (01) \\ c_2 \ \mathit{on}\ (01) \ \mathit{on}\ (10) <: c_3 \ \mathit{on}\ (1) \\ c_2 \ \mathit{on}\ (01) \ \mathit{on}\ (01) <: c_3 \ \mathit{on}\ (1) \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{l} c_1 \ \mathit{on}\ 10(1) \ \ <: c_2 \ \mathit{on}\ (01) \\ c_2 \ \mathit{on}\ (0100) <: c_3 \ \mathit{on}\ (1) \\ c_2 \ \mathit{on}\ (0001) <: c_3 \ \mathit{on}\ (1) \end{array} \right\} = A'$$

All the remaining constraints are of the form $c_x \ \mathit{on}\ p_x <: c_y \ \mathit{on}\ p_y$.

5.2 Constraint System Solving

The goal now is to solve adaptability constraint systems of the form:

$$\{c_{x_i} \ \mathit{on}\ p_{x_i} <: c_{y_i} \ \mathit{on}\ p_{y_i}\}_{i=1..\text{number of constraints}}$$

The values p_{x_i}, p_{y_i} are some known ultimately periodic binary words of non-null rate. The variables c_{x_i}, c_{y_i} are the unknowns of the system. Note that some

unknown variables can appear several times in a system like in A' (we only know that $c_{x_i} \neq c_{y_i}$ since simplification has been performed):[8]

$$\left\{ \begin{array}{l} c_1 \; \textit{on} \; p_1 \; <: c_2 \; \textit{on} \; p_2 \\ c_2 \; \textit{on} \; p_2' \; <: c_3 \; \textit{on} \; p_3 \\ c_2 \; \textit{on} \; p_2'' <: c_3 \; \textit{on} \; p_3' \end{array} \right\}$$

Solving the system means associating ultimately periodic words of non-null rate to the unknowns (c_1, c_2, c_3), such that the constraints are satisfied.

Remark 7. If solutions containing null rates are allowed, all systems have a solution. For example, the instantiation $\forall n.\ c_n = \texttt{(0)}$ is a trivial solution of all systems. A solution containing a null rate gives a system that will be executed at only a finite number of instants. We are not interested in such solutions.

An adaptability constraint $c_x \; \textit{on} \; p_x <: c_y \; \textit{on} \; p_y$ can be decomposed into a synchronizability constraint and a precedence constraint:

$$c_x \; \textit{on} \; p_x <: c_y \; \textit{on} \; p_y \Leftrightarrow \{ \text{ by Definition 7 } \}$$
$$(c_x \; \textit{on} \; p_x \bowtie c_y \; \textit{on} \; p_y) \; \wedge \; (c_x \; \textit{on} \; p_x \preceq c_y \; \textit{on} \; p_y)$$

The synchronizability constraint can itself be rewritten:

$$c_x \; \textit{on} \; p_x \bowtie c_y \; \textit{on} \; p_y \Leftrightarrow \{ \text{ by Proposition 2 and Proposition 1 } \}$$
$$\frac{|(c_x \; \textit{on} \; p_x).v|_1}{|(c_x \; \textit{on} \; p_x).v|} = \frac{|(c_y \; \textit{on} \; p_y).v|_1}{|(c_y \; \textit{on} \; p_y).v|}$$

As can the precedence constraint:

$$c_x \; \textit{on} \; p_x \preceq c_y \; \textit{on} \; p_y \Leftrightarrow \{ \text{ by Proposition 3 and Proposition 8 } \}$$
$$\forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$$

with $h = \max(|(c_x \; \textit{on} \; p_x).u|_1, |(c_y \; \textit{on} \; p_y).u|_1) + \text{lcm}(|(c_x \; \textit{on} \; p_x).v|_1, |(c_y \; \textit{on} \; p_y).v|_1)$

We are thus interested in the size of, and the number of 1s in, the prefixes and periodic patterns of $c_x \; \textit{on} \; p_x$ and of $c_y \; \textit{on} \; p_y$ forms.

We have seen in Section 4 (Proposition 7) that the size and number of 1s in the prefix and in the periodic pattern of $c_n \; \textit{on} \; p_n$ can easily be expressed as a function of the size and number of 1s in the prefixes and periodic patterns of c_n and p_n in the following case: $|c_n.u|_1 = |p_n.u| + k \times |p_n.v|$ and $|c_n.v|_1 = k' \times |p_n.v|$.

To put the system into this "simple" form, we adjust the known words of the system to satisfy the property: for any unknown c_n, all its samplers[9] p_n, p_n', ... have the same prefix size ($|p_n.u| = |p_n'.u| = \ldots$) and the same periodic pattern size ($|p_n.v| = |p_n'.v| = \ldots$). This operation is always possible (thanks to Remark 4) and does not change the semantics of the system.

We can then choose the number of 1s in the unknown c_n.

Choice 1 (number of 1s in the c_n). *Let $k \in \mathbb{N}$ and $k' \in \mathbb{N} - \{0\}$.*

$$\begin{array}{lcll} |c_n.u|_1 &=& |p_n.u| + k \times |p_n.v| & (= \; |p_n'.u| + k \times |p_n'.v| \; = \; \ldots) \\ |c_n.v|_1 &=& k' \times |p_n.v| & (= \; k' \times |p_n'.v| \; = \; \ldots) \end{array}$$

where p_n, p_n', ... are the samplers of c_n.

[8] We choose the same index for a variable c_n and the words p_n, p_n', ... that sample c_n.

[9] A word p_n is a *sampler* of c_n if $c_n \; \textit{on} \; p_n$ is in the constraint system.

Remark 8. The algorithm is parameterized by constants k and k', which restrict the number of 1s in any solution and may thus lead to failures in the resolution of constraints. We will discuss this choice in Section 5.4.

This choice allows us to express the size and the number of 1s in the prefixes and periodic patterns of the c_n *on* p_n in terms of those of c_n and p_n. Hence, a synchronizability constraint becomes:

$$c_x \text{ on } p_x \bowtie c_y \text{ on } p_y \Leftrightarrow \{ \text{ by Proposition 7 } \}$$

$$\frac{k' \times |p_x.v|_1}{|c_x.v|} = \frac{k' \times |p_y.v|_1}{|c_y.v|}$$

$$\Leftrightarrow |p_y.v|_1 \times |c_x.v| = |p_x.v|_1 \times |c_y.v| \tag{1}$$

And a precedence constraint becomes:

$$c_x \text{ on } p_x \preceq c_y \text{ on } p_y \Leftrightarrow \{ \text{ by Proposition 7 } \}$$

$$\forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$$

$$\text{with } h = \max(|p_x.u|_1 + k \times |p_x.v|_1, |p_y.u|_1 + k \times |p_y.v|_1) + \text{lcm}(k' \times |p_x.v|_1, k' \times |p_y.v|_1) \tag{2}$$

For example, we can adjust the system A' such that all the samplers of a particular variable have the same size:

$$A' = \left\{ \begin{array}{l} c_1 \text{ on } \texttt{10(1)} <: c_2 \text{ on } \texttt{(01)} \\ c_2 \text{ on } \texttt{(0100)} <: c_3 \text{ on } \texttt{(1)} \\ c_2 \text{ on } \texttt{(0001)} <: c_3 \text{ on } \texttt{(1)} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{l} c_1 \text{ on } \texttt{10(1)} <: c_2 \text{ on } \texttt{(0101)} \\ c_2 \text{ on } \texttt{(0100)} <: c_3 \text{ on } \texttt{(1)} \\ c_2 \text{ on } \texttt{(0001)} <: c_3 \text{ on } \texttt{(1)} \end{array} \right\}$$

Then, we choose the number of 1s for the c_ns to be equal to the size of the respective samplers:

$$\begin{array}{lll} |c_1.u|_1 = 2 + k \times 1 & |c_2.u|_1 = 0 + k \times 4 & |c_3.u|_1 = 0 + k \times 1 \\ |c_1.v|_1 = k' \times 1 & |c_2.v|_1 = k' \times 4 & |c_3.v|_1 = k' \times 1 \end{array}$$

By Formula (1), the synchronizability constraints become a system of linear equations on the size of the periodic patterns of the c_ns:

$$\left\{ \begin{array}{r} |\texttt{(0101)}.v|_1 \times |c_1.v| = |\texttt{10(1)}).v|_1 \times |c_2.v| \\ |\texttt{(1)}.v|_1 \times |c_2.v| = |\texttt{(0100)}.v|_1 \times |c_3.v| \\ |\texttt{(1)}.v|_1 \times |c_2.v| = |\texttt{(0001)}.v|_1 \times |c_3.v| \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{r} 2 \times |c_1.v| = |c_2.v| \\ |c_2.v| = |c_3.v| \\ |c_2.v| = |c_3.v| \end{array} \right\} \tag{Sync}$$

By Formula (2), if we choose the constants k and k' to be equal to 0 and 1, the precedence constraints become a system of linear inequalities on the indexes of 1s in the c_ns:

$$\left\{ \begin{array}{ll} \forall j,\ 1 \leq j \leq 3, & \mathcal{I}_{c_1}(\mathcal{I}_{\texttt{10(1)}}(j)) \leq \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0101)}}(j)) \\ \forall j,\ 1 \leq j \leq 1, & \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0100)}}(j)) \leq \mathcal{I}_{c_3}(\mathcal{I}_{\texttt{(1)}}(j)) \\ \forall j,\ 1 \leq j \leq 1, & \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0001)}}(j)) \leq \mathcal{I}_{c_3}(\mathcal{I}_{\texttt{(1)}}(j)) \end{array} \right\}$$

which is equivalent to the following system after the computation of the $\mathcal{I}_{p_n}(j)$:

$$\left\{ \begin{array}{l} \mathcal{I}_{c_1}(1) \leq \mathcal{I}_{c_2}(2) \\ \mathcal{I}_{c_1}(3) \leq \mathcal{I}_{c_2}(4) \\ \mathcal{I}_{c_1}(4) \leq \mathcal{I}_{c_2}(6) \\ \mathcal{I}_{c_2}(2) \leq \mathcal{I}_{c_3}(1) \\ \mathcal{I}_{c_2}(4) \leq \mathcal{I}_{c_3}(1) \end{array} \right\} \tag{Prec}$$

Now, to give a value to each unknown c_n, we must find its size (satisfying *Sync*) and the positions of its `1`s (satisfying *Prec*). Hence, the sizes $|c_n.v|$ and the indexes $\mathcal{I}_{c_n}(j)$ will no longer be considered as function applications, but rather as the new unknowns of the problem. These new unknowns must also satisfy the constraints of Remarks 2, 3, 5 and 6 which ensure that the solution will be a well formed ultimately periodic binary word. So, we have to augment *Sync* and *Prec* with the following four sets of constraints:[10]

Periodicity: $Per = \{\mathcal{I}_{c_n}(j + l \times |c_n.v|_1) - \mathcal{I}_{c_n}(j) = l \times |c_n.v|\}_{\mathcal{I}_{c_n}(j+l\times|c_n.v|_1) \in Prec \,\wedge\, |p.u|_1 < j \leq |p.u|_1 + |p.v|_1}$

Sufficient size: $Size = \{1 + \mathcal{I}_{c_n}(|c_n.u|_1 + |c_n.v|_1) - \mathcal{I}_{c_n}(|c_n.u|_1 + 1) \leq |c_n.v|\}$

Sufficient indexes: $Init = \{\mathcal{I}_{c_n}(j) \geq j\}_{\mathcal{I}_{c_n}(j) \in Prec \cup Per \cup Size}$

Increasing indexes: $Incr = \{\mathcal{I}_{c_n}(j') - \mathcal{I}_{c_n}(j) \geq j' - j\}_{(\mathcal{I}_{c_n}(j), \mathcal{I}_{c_n}(j')) \in Prec \cup Per \cup Size}$

Finally, we can use a generic solver for Integer Linear Programming (ILP) problems to solve the system: $S = Sync \cup Prec \cup Per \cup Size \cup Init \cup Incr$.

Applying a solver to the system associated with A' produces the results:

$$
\begin{array}{lllll}
|c_1.v| = 2 & \mathcal{I}_{c_1}(1) = 1 & \mathcal{I}_{c_1}(3) = 3 & \mathcal{I}_{c_1}(4) = 5 & \\
|c_2.v| = 4 & \mathcal{I}_{c_2}(1) = 1 & \mathcal{I}_{c_2}(2) = 2 & \mathcal{I}_{c_2}(4) = 4 & \mathcal{I}_{c_2}(6) = 6 \\
|c_3.v| = 4 & \mathcal{I}_{c_3}(1) = 4 & & &
\end{array}
$$

Thanks to this information and the number of `1`s in the prefixes and periodic patterns of the c_ns chosen previously, we can build the following solution to the A' system: $c_1 =$ `11(10)` $\quad c_2 =$ `(1111)` $=$ `(1)` $\quad c_3 =$ `000(1000)` $=$ `(0`^{3}`1)`.

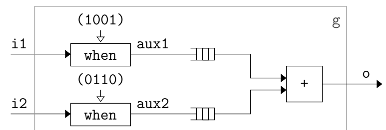
We now know all the clock types of the system of Figure 3. The result gives us the clock type of the node `f` which is $\forall\alpha,\ \alpha$ `on` $c_1 \times \alpha$ `on` $c_2 \rightarrow \alpha$ `on` c_3, that is: `f` $:: \forall\alpha,\ \alpha$ `on 11(10)` $\times\ \alpha$ `on (1)` $\rightarrow \alpha$ `on (0`^{3}`1)`.

And since we have the types of the buffers, we can compute their sizes. For example, we know that the writing clock of the first buffer is of type α `on` c_1 `on 10(1)` $= \alpha$ `on 11(10) on 10(1)` and that the reading clock is of type α `on` c_2 `on (01)` $= \alpha$ `on (1) on (01)`. By Proposition 4, the size of this buffer is: *size*(`11(10)` *on* `10(1)`, `(1)` *on* `(01)`) $= 1$.

5.3 Guiding the Resolution Algorithm

The resolution algorithm requires the solution of linear inequalities on the indexes of `1`s and on the size of the unknown words. Tools for solving such inequalities are parameterized by an *objective function* determining the criterion to optimize. In the previous example, we choose to optimize the sum of the indexes of `1`s to produce a kind of As-Soon-As-Possible schedule. But we can also use the objective function to favor either system throughput or buffer sizes minimization. We illustrate this trade-off on an example:

```
let node g (i1, i2) = o where
  rec aux1 = i1 when (1001)
  and aux2 = i2 when (0110)
  and o = buffer aux1 + buffer aux2
```



[10] The notation $\mathcal{I}_{c_n}(j) \in S$ designates the presence of the unknown $\mathcal{I}_{c_n}(j)$ in S.

If we assign the type $\forall\alpha.\,(\alpha \times \alpha) \rightarrow \alpha$ `on (01)` to the node `g`, the buffers will be of size 1. But, this node can be executed without buffers if we give it the type: $g :: \forall\alpha.(\alpha$ `on (011110)` $\times\ \alpha$ `on (110011)`$) \rightarrow \alpha$ `on (010010)`.

The first solution can be obtained by an objective function that minimizes the size of the solution. Indeed, since the number of `1`s in the solution is fixed, minimizing the size increases the rate.

The second solution is obtained by an objective function that, for all precedence constraints $\mathcal{I}_{c_x}(j_1) \leq \mathcal{I}_{c_y}(j_2)$, minimizes the value $\mathcal{I}_{c_y}(j_2) - \mathcal{I}_{c_x}(j_1)$. It means that we minimize the number of instants between the writing and the reading of a value in a buffer which has the consequence of reducing the buffer sizes.

5.4 Correctness, Completeness and Complexity

The resolution algorithm shown in Figure 5 relies on the step-by-step transformation of the adaptability constraint system (S2) into linear inequalities (S6), for which there exist algorithms that find a solution if it exists [20]. Each step, except the one between S4 and S5, is a rewriting of a constraint system into an equivalent one (thanks to the equivalence properties stated in Section 4). The step from S4 to S5 is the choice of the number of `1`s in the c_ns. It is correct to seek a solution in a subset of all possible words. Nevertheless, it may lead to incompleteness, since it is possible that a system has no solution in the subset of words considered.

We have parameterized our resolution algorithm by two constants k and k' which modify the number of `1`s in the sought solution. A semi-decidable algorithm to solve adaptability constraints iterates the previous algorithm with $k = 0, 1, 2, \ldots$ and $k' = k + 1$ until it finds a solution. We can prove that this algorithm is complete because if a system of adaptability constraints has a solution S, then there exists a solution S' such that $\forall c'_n \in S'$,

$$\begin{array}{lll} |c'_n.u|_1 &=& |p_n.u| + k \times |p_n.v| \quad (= \ |p'_n.u| + k \times |p'_n.v| \ = \ \ldots) \\ |c'_n.v|_1 &=& (k+1) \times |p_n.v| \qquad (= \ (k+1) \times |p'_n.v| \qquad = \ \ldots) \end{array}$$

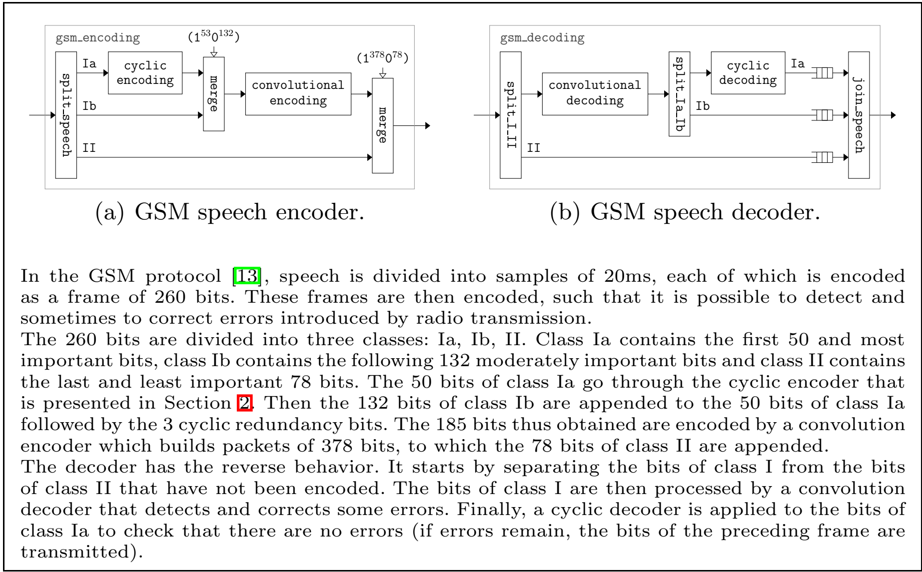
where k can be computed from the original solution S. The idea of the proof is to use Remark 4 to rewrite S into S' (the detailed proof is in the extended version of the paper).

Moreover, note that the step that equalizes of the size of the samplers (from S3 to S4, Figure 5) can be adapted such that the choice $k = 0$ and $k = 1$ always leads to a solution, if it exists, for (1) systems that do not have prefixes, (2) systems where the prefixes of the samplers of a variable are made of `0`s and have the same size, and (3) systems with only one constraint. The algorithm is given in the extended version of the paper.

Remark that we can sometimes find solutions that allow faster execution of a system if we choose a number of `1`s different than the one proposed by $k = 0$ and $k' = 1$. For example, consider the following adaptability constraints:

$$\{\, c_1 \text{ on } \mathtt{(1)} <: c_2 \text{ on } \mathtt{(110)} \,\}$$

If we are seeking a solution with one `1` for c_1, we compute the solution $\{c_1 = \mathtt{(10)};$ $c_2 = \mathtt{(1011)}\}$ where $rate(c_1) = \frac{1}{2}$ and $rate(c_2) = \frac{3}{4}$. Whereas, if we are seeking



(a) GSM speech encoder. (b) GSM speech decoder.

In the GSM protocol [13], speech is divided into samples of 20ms, each of which is encoded as a frame of 260 bits. These frames are then encoded, such that it is possible to detect and sometimes to correct errors introduced by radio transmission.
The 260 bits are divided into three classes: Ia, Ib, II. Class Ia contains the first 50 and most important bits, class Ib contains the following 132 moderately important bits and class II contains the last and least important 78 bits. The 50 bits of class Ia go through the cyclic encoder that is presented in Section 2. Then the 132 bits of class Ib are appended to the 50 bits of class Ia followed by the 3 cyclic redundancy bits. The 185 bits thus obtained are encoded by a convolution encoder which builds packets of 378 bits, to which the 78 bits of class II are appended.
The decoder has the reverse behavior. It starts by separating the bits of class I from the bits of class II that have not been encoded. The bits of class I are then processed by a convolution decoder that detects and corrects some errors. Finally, a cyclic decoder is applied to the bits of class Ia to check that there are no errors (if errors remain, the bits of the preceding frame are transmitted).

Fig. 6. Excerpt of the GSM speech encoder/decoder

a solution with two 1s for c_1, we compute the solution $\{c_1 = \texttt{(110)}; c_2 = \texttt{(1}^6\texttt{)}\}$ where $rate(c_1) = \frac{2}{3}$ and $rate(c_2) = 1$. The guarantee provided by the resolution algorithm is that for a given number of 1s, the result is optimal with respect to the objective function given to the ILP solver. It follows the fact that each transformation of the adaptability constraint system, except Choice 1, maintains equivalence. Therefore, there is no loss of information.

The complexity of the resolution algorithm is dominated by the resolution of the constraint system on the indexes of 1s and the sizes. This is an ILP problem which is known to be NP-complete [20]. Even if there is only one adaptability constraint per buffer, the size of the complete ILP problem can be big (e.g., millions of variables): it depends on the size of the samplers in the adaptability constraint system.

6 Comparison with Previous Resolution Algorithms

Three algorithms for the resolution of adaptability constraints have been proposed. The first one [8] is based on the successive application of local simplification rules. This algorithm does not always succeed because some systems can only be simplified globally, that is, by resolving all of their constraints simultaneously (one such example is given in the long version of this article).

A second algorithm [17], the *abstract resolution algorithm*, is based on the abstraction of clocks by sets of clocks defined by an asymptotic rate and two offsets bounding the potential delay with respect to this rate [9]. Thanks to this

abstraction, the adaptability relation can be tested by some simple operations on rational numbers.

Section 5 of this article presents the third resolution algorithm, the *concrete resolution algorithm*. Technically, the first steps of the algorithm (from S1 to S3, Figure 5) are similar to the ones of the abstract resolution algorithm. The subsequent steps that solve the adaptability constraints (from S3 to S6) differ. In the rest of the section, we will focus on the comparison of the concrete resolution algorithm and the abstract resolution algorithm via some specific examples.

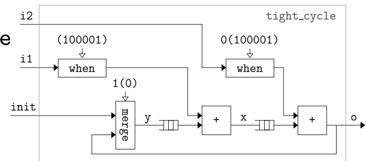
The concrete resolution algorithm allows us to type an excerpt of a GSM speech encoder/decoder. The principle of the encoder/decoder is described in Figure 6 and the source code is available at `http://www.lri.fr/~mandel/mpc12`. This example illustrates the advantages of concrete resolution.

The encoder is depicted in Figure 6(a). The different nodes contain buffers, but they are connected without buffers. The global unification mentioned in Remark 1 is essential to type this node. It can find a rhythm for consuming the input flow such that all the constraints imposed by the processing of the three branches are satisfied.

The abstract resolution algorithm cannot type this node because it cannot treat a unification constraint as a pair of inverse subtyping constraints as proposed in Remark 1. Indeed, when clocks are abstracted, we do not have sufficiently precise information about them to guarantee equality. So, to type the `gsm_encoding` node with the abstract resolution algorithm, we would have to add buffers to communicate the values of the flows `Ia`, `Ib` and `II`, which would transform the unification constraints into subtyping constraints which could be solved. With this new version of the `gsm_encoding` node, the buffer sizes estimated by abstract resolution are 50, 132 and 78, whereas the concrete resolution showed that, for the same throughput, such buffers are not necessary.

The GSM encoder example shows that the concrete resolution algorithm can handle programs that need subtle node scheduling. This advantage is also evident in programs that contain cycles with few initialization values such as the following one:

```
let node tight_cycle (init, i1, i2) = o where
  rec x = i1 when (100001) + buffer y
  and y = merge 1(0) init o
  and o = i2 when 0(100001) + buffer x
```



Here, since there is only one initial value in the cycle, the activations of the two `+` operators are tightly coupled: they must alternate. The abstract resolution algorithm cannot find such a schedule because, in this case, due to the lost information, it cannot guarantee safe communication through the buffers. The concrete resolution algorithm, on the other hand, finds a correct schedule.

Let us now consider the GSM speech decoder depicted in Figure 6(b). Notice that the flows `Ia`, `Ib` are `II` are buffered. The concrete resolution algorithm infers buffer sizes of, respectively, 1, 132 and 156, while the abstract resolution algorithm gives 51, 264 and 234. The buffer sizes estimated by the

abstract resolution algorithm are almost twice as large as those found by the concrete algorithm.

This example shows that when buffers are necessary, the concrete resolution algorithm can give better estimates of buffer sizes.

Notice, however, that the abstract resolution algorithm is still interesting for cases like the video application *Picture in Picture* [17]. Running the concrete resolution algorithm on this example takes several days of computer time! Indeed, because the size of the clock words involved in the system are on the order of two million bits, our algorithm generates a system of linear inequalities containing numbers of variables and constraints of the same order of magnitude. Constraint systems of this size cannot be handled efficiently with tools like GLPK [12] that solve systems of linear inequalities. Finally, the algorithm with abstraction can handle systems where some words are not exactly periodic [9], that is, those with some jitters.

For a given program, one algorithm may be more appropriate than the other. When the periodic words are well balanced, i.e., when the `1`s are regularly spread (as is the case for the nodes of the *Picture in Picture* application), the algorithm with abstraction gives good results quickly. However, it fails when there are some constraints that are difficult to satisfy: e.g. those requiring global unification or those containing cycles. When words are not well balanced, i.e., when the `1`s come in bursts (as in the GSM example) and they are not too long (only hundreds of elements), then the concrete algorithm is better: there is less risk of rejecting a system that has a solution, and the buffer size estimates are better. Finally, unlike the abstract algorithm, the concrete algorithm is not limited to optimizing system throughput. For example, it can find a schedule for *Picture in Picture* that reduces throughput in order to avoid buffering.

7 Conclusion

In this article, we have presented an algorithm that computes schedules and buffer sizes for networks of ultimately periodic processes described as Lucy-n programs.

Scheduling and finding buffer sizes for networks of processes is an old problem. Our particularity is to work in the context of a programming language. In that respect, the most related approaches are those of Ptolemy [11] and StreamIt [22] which are implementations of the Synchronous Data-Flow model [14]. In Ptolemy, the computation nodes are programmed in a host language and the production and consumption rates of nodes are declared by the user. If the values declared by the user are not correct, a program will fail at run-time. The approach of Lucy-n is different: the whole program is written in a single language and the production and consumption rates are inferred automatically from the source code. StreamIt follows the same approach as Lucy-n, but provides only a small number of combinators which restricts the set of networks that can be described.

The main contribution of this paper is to define a resolution algorithm of subtyping constraints that uses all the information contained in the types. Therefore,

it can accept more programs than previous algorithms and it does not overestimate buffer sizes.

Even if the algorithm presented in this paper is computationally more complex than the abstract algorithm presented in [17], our new algorithm can type some programs that would be impossible to type with the other one. In particular, the concrete resolution algorithm has been used [18] to type programs that model latency insensitive design [3]. The types that are obtained for the different nodes of such programs define static schedules for the modeled circuit [7,2,4]. Because of their shape, all these programs make the abstract algorithm fail.

Finally, a great advantage of the concrete resolution algorithm presented in this article is that it does not restrict the trade-off between buffering and throughput. A direction for future work is to provide new language constructs to declare resource constraints and to use them to guide the resolution algorithm.

Acknowledgments. First we would like to thank Timothy Bourke for his numerous and useful comments on the article. It has been a great pleasure to interact with him. We are grateful to Gwenaël Delaval for providing the GSM example which is ideal for motivating and illustrating our approach. Marc Pouzet has been very supportive and always has good advice. Last but not least, we would like to thank the reviewers and the MPC program committee for the quality and the benevolence of their remarks.

References

1. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-static dataflow. IEEE Trans. on Signal Processing 44(2), 397–408 (1996)
2. Boucaron, J., de Simone, R., Millo, J.-V.: Formal methods for scheduling of latency-insensitive designs. EURASIP Journal on Embedded Systems (1) (January 2007)
3. Carloni, L., McMillan, K., Sangiovanni-Vincentelli, A.: Theory of latency-insensitive design. IEEE Trans. on CAD of Integrated Circuits ans Systems 20(9), 1059–1076 (2001)
4. Carmona, J., Júlvez, J., Cortadella, J., Kishinevsky, M.: Scheduling synchronous elastic designs. In: Application of Concurrency to System Design (2009)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: Principles of Programming Languages (1987)
6. Caspi, P., Pouzet, M.: Synchronous Kahn networks. In: International Conference on Functional Programming (May 1996)
7. Casu, M., Macchiarulo, L.: A new approach to latency insensitive design. In: Design Automation Conference (2004)
8. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In: Principles of Programming Languages (2006)
9. Cohen, A., Mandel, L., Plateau, F., Pouzet, M.: Abstraction of Clocks in Synchronous Data-Flow Systems. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 237–254. Springer, Heidelberg (2008)
10. Colaço, J.-L., Pouzet, M.: Clocks as First Class Abstract Types. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 134–155. Springer, Heidelberg (2003)

11. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)
12. GLPK. GNU linear programming kit, `http://www.gnu.org/software/glpk/`
13. Lagrange, X., Godlewski, P., Tabbane, S.: Réseaux GSM: des principes à la norme. Hermès Science, Paris (2000)
14. Lee, E., Messerschmitt, D.: Synchronous data flow. IEEE Transactions on Computers 75(9) (September 1987)
15. Mandel, L., Plateau, F.: Typage des horloges périodiques en Lucy-n. In: Journées Francophones des Langages Applicatifs, La Bresse, France (January 2011)
16. Mandel, L., Plateau, F.: Scheduling and buffer sizing of n-synchronous systems — extended version (2012), `http://www.lri.fr/~mandel/mpc12`
17. Mandel, L., Plateau, F., Pouzet, M.: Lucy-n: a n-synchronous extension of Lustre. Mathematics of Program Construction (2010)
18. Mandel, L., Plateau, F., Pouzet, M.: Static scheduling of latency insensitive designs with Lucy-n. In: Formal Methods in Computer Aided Design (2011)
19. Wesley Peterson, W.: Error-Correcting Codes. The M.I.T. Press (1961)
20. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons (1986)
21. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: International Conference on Functional Programming, pp. 341–352 (2009)
22. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)

A Clock Constraints Collection

Clock types are separated into three categories: type schemes (σ) which represent the types of nodes, types of expressions (t) and types of streams (ct).

$$\begin{array}{lcl} \sigma & ::= & \forall \beta_1, \ldots, \beta_m. \forall \alpha_1, \ldots, \alpha_n.\, ct \to ct \\ t & ::= & \beta \mid t \times t \mid ct \\ ct & ::= & \alpha \mid ct \texttt{ on } ce \mid ct \texttt{ on not } ce \end{array}$$

The typing environment H is a triple which contains the types of the flow variables, the types of the nodes and the type of the clocks:

$$\begin{array}{rl} H ::= & ([\, x_1 : ct_1, \ldots, x_p : ct_p \,], \\ & \;[\, f_1 : \sigma_1, \ldots, f_m : \sigma_m \,], \\ & \;[\, c_1 : ce_1, \ldots, c_n : ce_n \,]) \end{array}$$

We use the notation $H + [z : t]$ to add the association $z : t$ to the appropriate part of the triple. We define the notation $[pat : t]$ as follows:

$$[pat : t] = \begin{cases} [x : t] & \text{if } pat = x \\ [pat_1 : t_1] + \cdots + [pat_n : t_n] & \text{if } pat = \texttt{(}pat_1\texttt{,}\ldots\texttt{,}pat_n\texttt{)} \text{ and } t = t_1 \times \cdots \times t_n \end{cases}$$

$$H \vdash i : \alpha \mid \emptyset \qquad \frac{H \vdash ce}{H \vdash ce : \alpha \mid \emptyset} \qquad H \vdash x : H(x) \mid \emptyset$$

$$\frac{H \vdash e_1 : t_1 \mid C_1 \qquad \ldots \qquad H \vdash e_n : t_n \mid C_n}{H \vdash (e_1, \ldots, e_n) : t_1 \times \cdots \times t_n \mid C_1 \cup \cdots \cup C_n}$$

$$\frac{H \vdash e_1 : t_1 \mid C_1 \qquad H \vdash e_2 : t_2 \mid C_2}{H \vdash e_1\ op\ e_2 : ct \mid \{t_1 = t_2 = ct\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \mid C \qquad H \vdash e_1 : t_1 \mid C_1 \qquad H \vdash e_2 : t_2 \mid C_2}{H \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : ct \mid \{t = t_1 = t_2 = ct\} \cup C \cup C_1 \cup C_2}$$

$$\frac{t_1 \to t_2 \in inst(H(f)) \qquad H \vdash e : t_3 \mid C}{H \vdash f e : t_2 \mid \{t_1 = t_3\} \cup C}$$

$$\frac{H \vdash eqs : H' \mid C_1 \qquad H + H' \vdash e : ct \mid C_2}{H \vdash e\ \texttt{where rec}\ eqs : ct \mid C_1 \cup C_2}$$

$$\frac{H \vdash e_1 : t_1 \mid C_1 \qquad H \vdash e_2 : t_2 \mid C_2}{H \vdash e_1\ \texttt{fby}\ e_2 : ct \mid \{t_1 = t_2 = ct\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \mid C \qquad H \vdash ce : ct \mid \emptyset}{H \vdash e\ \texttt{when}\ ce : ct\ \texttt{on}\ ce \mid \{t = ct\} \cup C} \qquad \frac{H \vdash e : t \mid C \qquad H \vdash ce : ct \mid \emptyset}{H \vdash e\ \texttt{whenot}\ ce : ct\ \texttt{on not}\ ce \mid \{t = ct\} \cup C}$$

$$\frac{H \vdash ce : ct \mid \emptyset \qquad H \vdash e_1 : t_1 \mid C_1 \qquad H \vdash e_2 : t_2 \mid C_2}{H \vdash \texttt{merge}\ ce\ e_1\ e_2 : ct \mid \{ct\ \texttt{on}\ ce = t_1, ct\ \texttt{on not}\ ce = t_2,\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \mid C}{H \vdash \texttt{buffer}(e) : \alpha \mid C \cup \{t <: \alpha\}}$$

$$\frac{H + [pat : \beta] \vdash e : t \mid C}{H \vdash pat = e : [pat : t] \mid \{\beta = t\} \cup C} \qquad \frac{H + H_2 \vdash eqs_1 : H_1 \mid C_1 \qquad H + H_1 \vdash eqs_2 : H_2 \mid C_2}{H \vdash eqs_1\ \texttt{and}\ eqs_2 : H_1 + H_2 \mid C_1 \cup C_2}$$

$$\frac{H + [x : \beta] \vdash e : t \mid C}{H \vdash \texttt{let node}\ f(x) = e : [f : gen(\beta \to t, C)]} \qquad \frac{H \vdash ce}{H \vdash \texttt{let clock}\ c = ce : [c : ce]}$$

$$\frac{H \vdash d_1 : H_1 \qquad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2}$$

Fig. 7. Clock type constraints collection

Types can be instantiated and generalized using the following rules:

$$\begin{aligned}
&\mathit{inst}(\forall \beta_1, \dots, \beta_m. \forall \alpha_1, \dots, \alpha_n. t) = \\
&\quad \{ t' \mid t' = t[\beta_1 \leftarrow t_1, \dots, \beta_m \leftarrow t_m, \alpha_1 \leftarrow ct_1, \dots, \alpha_n \leftarrow ct_n] \} \\
&\mathit{gen}(t, C) = \forall \beta_1, \dots, \beta_m. \forall \alpha_1, \dots, \alpha_n. t' \\
&\quad \text{where } t' = \theta(t) \text{ such that } \theta(C) \text{ is satisfied} \\
&\quad \text{and } \{\beta_1, \dots, \beta_m, \alpha_1, \dots, \alpha_n\} = FV(t')
\end{aligned}$$

The typing rules which collect the clocking constraints are given in Figure 7. The typing rules have the shape $H \vdash e : t \,|\, C$ which means that in the typing environment H, the expression e has type t and must satisfy the set of constraints C.

Finally, notice that $ct \times ct \equiv ct$. Therefore, constraints such as $t_1 \times t_2 = ct$ can be split into $t_1 = ct$ and $t_2 = ct$.

Deriving Real-Time Action Systems Controllers from Multiscale System Specifications

Brijesh Dongol[1,2] and Ian J. Hayes[1]

[1] School of Information Technology and Electrical Engineering
The University of Queensland, Australia
[2] Department of Computer Science, The University of Sheffield, UK
B.Dongol@sheffield.ac.uk, Ian.Hayes@itee.uq.edu.au

Abstract. This paper develops a method for deriving controllers for real-time systems in which the components of the system operate at different time granularities. To this end, we incorporate the theory of time bands into action systems, which allows one to structure a system into multiple abstractions of time. The framework includes a logic that facilitates reasoning about different types of sampling errors and transient properties (i.e., properties that only hold for a brief amount of time), and we develop theorems for simplifying proofs of hardware/software interaction. We formalise true concurrency and define refinement for the parallel composition of action systems. Our method of derivation builds on the verify-while-develop paradigm, where the action system code is developed side-by-side with its proof.

1 Introduction

Action systems provide a simple framework within which several theories of program refinement have been developed [3–6]. In its simplest form, an action system consists of a set of actions (i.e., guarded statements) and a loop that at each iteration non-deterministically chooses then executes an enabled action from the set of actions. The loop terminates iff all of the actions are disabled. Typically, an action system includes actions of both the controller and its environment and uses an execution model in which the controller and environment actions are interleaved with each other. To cope with continuous environments, action systems have been extended in several ways. *Continuous action systems* [2, 27] give a semantics using standard action systems but with an added time variable. At the end of each iteration of the main loop, time is incremented to the first time at which some guard of the action system is enabled. *Hybrid action systems* [30] take the approach that the actions of the controller are discrete (and instantaneous) and allow the environment to execute evolution actions, which describe the (continuous) evolution of the state over the interval in which the evolution guard is enabled. A prioritised alternating model of execution is used to ensure that the (discrete) controller actions are able to execute. Hybrid action systems have been extended to *qualitative action systems* [1], but this work

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 102–131, 2012.
© Springer-Verlag Berlin Heidelberg 2012

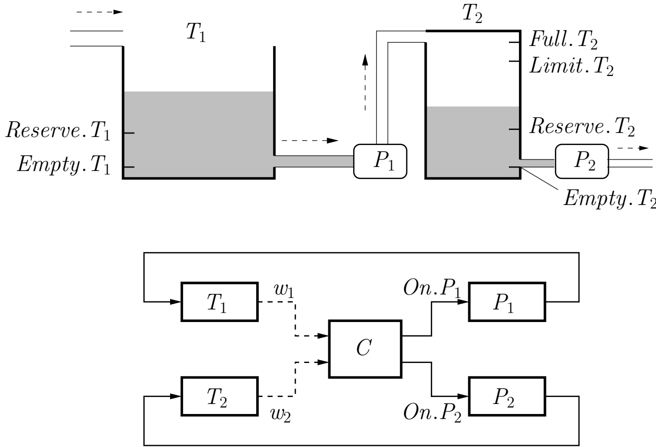


Fig. 1. Two-pump system

is focused on methods for testing real-time systems as opposed to their formal verification/derivation.

Ultimately, the interleaving execution model is problematic in contexts such as real-time and multi-core systems where the environment evolves with the controller in a truly concurrent manner. In such contexts, one must address issues with sampling multiple variables over a time interval [9, 17, 23] and be able to reason about transient properties [14, 17]. Furthermore, as software controllers are increasingly used in complex cyber-physical systems, it becomes important to be able to reason over multiple time granularities [8, 9, 16, 23].

1.1 Motivating Example

We consider a system consisting of two water tanks T_1, T_2 and two pumps P_1, P_2 depicted in Fig. 1 (also see [1]). The environment (of the system) adds water to tank T_1 and does not affect tank T_2. We assume that tank T_1 is allowed to overflow, but T_2 is not. Pump P_1 removes water from tank T_1 and fills tank T_2. Pump P_2 only operates if a button B (not shown in Fig. 1) is pressed and removes water from tank T_2. Aichernig et al [1] describe the following requirements. We have adapted their informal specification to clarify the input/output behaviours of the pump and to better distinguish safety (**S1**, **S2** and **S3**) and progress (**P1**, **P2** and **P3**). Note that a progress property to turn pump P_1 off is not needed because it is implied by safety properties **S1** and **S2**.

S1. If the water level in T_1 is *Empty* or below, then P_1 must be stopped.
S2. If the water level in T_2 is *Full* or above, then P_1 must be stopped.
S3. If the water level in T_2 is *Empty* or below, then P_2 must be stopped.

P1. If the water level in T_2 is definitely below the *Reserve* and water level in T_1 is definitely above the *Reserve*, then turn on P_1.

P2. If B is definitely pressed and the water level in T_2 is definitely above *Reserve*, then turn on P_2.

P3. If B is definitely not pressed, then turn off P_2.

Thus, we must keep track of water levels $Reserve.T_1$ and $Empty.T_1$ in tank T_1 and $Full.T_2$, $Reserve.T_2$, $Empty.T_2$ in tank T_2. For $i \in \{1, 2\}$, we distinguish between signal On_i that starts/stops pump P_i, and $Running_i$ and $Stopped_i$ that hold iff P_i is physically running and stopped, respectively. Note that pump P_i may also be associated with other states such as $Starting_i$, etc. We let w_1 and w_2 denote the water levels in tanks T_1 and T_2, respectively and say *Pressed* holds iff the button B is pressed.

A (digital) controller for pump P_2 must sample both the water level in tank T_2 and the state of the button B, perform some processing, then send on/off signals to pump P_2 if necessary. Each of these phases takes time. Furthermore, the components operate at different time granularities and hence have different notions of precision (the amount of time that may be regarded as instantaneous [8, 9]). For example, w_1 may have a precision of 30 seconds (i.e., there is no significant change in the water level in tank T_1 within 30 seconds) and pump P_2 turns on/off with precision 1 second (i.e., it takes pump P_2 at most 1 second to reach its operating speed or to come to a stop). Formally reasoning about the system in a manner that properly addresses each of these timing aspects is complicated [16, 17, 22]. To reduce the complexity of the reasoning, formal frameworks often simplify specifications by assuming that certain aspects of the system (e.g., sampling) are instantaneous or take a negligible amount of time. However, it is well-known that such simplifications can cause complications during implementation. In particular, the developed specifications become unimplementable because their timing requirements cannot be satisfied by any real system [22, 31].

Properties that use "definitely" are properties that hold over events of a time band. The progress properties are interpreted in the water time band. For example, within **P1**, "T_2 is definitely below the *Reserve*" is interpreted as "T_2 is definitely below the *Reserve* for at least the precision of the water time band".

1.2 Contributions and Overview

In this paper, we use time bands [8, 9] which facilitate reasoning about systems specified over multiple time granularities. Together with a logic of sampling, this allows one to properly address transient properties [14, 17]. We develop a framework using action systems and formalise true concurrency between an action system and its environment as well as between the parallel composition of two or more action systems. We define stream-based refinement of action systems (and their parallel composition) and present our method of derivation using "enforced properties" [12, 14], which allows one to use a verify-while-develop method [11, 18, 19]. We develop high-level theories for reasoning about systems that involve interaction between hardware and software over multiple time bands and as an example, we present the derivation of a pump controller.

Unlike Burns/Hayes [9] whose framework is based on sets of states, we develop our semantics using an interval-based framework. We present the background theory in Section 2, where we define interval predicates, methods of evaluating state predicates over an interval (including via sampling), our chop and iterated chop operators [32], and an LTL-like [26] logic for interval predicates. In Section 3, we present our formalisation of time bands (which includes time-band predicates), formalisation of the syntax and semantics of action systems with time bands and parallel composition of action systems. In Section 4, we present our methods for deriving action systems using enforced properties that builds on our previous refinement theories [12, 14, 17]. Section 5 presents our high-level theorems for reasoning about hardware/software interaction and an example derivation is given in Section 6.

1.3 Related Work

The idea of reasoning about systems using multiple granularities of time is not new. Moszkowski presents a method of abstracting between different time granularities for interval temporal logic using a projection operator for a discrete interval temporal logic [28]. Guelev and Hung present a projection operator for the duration calculus. Although computation is assumed to take time, the time taken is assumed to be negligible [21]. Henzinger presents a theory of timed refinement where sampling events are executed by a separate process [25]. Broy [7] presents a timed refinement framework that formalises the relationships between dense and discrete time where sampling is considered be a discretisation of dense streams.

This paper continues our research into methods for program derivation using the verify-while-develop paradigm. The method of enforced properties [12, 13] has been extended to enable development of action systems in a compositional manner [14]. The logic in [14] considers traces that consist of pre/post state relations, develops a temporal logic on relations and assumes that environment transitions are interleaved with those of an action system. Although the framework facilitates compositional derivation of action systems code, the underlying interleaving semantics assumption could not properly address sampling anomalies and transient properties. Hence, the framework was generalised so that traces consisted of adjoining intervals together with a sampling logic (Section 2.2), which allowed sampling-related issues to be properly addressed [17]. However, the logic in [17] does not adequately handle specifications over multiple time granularities. Instead, hardware is assumed to react and take effect instantaneously, which is unrealistic.

2 Background Theory

2.1 Interval Predicates

We model time using the real numbers, $\mathbb{R}$, and let *Interval* denote the set of all contiguous non-empty subsets of time. An interval may be open or closed at

either end, have a least upper bound ∞ or a greatest lower bound $-\infty$ (i.e., not in $\mathbb{R}$).

We let $glb.\Delta$ and $lub.\Delta$ denote the *greatest lower* and *least upper bounds* of interval Δ, respectively, where '.' denotes function application. For intervals Δ and Δ', we define the *length* of Δ and *adjoins* relation between Δ and Δ' as follows.

$$\ell.\Delta \mathrel{\widehat{=}} lub.\Delta - glb.\Delta$$
$$\Delta \propto \Delta' \mathrel{\widehat{=}} (lub.\Delta = glb.\Delta') \wedge (\Delta \cup \Delta' \in Interval) \wedge (\Delta \cap \Delta' = \{\})$$

Given that variable names are taken from the set Var, a *state space* over a set of variables $V \subseteq Var$ is given by $\Sigma_V \mathrel{\widehat{=}} V \to Val$, which is a total function from variables in V to values in Val. A *state* is a member of Σ_V. The (dense) stream of states over V is given by $Stream_V \mathrel{\widehat{=}} \mathbb{R} \to \Sigma_V$, which is a total function from real numbers to states. A *predicate* over a type T is given by $\mathcal{P}T \mathrel{\widehat{=}} T \to \mathbb{B}$ (e.g., a *stream predicate* is a member of $\mathcal{P}Stream_V$), where $\mathbb{B}$ is the type of a boolean. An interval stream predicate, which we shorten to *interval predicate*, has type $IntvPred_V \mathrel{\widehat{=}} Interval \to \mathcal{P}Stream_V$. We write Σ, $Stream$ and $IntvPred$ for Σ_V, $Stream_V$ and $IntvPred_V$, respectively when the set V is clear from the context.

For an interval predicate p and interval Δ we define the following, where the stream is implicit in both sides of the definitions.

$$(prev.p).\Delta \mathrel{\widehat{=}} \exists \Delta' \colon Interval \bullet (\Delta' \propto \Delta) \wedge p.\Delta'$$
$$(next.p).\Delta \mathrel{\widehat{=}} \exists \Delta' \colon Interval \bullet (\Delta \propto \Delta') \wedge p.\Delta'$$
$$(\boxplus p).\Delta \mathrel{\widehat{=}} \forall \Delta' \colon Interval \bullet \Delta' \subseteq \Delta \Rightarrow p.\Delta'$$

Thus $(prev.p).\Delta$ and $(next.p).\Delta$ hold iff p holds in some interval that immediately precedes and follows Δ, respectively and $(\boxplus p).\Delta$ holds iff p holds in each subinterval of Δ.

We assume pointwise lifting of the boolean operators on stream and interval predicates in the normal manner, e.g., if p_1 and p_2 are interval predicates, Δ is an interval and s is a stream, we have $(p_1 \wedge p_2).\Delta.s = (p_1.\Delta.s \wedge p_2.\Delta.s)$. When reasoning about programs and their properties, we must often state that if an interval predicate p_1 holds over an arbitrarily chosen interval Δ and stream s, then an interval predicate p_2 also holds over Δ and s. Hence, we define universal implication over intervals and streams as follows. Operators '$\equiv$' and '$\Lleftarrow$' are similarly defined.

$$p_1.\Delta \Rrightarrow p_2.\Delta \quad \mathrel{\widehat{=}} \quad \forall s \colon Stream \bullet p_1.\Delta.s \Rightarrow p_2.\Delta.s$$
$$p_1 \Rrightarrow p_2 \quad \mathrel{\widehat{=}} \quad \forall \Delta \colon Interval \bullet p_1.\Delta \Rrightarrow p_2.\Delta$$

2.2 Evaluating State Predicates over an Interval

Because there are multiple states of a stream within a non-point interval, there are several possible ways of evaluating a state predicate with respect to a given interval and stream [23].

We must often determine the value of a variable at the left and right ends of an interval. Because intervals may be open/infinite at either end, these values are determined using limits. We use $\lim_{x \to a^+} f.x$ and $\lim_{x \to a^-} f.x$ to denote the limit of $f.x$ as x tends to a from above and below, respectively. To ensure that the limits are well defined, we assume all variables are piecewise continuous [20]. For a vector of variables $\mathbf{v}$, interval Δ, stream s, time t, we let $(\mathbf{v}@t).s \mathrel{\widehat{=}} (s.t).\mathbf{v}$ denote the value of $\mathbf{v}$ in state $s.t$ and define:

$$\overrightarrow{\mathbf{v}}.\Delta \mathrel{\widehat{=}} \begin{cases} \mathbf{v}@(lub.\Delta) & \text{if } lub.\Delta \in \Delta \\ \lim_{t \to lub.\Delta^-} \mathbf{v}@t & \text{otherwise} \end{cases} \qquad \overleftarrow{\mathbf{v}}.\Delta \mathrel{\widehat{=}} \begin{cases} \mathbf{v}@(glb.\Delta) & \text{if } glb.\Delta \in \Delta \\ \lim_{t \to glb.\Delta^+} \mathbf{v}@t & \text{otherwise} \end{cases}$$

Thus, if Δ is right closed, then the value of $\overrightarrow{\mathbf{v}}$ in Δ is the value of $\mathbf{v}$ at the greatest upper bound of Δ, otherwise (i.e., Δ is right-open), the value of $\overrightarrow{\mathbf{v}}$ is the value of the $\mathbf{v}$ as it approaches $lub.\Delta$ from the right. The interpretation of $\overleftarrow{\mathbf{v}}.\Delta$ is similar. Given that $vec.c$ denotes the vector of all free variables of state predicate c, we define

$$\overleftarrow{c}.\Delta.s \mathrel{\widehat{=}} c[vec.c \setminus (\overleftarrow{vec.c}).\Delta.s] \qquad \overrightarrow{c}.\Delta.s \mathrel{\widehat{=}} c[vec.c \setminus (\overrightarrow{vec.c}).\Delta.s]$$

We must often specify properties on the actual states of a stream within an interval. Thus, we define the *always* and *sometime* operators as follows[1], where $(c@t).s \mathrel{\widehat{=}} c.(s.t)$ for any state predicate c, time t and stream s.

$$(\boxast c).\Delta \equiv \forall t{:}\, \Delta \bullet (c@t) \qquad (\boxdot c).\Delta \equiv \exists t{:}\, \Delta \bullet (c@t)$$

For a state predicate c, variable v and set of variables V:

$$st.v \mathrel{\widehat{=}} \forall k{:}\, Val \bullet prev.(\overrightarrow{v} = k) \Rightarrow \boxast(v = k) \qquad st.V \mathrel{\widehat{=}} \forall v{:}\, V \bullet st.v$$

Hence, a variable v is stable, denoted $st.v$, iff its value does not change from its value at the right end of some previous interval, and $st.V$ holds iff each variable in V is stable. Such definitions of stability are necessary because adjoining intervals are disjoint, and hence $prev.(\overrightarrow{v} = k)$ does not necessarily imply $\overleftarrow{v} = k$ and vice versa.

Example 1. Consider a variable x such that $(x@0) = 10$ and $(\boxast \widehat{x}).[0, 2] = 1$ hold, where $\widehat{x}$ denotes the rate of change of variable x (c.f. [24]). Thus, the value of x at time 0 is 10 and the rate of change of x throughout the closed interval $[0, 2]$ is 1. Then for adjoining intervals $[0, 1)$ and $[1, 2]$, both $(\overrightarrow{x} = 11).[0, 1)$ and $(\overleftarrow{x} = 11).[1, 2]$ hold. In fact, we can deduce both $(\boxast(x < 11)).[0, 1)$ and $(\boxast(x \geq 11)).[1, 2]$. However, for adjoining intervals $[0, 1]$ and $(1, 2]$, $(\boxast(x \leq 11)).[0, 1]$ and $(\boxast(x > 11)).(1, 2]$ hold.

[1] Our notation follows Burns and Hayes [9] and should not be confused with modal operator 'always' ($\Box$) (and 'next' ($\bigcirc$) later). Instead, we ask the reader to focus on the '$*$' within $\boxast$ (and $\circledast$ later), which represents "for all" and '$\cdot$' within $\boxdot$ (and $\odot$ later) which represents "for some" as used when writing regular expressions.

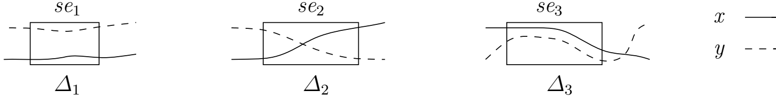


Fig. 2. Sampling events se_1, se_2 and se_3

Real-time controllers often evaluate an expression over an interval by sampling the variables of the expression (once per variable) at different instants within the interval. Hence, reasoning about an expression evaluation that samples two or more variables can be problematic. For example, consider the three sampling events se_1, se_2 and se_3 in Fig. 2, where environment variables x and y are sampled at different times within the interval. Event se_1 will return $x < y$ regardless of when x and y are read within the sampling interval because $x < y$ *definitely* holds for all sampled values of x and y. Event se_2 may return either $x > y$, $x = y$ or $x < y$ because it is *possibly* true that $x > y$, $x = y$ and $x < y$ hold. Event se_3 may have a sampling anomaly. Although $x > y$ holds throughout se_3, because x and y are sampled at different times, it is *possible* for se_3 to return either $x > y$, $x = y$ or $x < y$. If a variable occurs multiple times within an expression, the same sampled value is used for each occurrence of the variable. Hence, expression $x = x$ is guaranteed to evaluate to true regardless of how x changes within the evaluation interval, however, $x > y$ may evaluate to false even if $\boxast(x > y)$ holds [9, 16, 23] as in se_3 in Figure 2.

We use the set of apparent states of $s \in \mathit{Stream}_V$ within interval Δ (denoted $\mathit{apparent}.\Delta.s$) to reason about sampling-based expression evaluation. We define:

$$\mathit{apparent}.\Delta.s \mathrel{\widehat{=}} \{\sigma \colon \Sigma_V \mid (\forall\, v \colon V \bullet \sigma.v \in \{t \colon \Delta \bullet (s.t).v\})\}$$

where $\{t \colon \Delta \bullet (s.t).v\}$ is equivalent to $\{x \in \mathit{Val} \mid \exists t \colon \Delta \bullet x = (s.t).v\}$. To generate the apparent states, we first generate $\{t \colon \Delta \bullet (s.t).v\}$, the set of possible values of the variables within the interval, then generate the set of all possible states using these values. We formalise state predicates that are *definitely* true (denoted $\circledast$) and *possibly* true (denoted $\odot$) over a given interval Δ and stream s as follows:

$$(\circledast c).\Delta.s \mathrel{\widehat{=}} \forall \sigma \colon \mathit{apparent}.\Delta.s \bullet c.\sigma \qquad (\odot c).\Delta.s \mathrel{\widehat{=}} \exists \sigma \colon \mathit{apparent}.\Delta.s \bullet c.\sigma$$

Hence, $(\circledast c).\Delta.s$ and $(\odot c).\Delta.s$ hold iff c holds in every and in some apparent state of s within the interval Δ, respectively. For example, for Δ_1, Δ_2 and Δ_3 in Fig. 2 we can deduce both $(\circledast(x < y)).\Delta_1$ and $(\odot(x < y) \land \odot(x \geq y)).\Delta_2$. For se_3 (the event with a sampling anomaly), $(\odot(x \leq y)).\Delta_3$ holds, despite the fact that $\boxast(x > y).\Delta_3$ holds. Both $\circledast c \Rrightarrow \boxast c$ and $\boxdot c \Rrightarrow \odot c$ hold, but the converse of both implications is not necessarily true.

Lemma 2. *For any variable v and constant k, $st.v \land \odot(v = k) \Rrightarrow \boxast(v = k)$.*

We let $vars.c$ denote the free variables of state predicate c. The following lemma states that if all but one variable of c is stable over an interval $\varDelta$, then c definitely holds in $\varDelta$ iff c always holds in $\varDelta$ and c possibly holds in $\varDelta$ iff c holds sometime in $\varDelta$ [23].

Lemma 3. *For any state predicate c and variable v,*
$st.(vars.c \setminus \{v\}) \Rrightarrow (\circledast c = \boxast c) \wedge (\odot c = \boxdot c)$.

2.3 Chop and Iteration

The *chop* operator ';' is a useful basic operator in interval-based logics [29, 32]. For interval predicates p_1 and p_2 and interval $\varDelta$, we define:

$$(p_1 \ ; \ p_2).\varDelta \mathrel{\widehat=} \exists \varDelta_1, \varDelta_2 \colon Interval \bullet (\varDelta = \varDelta_1 \cup \varDelta_2) \wedge (\varDelta_1 \propto \varDelta_2) \wedge p_1.\varDelta_1 \wedge p_2.\varDelta_2$$

Thus $(p_1 \ ; \ p_2).\varDelta$ holds iff $\varDelta$ can be split into two adjoining intervals so that p_1 holds for the first interval and p_2 holds for the second. Unlike Moszkowski [29], we have a dense notion of time and unlike the duration calculus [32], our chop operator does not require $\varDelta_1$ and $\varDelta_2$ to be closed intervals. Thus, for x as given in Example 1, both $(\boxast(x < 11); \boxast(x \geq 11)).[0, 2]$ and $(\boxast(x \leq 11); \boxast(x > 11)).[0, 2]$ hold, but $(\boxast(x < 11) \ ; \ \boxast(x > 11)).[0, 2]$ does not.

Using chop, we define the *weak chop* and *iterated chop* operators as follows:

$$p_1 : p_2 \mathrel{\widehat=} p_1 \vee (p_1 \ ; \ p_2) \qquad\qquad p^\omega \mathrel{\widehat=} \mu\, q \bullet p : q$$

That is, $p_1 : p_2$ holds iff either p_1 holds or the given interval may be chopped so that $p_1 \ ; \ p_2$ holds. The iterated chop p^ω is the least fixed point of the weak chop (which defines both finite and infinite iteration of p) assuming that predicates are ordered using universal reverse entailment ($\Lleftarrow$).

Because we have a dense notion of time, there is a possibility for an iteration p^ω to behave in a Zeno-like manner, where p iterates an infinite number of times within a finite interval. We can rule out Zeno-like behaviour in our implementations because there is a physical lower limit on the time taken to perform each iteration and hence a specification that *allows* Zeno-like behaviour can be safely ignored. However, we must be careful not to *require* Zeno-like behaviour, which would cause our specifications to become unimplementable.

Lemma 4 (ω-unfolding). $p^\omega \equiv p \vee (p \ ; \ p^\omega)$

Definition 5. *We say an interval predicate p* splits *iff $p \Rrightarrow \boxplus p$ holds and* joins *iff $p^\omega \Rrightarrow p$ holds.*

For example, $\boxast c$ both splits and joins, $\ell \leq 42$ splits but does not join, $\ell \geq 42$ joins but does not split, and $\ell = 42$ neither splits nor joins. In particular, if $(\ell \leq 42).\varDelta$ holds, then $\ell \leq 42$ holds for all subintervals of $\varDelta$. On the other hand, if $(\ell \leq 42).\varDelta_1$ and $(\ell \leq 42).\varDelta_2$ where $\varDelta_1 \propto \varDelta_2$, we cannot guarantee that $(\ell \leq 42).(\varDelta_1 \cup \varDelta_2)$ holds. Interval predicates that join allow proofs to be decomposed more easily [16].

2.4 ILTL

To cope with durative behaviour, the traces of an action system are defined using a sequence of adjoining intervals. Thus, we use an interval-based linear temporal logic (ILTL) (as opposed to state-based [26]). If p is an interval predicate, the syntax of basic ILTL formulae is given by

$$F ::= p \mid \Box F \mid \bigcirc F \mid F_1 \,\mathcal{U}\, F_2 \mid \amalg p \mid \neg F$$

Given that seq.T denotes the possibly infinite sequences of type T we define

$$AdjSeq \mathrel{\widehat{=}} \{z\colon \text{seq}.Interval \mid \forall i\colon \text{dom}.z \setminus \{0\} \bullet z.(i-1) \propto z.i\}\,.$$

For the rest of this paper, we let z be a variable of type $AdjSeq$. The semantics of ILTL formulae is given below, where notation $(z, s)_i \vdash F$ states that the ILTL formula F holds for the pair (z, s) starting from index $i \in \text{dom}.z$.

Definition 6. *If p is an interval predicate, F is an ILTL formula, $z \in AdjSeq$, s is a stream and $i \in \text{dom}.z$, and $tr = (z, s)$ then:*

$$\begin{aligned}
tr \vdash F &\mathrel{\widehat{=}} tr_0 \vdash F\\
tr_i \vdash p &\mathrel{\widehat{=}} p.(z.i).s\\
tr_i \vdash \Box F &\mathrel{\widehat{=}} \forall j\colon \text{dom}.z \bullet j \geq i \Rightarrow (tr_j \vdash F)\\
tr_i \vdash \bigcirc F &\mathrel{\widehat{=}} i+1 \in \text{dom}.z \Rightarrow (tr_{i+1} \vdash F)\\
tr_i \vdash F_1 \,\mathcal{U}\, F_2 &\mathrel{\widehat{=}} \exists j\colon \text{dom}.z \bullet j \geq i \wedge (tr_j \vdash F_2) \wedge \forall k \bullet i \leq k < j \Rightarrow (tr_k \vdash F_1)\\
tr_i \vdash \amalg p &\mathrel{\widehat{=}} p.(\textstyle\bigcup_{j \in \text{dom}.z \wedge i \leq j}(z.j)).s\\
tr_i \vdash \neg F &\mathrel{\widehat{=}} \neg(tr_i \vdash F)
\end{aligned}$$

Thus, $(z, s)_i \vdash p$ holds iff p holds in s within the interval $z.i$. Operators $\Box$ and $\bigcirc$ and $\mathcal{U}$ express *always*, *next* and *until*, respectively, and $(z, s)_i \vdash \amalg p$ states that interval predicate p holds for the interval consisting of the union of all intervals in z from $z.i$ onwards. We define universal implication for temporal formulae F_1 and F_2 as follows. Both $F_1 \equiv F_2$ and $F_1 \Lleftarrow F_2$ are similarly defined.

$$F_1 \Rrightarrow F_2 \mathrel{\widehat{=}} \forall z\colon AdjSeq, s \in Stream \bullet ((z, s) \vdash F_1) \Rightarrow ((z, s) \vdash F_2)$$

Temporal operators *eventually*, *unless* and *leads-to* are defined as follows:

$$\Diamond F \mathrel{\widehat{=}} \neg\Box\neg F \qquad F_1 \,\mathcal{W}\, F_2 \mathrel{\widehat{=}} \Box F_1 \vee (F_1 \,\mathcal{U}\, F_2) \qquad F_1 \leadsto F_2 \mathrel{\widehat{=}} \Box(F_1 \Rightarrow \Diamond F_2)$$

Systems often require that a state predicate be maintained unless another property is established.Thus, for state predicates c_1 and c_2, we define a *maintained unless* operator $\mathcal{M}$ as follows:

$$c_1 \,\mathcal{M}\, c_2 \mathrel{\widehat{=}} \Box(prev.\overrightarrow{c_1} \Rightarrow (\circledast c_1 \,\mathcal{W}\, \odot c_2))$$

That is, if $(z, s) \vdash c_1 \,\mathcal{M}\, c_2$ holds, then for any $i \in \text{dom}.z$, if $(prev.\overrightarrow{c_1}).(z.i).s$, then either c_1 definitely holds for all $j \geq i$, or c_2 possibly holds in $z.k$ and c_1 definitely holds for all j such that $k > j \geq i$.

Like LTL [26], it is difficult to prove general ILTL formulae directly. However, certain forms of LTL formulae may be transformed into formulae of the form $\Box p$ (for an interval predicate p), which is simpler to prove [17].

Lemma 7. *For any interval predicate p and state predicates c_1 and c_2,*

(a) *if p joins, then* $\Box p \Rrightarrow \amalg p$,
(b) $\amalg \boxplus p \Rrightarrow \Box p$, *and*
(c) $(c_1 \mathcal{M}\, c_2) \equiv \Box(prev.\overrightarrow{c_1} \Rightarrow \circledast c_1 \vee \odot c_2)$.

3 Action Systems with Time Bands

3.1 Time Bands

Like Burns, we assume that the set of all time bands is given by the primitive type *TimeBand* [8, 9]. Each time band may be associated with *events* that execute within the *precision* of the time band. We use ρ: $TimeBand \to \mathbb{R}^{>0}$ to denote the precision of the given time band.

To simplify the specification of the behaviour of an event in a time band, we define the type of a *time band predicate* as $TBPred_V$: $TimeBand \to IntvPred_V$, which for a given time band returns an interval predicate. As with interval predicates, we assume time band predicates are lifted pointwise over boolean operators and for time band predicates tp_1 and tp_2, we define $tp_1 \Rrightarrow tp_2 \mathrel{\widehat{=}} \forall \beta\colon TimeBand \bullet tp_1.\beta \Rrightarrow tp_2.\beta$ (and similarly $\Lleftarrow$ and $\equiv$).

We define the following interval predicates, which are useful for reasoning about sampling events, where c is a state predicate and n is a real-valued constant.

$$\circledast_n c \mathrel{\widehat{=}} (\ell \leq n) \Rightarrow \circledast c \qquad \odot_n c \mathrel{\widehat{=}} (\ell \leq n) \wedge \odot c$$

Hence, $(\circledast_n c).\Delta$ holds iff c definitely holds within Δ provided that the length of Δ is at most n. Similarly, $(\odot_n c).\Delta$ holds iff c possibly holds within Δ and the length of Δ is at most n. Note that $\neg \circledast_n c \equiv \odot_n \neg c$.

Because sampling approximates the true value of an environment variable, we must reason about how the value of a variable changes within an interval [16]. For a real-valued variable v, the maximum difference to v in stream s within Δ is given by $(diff.v).\Delta.s$, where:

$$(diff.v).\Delta.s \mathrel{\widehat{=}} \mathsf{let}\ vs = \{t\colon \Delta \bullet (s.t).v\}\ \mathsf{in}\ lub.vs - glb.vs$$

Note that for any real-valued variable v, $st.v \Rrightarrow (diff.v = 0)$.

Sampled real-valued variables in a time band β are related to their true values within an event of β using the *accuracy* of the variable in β [16]. In particular, we let $\mathsf{acc}.v \in TimeBand \to \mathbb{R}^{\geq 0}$ denote the accuracy of variable v in a given time band. The maximum change to v within an event of time band β is an assumption on the environment. To enable this assumption to be stated more succinctly, we define a time band predicate:

$$DIFF.v.\beta \mathrel{\widehat{=}} \boxplus(\ell \leq \rho.\beta \Rightarrow diff.v \leq \mathsf{acc}.v.\beta)$$

$$\amalg \circledast (w_1 \leq \mathit{Empty}.T_1 \vee w_2 \geq \mathit{Full}.T_2 \Rightarrow \mathit{Stopped}_1) \tag{1}$$

$$\amalg \circledast (w_2 \leq \mathit{Empty}.T_2 \Rightarrow \mathit{Stopped}_2) \tag{2}$$

$$\amalg \boxplus (\circledast (w_2 \leq \mathit{Reserve}.T_2 \wedge w_1 \geq \mathit{Reserve}.T_1) \wedge \ell \geq \rho.\Gamma \Rightarrow \boxdot \mathit{On}_1 \vee \mathit{next}.(\boxdot \mathit{On}_1)) \tag{3}$$

$$\amalg \boxplus (\circledast (w_2 > \mathit{Reserve}.T_2 \wedge \mathit{Pressed}) \wedge \ell \geq \rho.\Gamma \Rightarrow \boxdot \mathit{On}_2 \vee \mathit{next}.(\boxdot \mathit{On}_2)) \tag{4}$$

$$\amalg \boxplus (\circledast \neg \mathit{Pressed} \wedge \ell \geq \rho.\Gamma \Rightarrow \boxdot \neg \mathit{On}_2 \vee \mathit{next}.(\boxdot \mathit{On}_2)) \tag{5}$$

Fig. 3. Formalisation of the two-pump system requirements

The lemma below allows one to relate a sampled variable to its values in the environment based on its accuracy and generalises the result in [16].

Lemma 8. *If x and y are real-valued variables and $\gg \in \{\geq, >\}$ then*
$\mathit{DIFF}.x \wedge \mathit{DIFF}.y \wedge \odot_\rho(x - \mathsf{acc}.x \gg y + \mathsf{acc}.y) \Rrightarrow \circledast(x \gg y)$.

Corollary 9. *If x and y are real-valued variables and $\gg \in \{\geq, >\}$ then*
$\mathit{DIFF}.x \wedge \mathit{st}.y \wedge \odot_\rho(x - \mathsf{acc}.x \gg y) \Rrightarrow \circledast(x \gg y)$.

Example 10. The informal requirements of the two-pump system in Section 1 are formalised using the ILTL formulae in Fig. 3.

Safety. We combine **S1** and **S2** as (1) and formalise **S3** as (2). By (1), over the interval in which the program is executing, in all actual (as opposed to apparent) states of the stream, if w_1 (the water level in tank T_1) is below $\mathit{Empty}.T_1$ or w_2 (the water level in tank T_2) is above $\mathit{Full}.T_2$, then pump P_1 must be stopped. Note that the consequent of (1) states that the P_1 has physically come to a stop, which we distinguish from the signal $\neg \mathit{On}_1$ that causes P_1 to stop. Condition (2) is similar.

Progress. Progress properties **P1**, **P2** and **P3** translate into properties (3), (4) and (5). Each of the progress properties involve time bands of the water. For simplicity, we assume that the time bands of the water in both tanks is Γ. Thus, condition (3) states that over the (infinite) interval corresponding to the execution of the program, in any subinterval say Δ of the interval, if it is definitely the case that w_2 is less than or equal to $\mathit{Reserve}.T_2$ and w_1 is greater than or equal to $\mathit{Reserve}.T_1$ for at least the precision of the water time band, then the pump must be turned on either within Δ or some interval that follows Δ. Conditions (4) and (5) are similar.

3.2 Actions

The syntax and semantics of actions are given in Definition 11 and Definition 12, respectively.

Definition 11. *Suppose b is a state predicate, d is a label, $\mathbf{y}$ is a vector of output variables, $\mathbf{E}$ is a vector of expressions that has the same length as $\mathbf{y}$, β is a time band, F is a set of output variables and p is an interval predicate. The abstract syntax of an* action *A is given by:*

$$\begin{array}{l} A ::= d\colon S \mid A_1 \mathbin{[\!]} A_2 \mid A \dagger \beta \\ S ::= b \rightarrow \mathsf{idle} \mid b \rightarrow \mathbf{y} := \mathbf{E} \mid b \rightarrow [\![F \cdot p]\!] \end{array}$$

Definition 12. *For the syntax of actions defined in Definition 11 and a set of output variables V, the function $beh_V\colon A \rightarrow IntvPred$ is defined inductively as follows:*

$$beh_V.(b \rightarrow \mathsf{idle}) \mathrel{\widehat{=}} (\odot b\ ;\ true) \wedge st.V \tag{6}$$

$$beh_V.(b \rightarrow \mathbf{y} := \mathbf{E}) \mathrel{\widehat{=}} (\exists \mathbf{k} \bullet (\odot(b \wedge (\mathbf{k} = \mathbf{E})) \wedge st.\mathbf{y}); \overrightarrow{\mathbf{y}} = \mathbf{k}) \wedge st.(V \backslash \mathbf{y}) \tag{7}$$

$$beh_V.(b \rightarrow [\![F \cdot p]\!]) \mathrel{\widehat{=}} (\odot b \wedge st.V)\ ;\ (p \wedge st.(V \setminus F)) \tag{8}$$

$$beh_V.(d\colon S) \mathrel{\widehat{=}} beh_V.S \wedge \boxtimes(\xi = d) \tag{9}$$

$$beh_V.(A_1 \mathbin{[\!]} A_2) \mathrel{\widehat{=}} beh_V.A_1 \vee beh_V.A_2 \tag{10}$$

$$beh_V.(A \dagger \beta) \mathrel{\widehat{=}} \ell \leq \rho.\beta \wedge beh_V.A \tag{11}$$

The primitive idle is a statement that does nothing but may take time to execute, $\mathbf{y} := \mathbf{E}$ is the *assignment* statement and $[\![F \cdot p]\!]$ is a *specification* statement. Action $d\colon b \rightarrow S$ is a *guarded statement* consisting of statement S with guard b and label d. Action $A_1 \mathbin{[\!]} A_2$ consists of the non-deterministic choice between A_1 and A_2, and $A \dagger \beta$ defines an action A within time band β. Note that action $d\colon b \rightarrow [\![F \cdot p]\!]$ is not directly executable, but needs to be refined to an executable implementation. Further note that we do not allow nested actions, i.e., each guarded action consists of a guard followed by a statement.

We define a function *grd* and shorthand **else** as follows:

$$\begin{aligned} grd.(d\colon b \rightarrow S) &\mathrel{\widehat{=}} b \\ grd.(A_1 \mathbin{[\!]} A_2) &\mathrel{\widehat{=}} grd.A_1 \vee grd.A_2 \\ grd.(A \dagger \beta) &\mathrel{\widehat{=}} grd.A \\ A \ \mathbf{else}\ d\colon S &\mathrel{\widehat{=}} A \mathbin{[\!]} (d\colon \neg grd.A \rightarrow S) \end{aligned}$$

We let $labels.A$ denote the set of all labels within action A. Action $((d\colon b \rightarrow S) \dagger \beta) \mathbin{[\!]} A$ is only well defined if $d \notin labels.A$, i.e., the label of each guarded statement within a non-deterministic choice is unique. If $d \in labels.A$, we let A_d denote the guarded statement labelled d in action A. We reserve a "program counter" variable ξ whose value is the label of the guarded statement that is currently executing [12].

The following lemma allows one to simplify the behaviour of guarded actions.

Lemma 13. $beh_V.((b \rightarrow S) \dagger \beta) \Rrightarrow \odot_{\rho.\beta} b$

Proof. The proof holds because $beh_V.((b \rightarrow S) \dagger \beta) \Rrightarrow \ell \leq \rho.\beta \wedge (\odot b\ ;\ true)$ and $(\odot b\ ;\ true) \Rrightarrow \odot b$. □

Definition 14. *If V is a set of variables, an action A is* refined *by an action C, denoted $A \sqsubseteq_V C$, iff $beh_V.C \Rrightarrow beh_V.A$ holds. If $A \sqsubseteq_V C$ and $C \sqsubseteq_V A$, we write $A \sqsupseteq\sqsubseteq_V C$.*

A refinement may reduce the non-determinism or strengthen the guard of an action.

3.3 Action Systems

Definition 15. *If I is an interval predicate and A is an action such that $grd.A$ holds,* action system $\mathscr{A} \mathrel{\widehat{=}} \mathbf{init}\ I \bullet \mathbf{do}\, A\, \mathbf{od}$ *consists of an* initial property I *followed by an infinite loop that executes action A.*

We use $in.\mathscr{A} \subseteq Var$ and $out.\mathscr{A} \subseteq Var$ to distinguish the input and output variables of action system $\mathscr{A}$ and use $vars.\mathscr{A} \mathrel{\widehat{=}} in.\mathscr{A} \cup out.\mathscr{A}$ for the variables of $\mathscr{A}$. Because we assume true concurrency between an action system $\mathscr{A}$ and its environment, we require that $in.\mathscr{A}$ and $out.\mathscr{A}$ are disjoint. We use $\mathscr{A} \dagger \beta$ as shorthand for the action system whose action executes in time band β, i.e., $\mathscr{A} \dagger \beta \mathrel{\widehat{=}} \mathbf{init}\ I \bullet \mathbf{do}\, A \dagger \beta\, \mathbf{od}$.

Execution of an action system starts in an interval for which the initial property holds for an immediately preceding interval. Then each successive interval of the trace is generated by the behaviour of some guarded action.

Definition 16. *Given $AS \mathrel{\widehat{=}} AdjSeq \times Stream$, the set of all* complete traces *of action system $\mathscr{A}$ with outputs $V \mathrel{\widehat{=}} out.\mathscr{A}$ is given by $\mathsf{Tr}.\mathscr{A}$, where:*

$$\mathsf{Tr}.\mathscr{A} \mathrel{\widehat{=}} \{(z, s){:}\ AS \mid \mathrm{dom}.z = \mathbb{N} \wedge ((z, s) \vdash prev.I \wedge \Box(beh_V.A))\}$$

Thus, for each $(z, s) \in \mathsf{Tr}.\mathscr{A}$, it is assumed that I holds for some interval that precedes $z.0$ and some action executes in each interval $z.i$. Furthermore, because the action systems we consider are non-terminating, z is an infinite sequence.

Given any action system $\mathscr{A}$ and variable $v \in out.\mathscr{A}$, we require a healthiness condition:

$$\mathscr{A} \models \Box(prev.\overrightarrow{v} = \overleftarrow{v}) \tag{12}$$

i.e., the value of v does not change over the boundary between adjoining intervals.

Lemma 17. *If boolean variable x is an output variable, then*

$$beh_V.(x \longrightarrow S) \Rrightarrow prev.\overrightarrow{x} \tag{13}$$

$$(beh_V.(x \longrightarrow S) \Rightarrow \overrightarrow{\neg x}) \Rrightarrow ((beh_V.(x \longrightarrow S))^\omega = beh_V.(x \longrightarrow S)) \tag{14}$$

Proof (13). We first show that $beh_V.(x \longrightarrow S) \Rrightarrow (\odot x \wedge st.x) \,;\, true$. The proofs for $S \in \{\mathsf{idle}, [\![F \cdot p]\!]\}$ are trivial because $st.V$ splits and ';' is monotonic. For $S = (\mathbf{y} := \mathbf{E})$, we have:

$$
\begin{array}{ll}
 & beh_V.(x \rightarrow \mathbf{y} := \mathbf{E}) \\
\Rrightarrow & \quad \text{definition of } beh_V,\ st.V \text{ splits, `; ' is monotonic} \\
 & \exists \mathbf{k} \bullet \odot(x \wedge st.V)\ ;\ true \\
\Rrightarrow & \quad \text{logic} \\
 & (\odot x \wedge st.x)\ ;\ true
\end{array}
$$

Thus, we have:

$$
\begin{array}{ll}
 & beh_V.(x \rightarrow S) \\
\Rrightarrow & \quad \text{proof above} \\
 & (\odot x \wedge st.x)\ ;\ true \\
\Rrightarrow & \quad \text{by Lemma 2 } \circledast(x = true) \text{ (because } x \text{ is boolean) and for any } c,\ \circledast c \Rrightarrow \overleftarrow{c} \\
 & \overleftarrow{x} \\
\Rrightarrow & \quad \text{healthiness condition (12)} \\
 & prev.\overrightarrow{x}
\end{array}
$$

Proof (14). $(beh_V.(x \rightarrow S))^\omega \Lleftarrow beh_V.(x \rightarrow S)$ trivially holds by Lemma 4 (ω-unfolding). Assuming $beh_V.(x \rightarrow S) \Rightarrow \overrightarrow{\neg x}$, we have

$$
\begin{array}{ll}
 & (beh_V.(x \rightarrow S))^\omega \\
\equiv & \quad \text{Lemma 4 } (\omega\text{-unfolding}) \\
 & beh_V.(x \rightarrow S) \vee (beh_V.(x \rightarrow S)\ ;\ (beh_V.(x \rightarrow S))^\omega) \\
\Rrightarrow & \quad \text{assumption } beh_V.(x \rightarrow S) \Rightarrow \overrightarrow{\neg x} \text{ and (13)} \\
 & beh_V.(x \rightarrow S) \vee (\overrightarrow{\neg x}\ ;\ (prev.\overrightarrow{x})^\omega) \\
\equiv & \quad (prev.\overrightarrow{c})^\omega \Rrightarrow (prev.\overrightarrow{c}) \text{ and } \neg(\overrightarrow{\neg c}\ ;\ prev.\overrightarrow{c}) \\
 & beh_V.(x \rightarrow S)
\end{array}
$$

□

Definition 18. *We say that an action system* $\mathscr{A}$ satisfies *an ILTL formula* F *(denoted* $\mathscr{A} \models F$*) iff* $\forall tr \colon \mathsf{Tr}.\mathscr{A} \bullet tr \vdash F$ *holds.*

To show that an action system $\mathscr{A}$ with output context V satisfies $\Box p$, one may show that, $beh_V.A \Rrightarrow p$, i.e., the execution of each guarded action of $\mathscr{A}$ satisfies p.

Theorem 19. $\mathscr{A} \models \Box p$ *if* $beh_{out.\mathscr{A}}.A \Rrightarrow p$.

Proof. The proof follows by definitions 16 and 18 and the definition of $\Box$.

Reactive systems are often structured so that a controller sends signals to the environment, then becomes idle while changes occur in the environment based on the controller signals. Using Theorem 19 to prove $\mathscr{A} \models \Box p$ can be difficult when p includes properties of the environment. Instead, we often use Theorem 21 below which allows one to consider the iterated execution of an action (usually idle) and the properties that held before the action started executing. We first prove a preliminary lemma.

Lemma 20. *If* $(z, s) \in \mathsf{Tr}.\mathscr{A}$ *and* $i \in \mathrm{dom}.z$*, there exists a* $j \in \mathrm{dom}.z$*, where* $j \leq i$ *and a* $d \in label.A$ *such that* $(prev.(I \vee (\exists e \colon label.A \setminus \{d\} \bullet beh_{out.\mathscr{A}}.A_e))).(z.j).s$ *and for all* $j \leq k \leq i$*,* $(beh_{out.\mathscr{A}}.A_d).(z.k).s$*.*

Proof. The proof is trivial for $i = 0$. For any $i \in \mathrm{dom}.z \setminus \{0\}$, because $(z, s) \in \mathsf{Tr}.\mathscr{A}$, there exists a $d \in \mathit{label}.\mathscr{A}$ such that $(\mathit{beh}_{\mathit{out}.\mathscr{A}}.A_d).(z.i).s$. Then, either

- $(\mathit{prev}.(I \vee (\exists e\colon \mathit{label}.A \setminus \{d\} \bullet \mathit{beh}_{\mathit{out}.\mathscr{A}}.A_e))).(z.i).s$ holds, in which case the proof is trivial, or
- $(\mathit{prev}.(\mathit{beh}_{\mathit{out}.\mathscr{A}}.A_d)).(z.i).s$ holds, i.e., $(\mathit{beh}_{\mathit{out}.\mathscr{A}}.A_d).(z.(i-1)).s$ holds and the proof follows by induction. □

For a label $d \in \mathit{label}.\mathscr{A}$, we define

$$\mathit{iterate}.\mathscr{A}.d \mathrel{\widehat{=}} (\mathit{beh}_{\mathit{out}.\mathscr{A}}.A_d)^\omega \wedge \mathit{prev}.(I \vee \exists e\colon \mathit{label}.\mathscr{A} \setminus \{d\} \bullet \mathit{beh}_{\mathit{out}.\mathscr{A}}.A_e)$$

which holds over an interval Δ iff the action of $\mathscr{A}$ labelled d iterates over Δ and in some interval that precedes Δ, either the initialisation I of $\mathscr{A}$ holds or some action different from e executes.

Theorem 21. *If p splits, then $\mathscr{A} \models \Box p$ holds provided that*

$$\mathscr{A} \models \amalg \boxplus (\forall d\colon \mathit{label}.\mathscr{A} \bullet \mathit{iterate}.\mathscr{A}.d \Rightarrow p) \tag{15}$$

Proof. By Lemma 20, for any $(z, s) \in \mathsf{Tr}.\mathscr{A}$ and $i \in \mathrm{dom}.z$,

$$\begin{array}{l} \exists j\colon \mathrm{dom}.z,\ d\colon \mathit{label}.A \bullet \\ \quad j \le i \wedge (\mathit{prev}.(I \vee (\exists e\colon \mathit{label}.A \setminus \{d\} \bullet \mathit{beh}_{\mathit{out}.\mathscr{A}}.A_e))).(z.j).s \wedge \\ \quad (\forall k \bullet j \le k \le i \Rightarrow (\mathit{beh}_{\mathit{out}.\mathscr{A}}.A_d).(z.k).s) \end{array}$$

Hence,

$$(\mathit{iterate}.\mathscr{A}.d).(\textstyle\bigcup_{j \le k \le i} z.k).s$$

holds and therefore by (15), $p.(\bigcup_{j \le k \le i} z.k).s$ holds. Because p splits, $p.(z.i).s$ holds and because i was arbitrarily chosen, $(z, s) \vdash \Box p$ holds. □

Like safety properties, it is often simpler to first translate progress properties into '$\leadsto$' formulae.

Lemma 22. *For any state predicates c_1 and c_2, If $\mathscr{A} \dagger \beta \models \circledast c_1 \leadsto \boxdot c_2$ then $\mathscr{A} \dagger \beta \models \amalg \boxplus (\circledast c_1 \wedge \ell \ge 2\rho.\beta \Rightarrow \boxdot c_2 \vee \mathit{next}.\boxdot c_2)$.*

Proof. For any $(z, s) \in \mathsf{Tr}.(\mathscr{A} \dagger \beta)$, suppose $\Delta = \bigcup \mathrm{ran}.z$ and $\Delta' \subseteq \Delta$ such that $(\circledast c_1 \wedge \ell \ge 2\rho.\beta).\Delta'$. We have:

$$\begin{array}{ll} & (z, s) \in \mathsf{Tr}.(\mathscr{A} \dagger \beta) \wedge (\Delta = \bigcup \mathrm{ran}.z) \wedge (\Delta' \subseteq \Delta) \wedge (\circledast c_1 \wedge \ell \ge 2\rho.\beta).\Delta' \\ \Rightarrow & \quad \text{definition of } \mathscr{A} \dagger \beta \\ & (\forall i\colon \mathrm{dom}.z \bullet (\ell \le \rho.\beta).(z.i)) \wedge (\Delta = \bigcup \mathrm{ran}.z) \wedge \\ & \hfill (\Delta' \subseteq \Delta) \wedge (\circledast c_1 \wedge \ell \ge 2\rho.\beta).\Delta' \\ \Rightarrow & \quad \text{logic} \\ & \exists i\colon \mathrm{dom}.z \bullet (\circledast c_1).(z.i) \wedge z.i \subseteq \Delta' \\ \Rightarrow & \quad \text{assumption } \mathscr{A} \dagger \beta \models \circledast c_1 \leadsto \boxdot c_2 \\ & \exists i\colon \mathrm{dom}.z \bullet z.i \subseteq \Delta' \wedge \exists j\colon \mathrm{dom}.z \bullet i \ge j \wedge (\boxdot c_2).(z.j) \\ \Rightarrow & \quad \text{logic} \\ & (\boxdot c_2).\Delta' \vee (\mathit{next}.(\boxdot c_2)).\Delta' \end{array}$$

□

3.4 Parallel Composition

We use $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ to denote the parallel composition of action systems $\mathscr{A}$ and $\mathscr{B}$. For the program $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ to be well formed, we require:

$$(in.\mathscr{A} \cup out.\mathscr{A}) \cap out.\mathscr{B} = \{\} \tag{16}$$

i.e., $\mathscr{B}$ cannot modify the inputs and outputs of $\mathscr{A}$ but the outputs of $\mathscr{A}$ may be used as inputs to $\mathscr{B}$ and furthermore $\mathscr{A}$ and $\mathscr{B}$ may share inputs. Hence, $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ is not necessarily equivalent to $\mathscr{B} \overrightarrow{\|} \mathscr{A}$.

Within $\mathscr{A} \overrightarrow{\|} \mathscr{B}$, action systems $\mathscr{A}$ and $\mathscr{B}$ execute in a truly concurrent manner. Furthermore, $\mathscr{A}$ and $\mathscr{B}$ may execute in different time bands. Hence, the adjoining intervals defined by the execution of $\mathscr{A}$ and $\mathscr{B}$ are unrelated and we cannot use $\mathsf{Tr}.\mathscr{A}$ and $\mathsf{Tr}.\mathscr{B}$ to define the traces of $\mathscr{A} \overrightarrow{\|} \mathscr{B}$. Instead, we consider the whole interval over which the action systems execute.

Definition 23. *For an action system $\mathscr{A}$, interval Δ and stream s, we say $\mathscr{A}$ executes over Δ in s iff $(exec.\mathscr{A}).\Delta.s$ holds, where:*

$$(exec.\mathscr{A}).\Delta.s \mathrel{\widehat{=}} \exists z \bullet (z, s) \colon \mathsf{Tr}.\mathscr{A} \wedge \Delta = \bigcup \mathrm{ran}.z \tag{17}$$

Definition 24. *Suppose $\mathscr{A}$ and $\mathscr{B}$ are action systems such that (16) holds. Then,*

$$exec.(\mathscr{A} \overrightarrow{\|} \mathscr{B}) \mathrel{\widehat{=}} exec.\mathscr{A} \wedge exec.\mathscr{B}$$

Thus, for any interval Δ and stream s, $(exec.(\mathscr{A} \overrightarrow{\|} \mathscr{B})).\Delta.s$ holds iff there exist traces $(z_1, s) \in \mathsf{Tr}.\mathscr{A}$ and $(z_2, s) \in \mathsf{Tr}.\mathscr{B}$ such that $\Delta = \bigcup \mathrm{ran}.z_1 = \bigcup \mathrm{ran}.z_2$, i.e., it is possible for $\mathscr{A}$ and $\mathscr{B}$ execute in the same overall interval and stream.

A special case of parallel composition is *simple parallelism*, denoted $\mathscr{A} \| \mathscr{B}$, where no output of $\mathscr{A}$ is an input to $\mathscr{B}$ and vice versa, i.e., $\mathscr{A} \| \mathscr{B}$ is defined iff (16) $\wedge$ $(out.\mathscr{A} \cap in.\mathscr{B} = \{\})$ holds. Note that $in.\mathscr{A} \cap in.\mathscr{B}$ may be non-empty, i.e., $\mathscr{A}$ and $\mathscr{B}$ may share inputs. Unlike $\mathscr{A} \overrightarrow{\|} \mathscr{B}$, $\mathscr{A} \| \mathscr{B}$ is equivalent to $\mathscr{B} \| \mathscr{A}$.

4 Deriving Action System Controllers

4.1 Enforced Properties

Our derivation method uses enforced properties [12, 14], which are ILTL formulae that restrict the traces of an action system to those that satisfy the formulae. Enforced properties are temporal formulae and hence may be used to state general properties on the traces, e.g., we may formalise fairness assumptions on the scheduler [12]. We first present enforced properties on actions, which allows finer-grained control over the execution of an action system.

Definition 25. *An action A with* enforced property *$p \in IntvPred$, is an action $A\,!\,p$ and its behaviour in an output context $V \subseteq Var$ is given by $beh_V.(A\,!\,p) \mathrel{\widehat{=}} beh_V.A \wedge p$.*

init I_1 •	**init** I_2 •
do d_0: $true \rightarrow [\![On_1 \cdot true]\!]$	**do** e_0: $true \rightarrow [\![On_2 \cdot true]\!]$
od $\dagger\tau_1 \mathbin{?} SP_1 \mathbin{?} DA.w_1 \wedge DA.w_2$	**od** $\dagger\tau_2 \mathbin{?} SP_2 \mathbin{?} DA.w_2$

Fig. 4. Initial action system for P_1 **Fig. 5.** Initial action system for P_2

Thus, when executing $A \,!\, p$, in addition to behaving as specified by $beh_V.A$, the interval predicate p must also hold. We may represent time bands on actions using enforced properties.

Lemma 26. *For an action A, time band β and $V \subseteq Var$, $A \dagger \beta \sqsubseteq_V A!(\ell \leq \rho.\beta)$*

We obtain straightforward lemmas on actions with enforced properties [17].

Lemma 27. *For an action A, interval predicates p and q and set of variables V,*

(a) $A \,!\, (p \wedge q) \sqsubseteq_V (A \,!\, p) \,!\, q$,
(b) $A \,!\, (p \vee q) \sqsubseteq_V (A \,!\, p) \;[\!]\; (A \,!\, q)$ *and*
(c) *if* $beh_V.A \Rrightarrow p$ *then* $A \,!\, p \sqsubseteq_V A$.

Note that it is possible to enforce unimplementable behaviour, e.g., $beh_V.(A \,!\, false)$. Hence, we typically introduce or strengthen an enforced property to the weakest possible predicate to allow greater flexibility in an implementation.

We extend the concept of enforced properties on actions to enforced properties on action systems, which are specified using ILTL formula.

Definition 28. *If F is an ILTL formula then action system $\mathscr{A}$ with* enforced property *F is denoted $\mathscr{A} \mathbin{?} F$, and its traces are given by $\mathsf{Tr}.(\mathscr{A} \mathbin{?} F) \mathrel{\widehat{=}} \{tr \colon \mathsf{Tr}.\mathscr{A} \mid tr \vdash F\}$.*

Thus, although $\mathsf{Tr}.\mathscr{A}$ may contain traces that do not satisfy F, by definition, $\mathscr{A} \mathbin{?} F$ is guaranteed to satisfy F. We have used enforced properties to develop theories of refinement, where the enforced properties are LTL formulae [12], relational LTL formulae [14] and ILTL formulae [17].

Example 29. We specify an initial action system controller for the two-pump system in Section 1. The initial actions are liberal and allow arbitrary modification of signals On_1 and On_2. However, execution of the action systems are constrained by their enforced properties, which ensure that the programs are correct with respect to the given properties. We develop the system as the simple parallel composition between the controllers for pumps P_1 and P_2, which allows the pumps to be controlled independently (see Fig. 4 and Fig. 5). We assume that the time band of pump P_i is ϕ_i and recall that the time band of the controller for P_i is τ_i. Thus, each iteration of the **do** loop of the controller for pump P_1 can be completed within an interval of length $\rho.\tau_1$ and events of P_1 take at most $\rho.\phi_1$ time (similarly for P_2). The action for the initial version of the controller of P_1 is d_0: $true \rightarrow [\![On_1 \cdot true]\!]$, where d_0 is a label, guard $true$

never blocks the action from executing and $[\![On_1 \cdot true]\!]$ allows the output On_1 to be set to true or false non-deterministically. We collate the properties on P_1 and P_2 as ILTL formulae SP_1 and SP_2, respectively:

$$SP_1 \mathrel{\widehat{=}} (1) \wedge (3)$$
$$SP_2 \mathrel{\widehat{=}} (2) \wedge (4) \wedge (5)$$

Both SP_1 and SP_2 are introduced to the program in Fig. 4 and Fig. 5 as enforced properties, which guarantees that the programs are correct with respect to these requirements. That is, although the program without the enforced properties is able to arbitrarily change the values of On_1 and On_2, by including the enforced properties, the traces of the controllers are guaranteed to satisfy the required properties SP_1 and SP_2, respectively.

Enforced properties can also be used to specify assumptions about the behaviour of the environment. Here, we use enforced properties to ensure that the accuracies of w_1 and w_2 bound the maximum possible change to w_1 and w_2 in any time band. For $i \in \{1, 2\}$, we define:

$$DA.w_i \mathrel{\widehat{=}} \text{II}(DIFF.w_i.\phi_i \wedge DIFF.w_i.\tau_i)$$

Hence, the maximum difference between two values of w_1 in events of time bands ϕ_1 and τ_1 are bounded by the accuracy of w_1 in ϕ_1 and τ_1, respectively. By using II in the formula above, we are stating that $DIFF.w_i.\phi_i \wedge DIFF.w_i.\tau_i$ holds over the whole interval in which the action systems executes.

The controllers for P_1 and P_2 are only partially developed and actions d_0 and e_0 are not yet executable. Hence, we perform a series of refinements to obtain an implementation that can be executed.

4.2 Action System Refinement

Our method of derivation allows programs to be developed in an incremental manner. In particular, we calculate the effect of (partially) developed actions on the enforced properties, which generates new properties and actions. However unlike Dijkstra [11], Feijen/van Gasteren [19] and Dongol/Mooij [18], we disallow arbitrary modifications to the program; each change must be justified using a lemma/theorem that ensures that the previous version is refined [14, 17].

Definition 30. *Action system $\mathscr{C}$* refines *$\mathscr{A}$ (denoted $\mathscr{A} \sqsubseteq \mathscr{C}$) iff $exec.\mathscr{C} \Rightarrow exec.\mathscr{A}$.*

Thus, for any interval Δ and stream s, if it is possible to execute $\mathscr{C}$ within Δ in s, then it must be possible to execute $\mathscr{A}$ within Δ in s. The lemma below provides a sufficient condition for action system refinement [17].

Lemma 31. *Suppose $\mathscr{A} \mathrel{\widehat{=}}$ **init** I_A; **do** A **od** and $\mathscr{C} \mathrel{\widehat{=}}$ **init** I_C; **do** C **od**. Then $\mathscr{A} \sqsubseteq \mathscr{C}$ holds if $out.\mathscr{C} \subseteq out.\mathscr{A}$ and $(I_C \Rightarrow I_A) \wedge (A \sqsubseteq_{out.\mathscr{C}} C)$.*

Lemma 32. *If $\mathscr{A} \mathrel{\widehat{=}}$ **init** I $\bullet$ **do** A **od**, then $\mathscr{A} \sqsubseteq$ **init** I $\bullet$ **do** $A \mathbin{[\!]} B$ **od** if $A \sqsubseteq_V B$.*

The following lemma allows trace refinement of action systems with enforced properties [17].

Lemma 33. *For action systems $\mathscr{A}$ and $\mathscr{C}$ and ILTL formulae F and F',*

(a) $\mathscr{A} \sqsubseteq \mathscr{A} \,?\, true$ *holds,*
(b) $\mathscr{A} \,?\, F \sqsubseteq \mathscr{A} \,?\, F'$ *holds if* $F' \Rrightarrow F$*,*
(c) $\mathscr{A} \,?\, F \sqsubseteq \mathscr{C} \,?\, F$ *holds if* $\mathscr{A} \sqsubseteq \mathscr{C}$ *and*
(d) $\mathscr{A} \,?(F \wedge F') \sqsubseteq \mathscr{A} \,?\, F$ *holds if* $\mathscr{A} \,?\, F \models_V F'$*.*

Thus, introducing *true* or strengthening existing enforced properties results in a refinement. Furthermore, if an action system without an enforced property refines another, then the refinement holds with the enforced property included. An enforced property $F \wedge F'$ may be simplified to F if the action system $\mathscr{A} \,?\, F$ satisfies F'.

Lemma 34. *For an action system $\mathscr{A}$ and time band β, $\mathscr{A} \dagger \beta \sqsubseteq \mathscr{A} \,?\, \Box(\ell \leq \rho.\beta)$.*

Example 35. Using Lemma 22, given the following relationship between Γ and the controller time bands τ_1 and τ_2,

$$\left(\rho.\tau_1 \leq \frac{\rho.\Gamma}{2}\right) \wedge \left(\rho.\tau_2 \leq \frac{\rho.\Gamma}{2}\right) \tag{18}$$

progress properties (3), (4) and (5) are implied by leads-to properties (19), (20) and (21), respectively (see below).

$$\circledast(w_2 \leq Reserve.T_2 \wedge w_1 \geq Reserve.T_1) \rightsquigarrow \boxdot On_1 \tag{19}$$

$$\circledast(w_2 > Reserve.T_2 \wedge Pressed) \rightsquigarrow \boxdot On_2 \tag{20}$$

$$\circledast\neg Pressed \rightsquigarrow \boxdot\neg On_2 \tag{21}$$

By (19), if it is definitely the case that the water level in tank T_2 is below $Reserve.T_2$ and the water level in tank T_1 is above $Reserve.T_1$, then pump P_1 must eventually be turned on. Conditions (20) and (21) are similar.

Unlike Aichernig et al, we specify maintenance properties to ensure that the pump does not arbitrarily change its state.

M1. If P_1 has been turned on, then it must remain on unless w_1 is possibly below $Reserve.T_1$ or w_2 is possibly above $Limit.T_2$.
M2. If P_1 has been turned off, then it must remain off unless w_1 is possibly above $Reserve.T_1$ and w_2 is possibly below $Limit.T_2$.
M3. If P_2 has been turned on, then it must remain on unless w_2 is possibly below $Reserve.T_2$ or the button B is possibly released.
M4. If P_2 has been turned on, then it must remain off unless B is possibly pressed and w_2 is possibly above $Reserve.T_2$.

Note that condition **M1** must allow pump P_1 to be turned off before the water level drops to empty because by **S1**, the pump must (physically) be off if the

init I_1 •	init I_2 •
do d_0: $true \rightarrow [\![On_1 \cdot true]\!]$	do e_0: $true \rightarrow [\![On_2 \cdot true]\!]$
od $\dagger\tau_1$? SPM_1 ? $DA.w_1 \wedge DA.w_2$	od $\dagger\tau_2$? SPM_2 ? $DA.w_2$

Fig. 6. Refined action system for P_1 **Fig. 7.** Refined action system for P_2

water level ever reaches $Empty.T_1$. The maintenance properties are interpreted in the controller time band, e.g., within **M1**, "w_1 is possibly below $Reserve.T_1$" is interpreted as "w_1 is possibly below $Reserve.T_1$ for an event of the controller time band" (see Example 10).

Properties **M1**, **M2**, **M3** and **M4** translate directly to maintained unless properties (22), (23), (24) and (25), respectively (see below).

$$On_1\ \mathcal{M}\ (w_1 \leq Reserve.T_1 \vee w_2 \geq Limit.T_2) \tag{22}$$

$$\neg On_1\ \mathcal{M}\ (w_1 > Reserve.T_1 \wedge w_2 < Limit.T_2) \tag{23}$$

$$On_2\ \mathcal{M}\ (\neg Pressed \vee w_2 \leq Reserve.T_2) \tag{24}$$

$$\neg On_2\ \mathcal{M}\ (Pressed \wedge w_2 > Reserve.T_2) \tag{25}$$

By (22) if signal On_1 holds at the end of a preceding interval at any point during the program's execution, then On_1 must continue to hold unless there is an interval in which $w_1 \leq Reserve.T_1$ or $w_2 \geq Full.T_2$ is possibly true. Conditions (23), (24) and (25) are similar.

We define:

$$SPM_1 \mathrel{\widehat{=}} (1) \wedge (19) \wedge (22) \wedge (23)$$

$$SPM_2 \mathrel{\widehat{=}} (2) \wedge (20) \wedge (21) \wedge (24) \wedge (25)$$

Then using Lemma 33, we replace SP_1 in Fig. 4 by SPM_1 to obtain the refined action system in Fig. 6. Similarly, the action system in Fig. 5 is refined by the one in Fig. 7.

Note that using a sampling logic over intervals allows one to reason about transient behaviour and hence avoid formalisation of unimplementable specifications. We say a state predicate is *transient* in a stream if the predicate only holds for a brief (e.g., an attosecond) amount of time, whereby it is not possible to reliably detect that the predicate held [14, 17]. For example, property (20) without using a sampling logic would be stated using LTL [26] as $w > Reserve.T_2 \wedge Pressed \leadsto On_2$ [10, 14]. Such a property is unimplementable if the state predicate $w > Reserve.T_2 \wedge Pressed$ on the left of $\leadsto$ is transient (which can happen if the button is quickly pressed then released). In this paper, because we use $\circledast$ on the left of '$\leadsto$' within (19), (20) and (21), the corresponding state predicate must hold for all *apparent* states, i.e., the state predicate on the left of '$\leadsto$' is guaranteed to be detected by the controller provided that the variables are sampled within the sampling interval. For example, in (20), we can guarantee that $w_2 > Reserve.T_2 \wedge Pressed$ is detected by the controller

regardless of when the variables w_2 and *Pressed* are sampled within the sampling interval. If $\odot(w \leq Reserve.T_2 \vee \neg Pressed)$ holds over a sampling interval, i.e., it is possible for the controller not to detect $w > Reserve.T_2 \wedge Pressed$, then the controller is not required to turn P_2 on.

We define trace refinement of a parallel composition of action systems as follows.

Definition 36. *For action systems* $\mathscr{A}$, $\mathscr{A}'$, $\mathscr{B}$ *and* $\mathscr{B}'$, *such that* $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ *and* $\mathscr{A}' \overrightarrow{\|} \mathscr{B}'$ *are well formed, we say* $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ *is* trace refined *by* $\mathscr{A}' \overrightarrow{\|} \mathscr{B}'$ *(i.e.,* $\mathscr{A} \overrightarrow{\|} \mathscr{B} \sqsubseteq \mathscr{A}' \overrightarrow{\|} \mathscr{B}'$*) iff* $exec.(\mathscr{A}' \overrightarrow{\|} \mathscr{B}') \Rrightarrow exec.(\mathscr{A} \overrightarrow{\|} \mathscr{B})$.

Lemma 37. *For action systems* $\mathscr{A}$, $\mathscr{A}'$, $\mathscr{B}$ *and* $\mathscr{B}'$, *such that* $\mathscr{A} \overrightarrow{\|} \mathscr{B}$ *and* $\mathscr{A}' \overrightarrow{\|} \mathscr{B}'$ *are well formed,* $\mathscr{A} \overrightarrow{\|} \mathscr{B} \sqsubseteq \mathscr{A}' \overrightarrow{\|} \mathscr{B}'$ *holds iff both* $\mathscr{A} \sqsubseteq \mathscr{A}'$ *and* $\mathscr{B} \sqsubseteq \mathscr{B}'$.

5 Hardware/Software Interaction

One of the benefits of using time bands is that it simplifies reasoning about the interaction between hardware and software. Namely, we may formalise delays in turning signals on/off within a digital controller, and the effect of the signal in the physical world. In this section, we present some high-level interval predicates for expressing properties of hardware/software interaction and theorems for reasoning about such systems.

We assume that the controller sends a boolean signal *sig* to achieve a boolean effect *eff* in the environment, where *eff* occurs if *sig* holds continuously over a long enough interval (i.e., not instantaneously). There are often delays in the controller setting *sig* and in the environment reacting to *sig* to achieve *eff*. Furthermore, the time bands of the controller and environment may differ (e.g., if the environment consists of physical hardware). Thus, we define a time band predicate signal that formalises the behaviour of the controller setting the signal *sig* (in the time band of the controller) and a time band predicate effect that formalises the relationship between *sig* and *eff* (in the time band of the environment). Within the controller, we may distinguish between actions that set and maintain *sig*, where the *sig* is set to true due to a trigger c. Signal *sig* is set to true by a single action, but is maintained by the iterated execution of several "maintenance" actions (e.g., idle). Hence, given time bands β and γ, we define:

$$\mathsf{signal}(c, sig).\beta \mathrel{\widehat{=}} \mathsf{if}\ prev.\overrightarrow{\neg sig}\ \mathsf{then}(\ell \leq \rho.\beta \wedge \circledast c \wedge \overrightarrow{sig})\ \mathsf{else}\ (prev.\overrightarrow{c} \wedge \circledast sig)$$
$$\mathsf{effect}(sig, eff).\gamma \mathrel{\widehat{=}} \circledast sig \Rightarrow \mathsf{if}\ prev.\overrightarrow{eff}\ \mathsf{then}\ \circledast eff\ \mathsf{else}(\ell \leq \rho.\gamma : \circledast eff)$$

which describe relationships between the signal predicate *sig* and the corresponding effect predicate *eff*. If $\mathsf{signal}(c, sig).\beta$ holds, then if *sig* does not hold, then $\circledast c$ is guaranteed to hold and *sig* is guaranteed to be established within the precision of β, otherwise c must hold initially and *sig* must hold continuously in the interval. In essence, this ensures that the precision of setting *sig* to true is $\rho.\beta$. For example, the controller for P_2 sends a signal On_2 (i.e., *sig* is On_2)

and has effect *Stopped*$_2$ (i.e., *eff* is *Stopped*$_2$). The action that sets On_2 to true may guarantee a condition c that actions that maintain On_2 can rely on holding initially. We calculate c for specific problem instances using Theorem 39 below.

If $\mathsf{effect}(sig, \mathit{eff}).\gamma$ holds, then provided that sig is continuously true over an interval, *eff* continues to hold if it held at the right end of the previous interval, otherwise there is a delay of at most $\rho.\gamma$ before $\circledast \mathit{eff}$ holds. This models the fact that the effect takes place within the precision of the given time band.

Real-time controllers must often sample a variable in the environment and control a physical mechanism so that the effect of the mechanism occurs before the value of the variable in the environment drops below a critical level. The following lemma provides conditions that allow one to show that in all states of the stream, either the value of a variable v is above a critical level C, or the physical effect *eff* has taken place.

Lemma 38. *Suppose v is a real-valued variable, $\gg \in \{>, \geq\}$, sig and eff are state predicates, and $T, C \in \mathbb{R}$ are constants. Then*

$$(T \geq C + \mathsf{acc}.v) \wedge \mathit{DIFF}.v \wedge \mathsf{effect}(sig, \mathit{eff}) \wedge \overleftarrow{v \gg T} \wedge \circledast sig \Rrightarrow \circledast(v \gg C \vee \mathit{eff})$$

Proof.

$$\begin{aligned}
& (T \geq C + \mathsf{acc}.v) \wedge \mathit{DIFF}.v \wedge \mathsf{effect}(sig, \mathit{eff}) \wedge \overleftarrow{v \gg T} \wedge \circledast sig \\
\Rrightarrow\ & \quad \text{definition of } \mathsf{effect}(sig, \mathit{eff}) \text{ using } \circledast sig \\
& (T \geq C + \mathsf{acc}.v) \wedge \mathit{DIFF}.v \wedge \overleftarrow{v \gg T} \wedge (\circledast \mathit{eff} \vee (\ell \leq \rho : \circledast \mathit{eff})) \\
\Rrightarrow\ & \quad \text{Corollary 9} \\
& \circledast \mathit{eff} \vee (\circledast(v \gg C) : \circledast \mathit{eff}) \\
\Rrightarrow\ & \quad (\circledast c_1 : \circledast c_2) \Rrightarrow \circledast(c_1 \vee c_2) \\
& \circledast(v \gg C \vee \mathit{eff})
\end{aligned}$$

□

We use Lemma 38 in the proof of the theorem below, which defines a relationship between signals and their effects in the context of an action system controller. In particular, it provides conditions necessary for a property of the form $\Box\circledast(v \gg C \vee \mathit{eff})$ to be established.

Theorem 39. *If v is a continuous variable, β and γ are time bands, $\gg \in \{>, \geq\}$, sig and eff are state predicates, $O \mathrel{\widehat{=}} out.\mathscr{A}$ and $T, C \in \mathbb{R}$ are constants such that $T \geq C + max(\mathsf{acc}.v.\beta, \mathsf{acc}.v.\gamma)$, then $\mathscr{A} \dagger \beta \models \Box\circledast(v \gg C \vee \mathit{eff})$ holds if:*

$$\mathscr{A} \dagger \beta \models \amalg \boxplus (\mathsf{effect}(sig, \mathit{eff})).\gamma \tag{26}$$

$$\mathscr{A} \dagger \beta \models \amalg \boxplus \begin{pmatrix} \forall d : label.\mathscr{A} \bullet iterate.\mathscr{A}.d \Rightarrow \\ \qquad (\odot_{\rho.\beta}(v \gg C + \mathsf{acc}.v.\beta))^{\omega} \vee (\mathsf{signal}(v \gg T, sig)).\beta \end{pmatrix} \tag{27}$$

$$\mathscr{A} \dagger \beta \models \amalg(\mathit{DIFF}.v.\beta \wedge \mathit{DIFF}.v.\gamma) \tag{28}$$

Proof. By Theorem 21, because $\circledast c$ splits, $\mathscr{A} \dagger \beta \models \Box\circledast(v \gg C \vee \mathit{eff})$ holds if

$$\mathscr{A} \dagger \beta \models \amalg \boxplus (\forall d : label.\mathscr{A} \bullet iterate.\mathscr{A}.d \Rightarrow \circledast(v \gg C \vee \mathit{eff}))$$

Using (27) and transitivity, the condition above holds if we prove

$$\mathscr{A} \dagger \beta \models \amalg \boxplus \begin{pmatrix} (\odot_{\rho.\beta}(v \gg C + \mathsf{acc}.v.\beta))^{\omega} \vee (\mathsf{signal}(v \gg T, sig)).\beta \Rightarrow \\ \qquad \circledast(v \gg C \vee \mathit{eff}) \end{pmatrix} \tag{29}$$

We have

$$
\begin{array}{ll}
 & (29) \\
= & \quad \text{logic} \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus ((\odot_{\rho.\beta}(v \gg C + \mathsf{acc}.v.\beta))^{\omega} \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \wedge \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus ((\mathsf{signal}(v \gg T, \mathit{sig})).\beta \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \\
\Leftarrow & \quad \text{Corollary 9 and (28)} \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus ((\mathsf{signal}(v \gg T, \mathit{sig})).\beta \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \\
= & \quad \mathit{sig} \text{ is a boolean, logic} \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus ((\mathsf{signal}(v \gg T, \mathit{sig})).\beta \wedge \mathit{prev}.\overrightarrow{\mathit{sig}} \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \wedge \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus ((\mathsf{signal}(v \gg T, \mathit{sig})).\beta \wedge \mathit{prev}.\overrightarrow{\neg \mathit{sig}} \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \\
\Leftarrow & \quad \text{definition of } \mathsf{signal} \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus (\mathit{prev}.(\overrightarrow{v} \gg T) \wedge \circledast \mathit{sig} \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \wedge \\
 & \mathscr{A} \dagger \beta \models \amalg \boxplus (\ell \leq \rho.\beta \wedge \circledast(v \gg T) \wedge \overrightarrow{\mathit{sig}} \Rightarrow \circledast(v \gg C \vee \mathit{eff})) \\
\Leftarrow & \quad \text{first conjunct: (26), (28) and } T \geq C + \mathit{max}(\mathsf{acc}.v.\beta, \mathsf{acc}.v.\gamma) \\
 & \quad \text{second conjunct: } \circledast(v \gg T) \Rightarrow \circledast(v \gg C) \\
 & \mathit{true}
\end{array}
$$

□

By (26) the effect predicate holds between *sig* and *eff* within time band γ. By (27), either it is possible to sample that the value of v is above $C + \mathsf{acc}.v.\beta$ in each sampling interval, or *sig* is set to true. By (28) the difference between two values of v within events of time bands β and γ does not exceed the accuracy of v in β and γ, respectively. Conditions (26) and (28) are usually properties of the environment of the action system and hence are introduced as enforced properties. On the other hand, properties such as (27) must be guaranteed by the actions of the action system.

Example 40. We demonstrate the use of Theorem 39 and derive the necessary conditions on the control signal On_2 and the state of pump P_2 to prove (2). Because $\circledast c$ joins for any state predicate c, using Lemma 7, condition (2) holds if

$$\Box\circledast(w_2 \leq \mathit{Empty}.T_2 \Rightarrow \mathit{Stopped}_2)$$

which by logic is equivalent to

$$\Box\circledast(w_2 > \mathit{Empty}.T_2 \vee \mathit{Stopped}_2)$$

Using Theorem 39 this holds if for some constant T, (30) $\wedge$ (31) $\wedge$ $\Box$(32) holds (see Fig. 8). Condition (28) is satisfied by enforced property DA_{w_2} in the program in Fig. 7. Condition (30) establishes a relationship between T and $\mathit{Empty}.T_2$ based on the accuracy of the water in time bands τ_2 and ϕ_2. By (31), in any subinterval of the action system's execution, the pump must stop if the length of the subinterval is $\rho.\phi_2$ or greater and signal $\neg On_2$ holds continuously. Condition (32) states that either it is possible to iteratively sample $w_2 > \mathit{Empty}.T_2 + \mathsf{acc}.w_2.\tau_2$ or the signal predicate holds, which ensures that $\neg On_2$ holds within an interval of length $\rho.\tau_2$ and $\neg On_2$ is maintained if it already holds.

$$T \geq Empty.T_2 \;+\; max(\mathsf{acc}.w_2.\tau_2, \mathsf{acc}.w_2.\phi_2) \tag{30}$$

$$\text{II} \boxplus (\mathsf{effect}(\neg On_2, Stopped_2)).\phi_2 \tag{31}$$

$$\text{II} \boxplus \left(\begin{array}{l} \forall d\colon label.\mathscr{A} \bullet iterate.\mathscr{A}.d \;\Rightarrow\; (\odot_{\rho.\tau_2}(w_2 > Empty.T_2 + \mathsf{acc}.w_2.\tau_2)) \vee \\ \hfill (\mathsf{signal}(w_2 \geq T, \neg On_2)).\tau_2 \end{array} \right) \tag{32}$$

$$\circledast(w_2 > Reserve.T_2 \wedge Pressed) \;\Rightarrow\; \overrightarrow{On_2} \tag{33}$$

$$\circledast\neg Pressed \;\Rightarrow\; \overrightarrow{\neg On_2} \tag{34}$$

$$prev.\overrightarrow{On_2} \Rightarrow \boxast On_2 \;\vee\; \odot(\neg Pressed \vee w_2 \leq Reserve.T_2) \tag{35}$$

$$prev.\overrightarrow{\neg On_2} \Rightarrow \boxast\neg On_2 \;\vee\; \odot(Pressed \wedge w_2 > Reserve.T_2) \tag{36}$$

Fig. 8. Transformed properties

6 Example: Two-Tank Pump System

Due to Lemma 37, we may refine the system by refining the controllers of each pump separately. In this paper, we focus on the controller for pump P_2; the controller for P_1 may be derived in a similar manner.

6.1 Formulae Transformation

We first transform the general ILTL formula SPM_2, into formulae with conjuncts of the form $\Box p$ where p is an interval predicate [14, 17]. We may prove that the program satisfies $\Box p$ using Theorem 21.

Safety. The transformation to satisfy safety condition (2) is given in Example 40.

Progress. We may ensure the right hand side of $\rightsquigarrow$ in (20) and (21) holds immediately, i.e., without any intermediate intervals. Hence, we obtain (33) and (34), where (20) and (21) hold if $\Box$(33) and $\Box$(34) hold, respectively. By (33) if it is definitely the case that the water in tank T_2 is above $Reserve.T_2$ and the button is pressed, then signal On_2 must be set to true. Condition (34) is similar.

Maintenance. Using Lemma 7, (24) and (25) hold if $\Box$(35) and $\Box$(36) hold, respectively. By (35), if On_2 holds at the end of the previous interval, then it must hold throughout the current interval, or it must be possible detect that *Pressed* does not hold or the water in tank T_2 is below $Reserve.T_2$. Condition (36) is similar. We distinguish between the properties of the controller of P_2 and those of the its environment and define:

$$RefSPM_2 \mathrel{\widehat{=}} (32) \wedge (33) \wedge (34) \wedge (35) \wedge (36)$$

Using Lemma 33, we replace enforced property in SPM_2 within the controller for P_2 in Fig. 7 by $\Box RefSPM_2 \wedge$ (31) and we obtain the program in Fig. 9.

---

do e_0: $true \longrightarrow [\![On_2 \cdot true]\!]$
od $\dagger\tau_2 \,?(\Box RefSPM_2 \wedge (31))\,?\, DA.w_2$

Fig. 9. Replace SPM_2

6.2 Action Calculation

Using Theorem 21, for each action a that we introduce, we check that $(beh_V.a)^\omega \Rrightarrow RefSPM_2$ holds.

Turning Pump P_2 Off. The controller achieves this by setting the output signal On_2 to false under a guard b_1 and enforced property p_1. Thus, we check the effect of action $(e_1\colon b_1 \longrightarrow On_2 := false)\,!\,p_1$, where e_1 is a fresh label, b_1 is a state predicate and p_1 is an interval predicate. Both b_1 and p_1 are instantiated below when calculating the conditions for e_1 to establish $RefSPM_2$. To prove (32), we assert $b_1 \Rightarrow On_2$ and obtain:

$$\begin{array}{ll}
 & (beh_V.(e_1\,!\,p_1))^\omega \Rrightarrow (32) \\
\Leftarrow & \quad (14) \text{ of Lemma 17 because } b_1 \Rightarrow On_2 \\
 & beh_V.(e_1\,!\,p_1) \Rrightarrow (32) \\
\Leftarrow & \quad (13) \text{ of Lemma 17 using } b_1 \Rightarrow On_2\text{, strengthen consequent} \\
 & beh_V.e_1 \wedge p_1 \wedge prev.\overrightarrow{On_2} \Rrightarrow (\mathsf{signal}(w_2 \geq T, \neg On_2)).\tau_2 \\
\Leftarrow & \quad \text{definition of } \mathsf{signal}\text{, use } prev.\overrightarrow{On_2} \\
 & beh_V.e_1 \wedge p_1 \Rrightarrow \ell \leq \rho.\tau_2 \wedge \circledast(w_2 \geq T) \wedge \overrightarrow{\neg On_2} \\
\Leftarrow & \quad \text{definition of } beh_V,\ \ell \leq \rho.\tau_2 \text{ is implicit by } \dagger\tau_2 \\
 & \overrightarrow{\neg On_2} \wedge p_1 \Rrightarrow \circledast(w_2 \geq T) \wedge \overrightarrow{\neg On_2} \\
\Leftarrow & \quad \text{logic, weaken antecedent} \\
 & p_1 \Rrightarrow \circledast(w_2 \geq T)
\end{array}$$

By asserting $b_1 \Rightarrow w_2 \leq Reserve.T_2 \vee \neg Pressed$, condition (33) may be discharged using Lemma 13. Condition (34) is trivial because $beh_V.e_1 \Rrightarrow \overrightarrow{\neg On_2}$. Condition (35) is trivial by $b_1 \Rightarrow w_2 \leq Reserve.T_2 \vee \neg Pressed$ from above and Lemma 13, $\odot(w_2 \leq Reserve.T_2 \vee \neg Pressed)$ holds. Condition (36) holds because $b_1 \Rightarrow On_2$ holds, which by (13) of Lemma 17 implies $prev.\overrightarrow{\neg On_2}$. Thus, our derived action is

$$(e_1\colon On_2 \wedge (\neg Pressed \vee w_2 \leq Reserve.T_2) \longrightarrow On_2 := false)\,!\,\circledast(w_2 \geq T)$$

This action contains an enforced property $\circledast(w_2 \geq T)$, which we discharge at a later stage of the derivation.

Turning Pump P_2 On. As with action e_1 above, the template for turning the pump on is $(e_2\colon b_2 \longrightarrow On_2 := true)\,!\,p_2$. Because the calculations for e_2 to satisfy $RefSPM_2$ are similar to those for e_1 above, we only briefly describe the necessary modifications. For (32), we assert $b_2 \Rightarrow w_2 > Empty.T_2 + \mathsf{acc}.w_2.\tau_2$, condition (33) is trivially discharged because $\overrightarrow{On_2}$ holds, condition (34) is satisfied by asserting $b_2 \Rightarrow Pressed$, which by Lemma 13 ensures $\odot Pressed$ and (35) is

satisfied by asserting $b_2 \Rightarrow \neg On_2$, which by (13) of Lemma 17 implies $prev.\overrightarrow{On_2}$. Finally, for (36), we assert $b_2 \Rightarrow w_2 > Reserve.T_2$ (because $b_2 \Rightarrow Pressed$ already holds) and by assuming

$$Reserve.T_2 \geq Empty.T_2 + \mathsf{acc}.w_2.\tau_2 \tag{37}$$

we obtain action:

$$e_2\colon \neg On_2 \wedge Pressed \wedge w_2 > Reserve.T_2 \longrightarrow On_2 := true$$

idle Action. Because we are developing a reactive system, **idle** is executed when both e_1 and e_2 are disabled, i.e., when $\neg grd.e_1 \wedge \neg grd.e_2$ holds. We may simplify the (iterated) behaviour of the **idle** action as follows:

$$\begin{array}{ll}
& (beh_V.(e_3\colon \neg grd.e_1 \wedge \neg grd.e_2 \longrightarrow \mathbf{idle}))^\omega \\
\Rightarrow & \quad \text{definition of } beh_V,\ (p_1 \wedge p_2)^\omega \Rightarrow p_1^\omega \wedge p_2^\omega \\
& \left(\odot_{\rho.\tau_2}\begin{pmatrix}(\neg On_2 \vee (w_2 > Reserve.T_2 \wedge Pressed) \\ \wedge\ (w_2 \leq Reserve.T_2 \vee On_2 \vee \neg Pressed)\end{pmatrix}\right)^\omega \wedge (st.On_2)^\omega \\
\Rightarrow & \quad (st.On_2)^\omega \equiv st.On_2 \text{ and case analysis on } prev.\overrightarrow{On_2}\text{, use } st.On_2\text{,} \\
& \text{then simplify} \\
& \left(\left(\odot_{\rho.\tau_2}\left(w_2 > Reserve.T_2 \wedge Pressed\right)\right)^\omega \wedge prev.\overrightarrow{On_2} \wedge \circledast On_2\right) \vee \\
& \left(\left(\odot_{\rho.\tau_2}\left(w_2 \leq Reserve.T_2 \vee \neg Pressed\right)\right)^\omega \wedge prev.\overrightarrow{\neg On_2} \wedge \circledast \neg On_2\right)
\end{array}$$

Hence, we can prove that $(beh_V.(e_3\colon \neg grd.e_1 \wedge \neg grd.e_2 \longrightarrow \mathbf{idle}))^\omega$ satisfies $RefSPM_2$ by proving that both of the interval predicates below satisfy $RefSPM_2$:

$$(\odot_{\rho.\tau_2}(w_2 > Reserve.T_2 \wedge Pressed))^\omega \wedge prev.\overrightarrow{On_2} \wedge \circledast On_2 \tag{38}$$

$$(\odot_{\rho.\tau_2}(w_2 \leq Reserve.T_2 \vee \neg Pressed))^\omega \wedge prev.\overrightarrow{\neg On_2} \wedge \circledast \neg On_2 \tag{39}$$

Both (38) and (39) trivially satisfy (33), (34), (35) and (36). Condition (38) trivially satisfies (32) because by (37), $Reserve.T_2 \geq Empty.T_2 + \mathsf{acc}.w_2.\tau_2$ holds. To show that (39) satisfies (32), because (39) implies that $\circledast \neg On_2$ holds, we use Lemma 33 to introduce the following enforced property to the program:

$$\Box(\circledast(\xi = e_3) \wedge prev.(\overrightarrow{\neg On_2}) \Rightarrow prev.(\overrightarrow{w_2 \geq T})) \tag{40}$$

which states that if e_3 is being executed and signal On_2 is disabled at the end of the preceding interval, then the water level in tank T_2 must be above T at the end of the preceding interval. This ensures that (39) satisfies $(\mathsf{signal}(w_2 \geq T, \neg On_2)).\tau_2$.

We prove (40) using Theorem 21, which gives us the following proof obligation:

$$\begin{array}{c}(beh_V.e_3)^\omega \wedge \circledast(\xi = e_3) \wedge \\ prev.(\overrightarrow{I} \vee (\exists k\colon \{e_1, e_2\} \bullet beh_V.k))\end{array} \Rightarrow prev.(\overrightarrow{On_2 \vee w_2 \geq T}) \tag{41}$$

For $\overrightarrow{I}$, we require that $I \Rightarrow w_2 \geq T$, for $k = e_1$, the proof holds by enforced property $\overrightarrow{w_2 \geq T}$ in e_1 and for $k = e_2$ the proof holds because $beh_V.e_2$ implies $\overrightarrow{On_2}$.

$$\begin{array}{lrl}
\mathbf{do}\ e_0: & \mathit{true} & \rightarrow [\![\mathit{On}_2 \cdot \mathit{true}]\!] \\
[\!]\ \ e_1: & \mathit{On}_2 \wedge (\neg \mathit{Pressed} \vee w_2 \leq \mathit{Reserve}.T_2) & \rightarrow \mathit{On}_2 := \mathit{false}\ !\ \circledast(w_2 \geq T) \\
[\!]\ \ e_2: & \neg \mathit{On}_2 \wedge \mathit{Pressed} \wedge w_2 > \mathit{Reserve}.T_2 & \rightarrow \mathit{On}_2 := \mathit{true} \\
[\!]\ \ e_3: & \neg \mathit{grd}.e_1 \wedge \neg \mathit{grd}.e_2 & \rightarrow \mathbf{idle} \\
\mathbf{od}\ \dagger \tau_2\ ?((31) \wedge \mathit{DA}.w_2) & &
\end{array}$$

Fig. 10. Introduce actions

We introduce actions e_1, e_2 and e_3 to the program using Lemma 32, which gives us the program in Fig. 10. Note that we do not remove e_0 at this stage to enable future modifications to the actions to be made more easily.

6.3 Discharge Enforced Properties on Actions

We now make modifications to remove the enforced property within e_1 in Fig. 10. We refrain from modifying the guards to avoid reproving conditions that have already been established. Instead, noting that $beh_V.e_1 \Rrightarrow prev.\overrightarrow{On_2}$ (by (13) of Lemma 17) and that $prev.(\overrightarrow{w_2 \geq T + \mathsf{acc}.w_2.\tau_2}) \wedge \ell \leq \rho.\tau_2 \Rrightarrow \circledast(w_2 \geq T)$ (by Corollary 9), $\circledast(w_2 \geq T)$ within e_1 holds if the program satisfies

$$\Box(\overrightarrow{On_2} \Rightarrow \odot(w_2 \geq T + \mathsf{acc}.w_2.\tau_2)) \tag{42}$$

Hence, using Lemma 33, we introduce (42) as an enforced property to the program. Condition (42) and Lemma 27 allows us to remove the enforced property $\circledast(w_2 \geq T)$ within e_1, however, we must now consider the changes necessary ensure it holds.

We use Theorem 21, which allows (42) to be proved by showing that for each $i \in \{1, 2, 3\}$, if On_2 holds at the end of the interval corresponding the execution of e_i, then $w_2 \geq T + \mathsf{acc}.w_2.\tau_2$ must also hold. Action e_1 is trivial because $beh_V.e_1 \Rrightarrow \overrightarrow{\neg On_2}$ holds. For e_2, we let

$$Reserve.T_2 \geq T + 2\mathsf{acc}.w_2.\tau_2 \tag{43}$$

which by Corollary 9 ensures $beh_V.e_2 \Rrightarrow \odot(w_2 \geq T + \mathsf{acc}.w_2.\tau_2)$. For e_3, we have:

$$\begin{array}{ll}
 & (38) \vee (39) \Rrightarrow \overrightarrow{\neg On_2} \vee \odot(w_2 \geq T + \mathsf{acc}.w_2.\tau_2) \\
\Leftarrow & \quad (39) \Rrightarrow \overrightarrow{\neg On_2}, \text{ strengthen consequent} \\
 & (38) \Rrightarrow \odot(w_2 \geq T + \mathsf{acc}.w_2.\tau_2) \\
\Leftarrow & \quad \text{Corollary 9} \\
 & (43)
\end{array}$$

Because $grd.e_1 \vee grd.e_2 \vee grd.e_3$ holds, we may use Lemma 31 to strengthen the guard of e_0 to *false* without strengthening the guard of the action system. Then using Lemma 32, we may remove action e_0 from the program. Thus, we obtain the final controller in Fig. 11.

$$\begin{array}{ll}
\textbf{do} & e_1\colon On_2 \wedge (\neg Pressed \vee w_2 \leq Reserve.T_2) \rightarrow On_2 := false \\
[] & e_2\colon \quad \neg On_2 \wedge Pressed \wedge w_2 > Reserve.T_2 \rightarrow On_2 := true \\
\textbf{else} & e_3\colon \textbf{idle} \\
\textbf{od} & \dagger\tau_2\,?((31) \wedge DA.w_2)
\end{array}$$

Fig. 11. Final controller

The program in Fig. 11, behaves in time band τ_2 and the environment operates as specified by (31) and $DA.w_2$. Property (31) ensures that $Stopped_2$ holds within an interval of length $\rho.\phi_2$ in which $\circledast\neg On_2$ holds and $DA.w_2$ ensures that that maximum difference of w_2 in any time bands τ_2 and ϕ_2 are within the accuracies of w_2 in τ_2 and ϕ_2, respectively. Combining (30), (37) and (43), we derive a necessary relationship:

$$Reserve.T_2 \geq Empty.T_2 + max(\mathsf{acc}.w_2.\tau_2, \mathsf{acc}.w_2.\phi_2) + 2\mathsf{acc}.w_2.\tau_2$$

on the values of $Reserve.T_2$ and $Empty.T_2$.

7 Conclusions

This paper incorporates a time bands theory [9] into action systems and we develop an interval-based semantics for reasoning about sampling over continuous environments. We use ILTL [17], a temporal logic for sequences of adjoining intervals, and develop a refinement theory using enforced properties specified by ILTL formulae. We have developed high-level methods that use time bands to simplify reasoning about hardware/software interaction. As an example, we have derived an action systems controller for a real-time pump. Notable in our derivation is the development of side conditions that formalise the assumptions on the environment and the derivation of relationships between threshold and critical levels based on the (different) time bands of the controller and pump.

As part of future work, we aim to further develop the theories for parallel composition of action systems by developing (compositional) rely/guarantee-style methods. We also aim to explore the links between action systems and teleo-reactive programs [15, 16]. In particular it will be interesting to consider a development method that starts with a teleo-reactive program (whose semantics are closer to abstract specifications) and refining the teleo-reactive program to an action system.

Acknowledgements. This research is supported by Australian Research Council Discovery Grant DP0987452 and EPSRC Grant EP/J003727/1. We thank our anonymous reviewers for their insightful comments that have helped improve this paper.

References

1. Aichernig, B.K., Brandl, H., Krenn, W.: Qualitative Action Systems. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 206–225. Springer, Heidelberg (2009)
2. Back, R.-J., Petre, L., Porres, I.: Generalizing Action Systems to Hybrid Systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 202–213. Springer, Heidelberg (2000)
3. Back, R.-J.R., Sere, K.: Stepwise refinement of action systems. Structured Programming 12(1), 17–30 (1991)
4. Back, R.-J.R., von Wright, J.: Trace Refinement of Action Systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
5. Back, R.-J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag New York, Inc., Secaucus (1998)
6. Back, R.-J.R., von Wright, J.: Compositional action system refinement. Formal Asp. Comput. 15(2-3), 103–117 (2003)
7. Broy, M.: Refinement of time. Theor. Comput. Sci. 253(1), 3–26 (2001)
8. Burns, A., Baxter, G.: Time bands in systems structure. In: Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, pp. 74–88. Springer (2006)
9. Burns, A., Hayes, I.J.: A timeband framework for modelling real-time systems. Real-Time Systems 45(1), 106–142 (2010)
10. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley Longman Publishing Co., Inc. (1988)
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
12. Dongol, B.: Progress-based verification and derivation of concurrent programs. PhD thesis, The University of Queensland (2009)
13. Dongol, B., Hayes, I.J.: Enforcing safety and progress properties: An approach to concurrent program derivation. In: 20th ASWEC, pp. 3–12. IEEE Computer Society (2009)
14. Dongol, B., Hayes, I.J.: Compositional Action System Derivation Using Enforced Properties. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 119–139. Springer, Heidelberg (2010)
15. Dongol, B., Hayes, I.J.: Reasoning about teleo-reactive programs under parallel composition. Technical Report SSE-2011-01, The University of Queensland (April 2011)
16. Dongol, B., Hayes, I.J.: Approximating idealised real-time specifications using time bands. In: AVoCS 2011. ECEASST, vol. 46, pp. 1–16. EASST (2012)
17. Dongol, B., Hayes, I.J.: Deriving real-time action systems in a sampling logic. Sci. Comput. Program. (2012); accepted October 17, 2011
18. Dongol, B., Mooij, A.J.: Streamlining progress-based derivations of concurrent programs. Formal Aspects of Computing 20(2), 141–160 (2008)
19. Feijen, W.H.J., van Gasteren, A.J.M.: On a Method of Multiprogramming. Springer (1999)
20. Gargantini, A., Morzenti, A.: Automated deductive requirements analysis of critical systems. ACM Trans. Softw. Eng. Methodol. 10, 255–307 (2001)
21. Guelev, D.P., Hung, D.V.: Prefix and projection onto state in duration calculus. Electr. Notes Theor. Comput. Sci. 65(6), 101–119 (2002)

22. Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust Timed Automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)
23. Hayes, I.J., Burns, A., Dongol, B., Jones, C.B.: Comparing models of nondeterministic expression evaluation. Technical Report CS-TR-1273, Newcastle University (2011)
24. Henzinger, T.A.: The theory of hybrid automata. In: LICS 1996, pp. 278–292. IEEE Computer Society, Washington, DC (1996)
25. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Assume-Guarantee Refinement Between Different Time Scales. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 208–221. Springer, Heidelberg (1999)
26. Manna, Z., Pnueli, A.: Temporal Verification of Reactive and Concurrent Systems: Specification. Springer-Verlag New York, Inc. (1992)
27. Meinicke, L.A., Hayes, I.J.: Continuous Action System Refinement. In: Yu, H.-J. (ed.) MPC 2006. LNCS, vol. 4014, pp. 316–337. Springer, Heidelberg (2006)
28. Moszkowski, B.C.: Compositional reasoning about projected and infinite time. In: ICECCS, pp. 238–245. IEEE Computer Society (1995)
29. Moszkowski, B.C.: A complete axiomatization of interval temporal logic with infinite time. In: LICS, pp. 241–252 (2000)
30. Rönkkö, M., Ravn, A.P., Sere, K.: Hybrid action systems. Theoretical Computer Science 290(1), 937–973 (2003)
31. Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Form. Methods Syst. Des. 33, 45–84 (2008)
32. Zhou, C., Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. EATCS: Monographs in Theoretical Computer Science. Springer (2004)

Calculating Graph Algorithms for Dominance and Shortest Path*

Ilya Sergey[1], Jan Midtgaard[2], and Dave Clarke[1]

[1] KU Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be
[2] Aarhus University, Denmark
jmi@cs.au.dk

Abstract. We calculate two iterative, polynomial-time graph algorithms from the literature: a dominance algorithm and an algorithm for the single-source shortest path problem. Both algorithms are calculated directly from the definition of the properties by *fixed-point fusion* of (1) a least fixed point expressing all finite paths through a directed graph and (2) Galois connections that capture dominance and path length.

The approach illustrates that reasoning in the style of fixed-point calculus extends gracefully to the domain of graph algorithms. We thereby bridge common practice from the school of program calculation with common practice from the school of static program analysis, and build a novel view on iterative graph algorithms as instances of abstract interpretation.

Keywords: graph algorithms, dominance, shortest path algorithm, fixed-point fusion, fixed-point calculus, Galois connections.

1 Introduction

Calculating an *implementation* from a *specification* is central to two active sub-fields of theoretical computer science, namely the calculational approach to program development [1, 12, 13] and the calculational approach to abstract interpretation [18, 22, 33, 34]. The advantage of both approaches is clear: the resulting implementations are provably correct by construction. Whereas the former is a general approach to program development, the latter approach is mainly used for developing provably sound static analyses (with notable exceptions [19, 25]). Both approaches are anchored in some of the same discrete mathematical structures, namely partial orders, complete lattices, fixed points and Galois connections.

Graphs and graph algorithms are foundational to computer science as they capture the essence of networks, compilers, social connections, and much more. One well-known class of graph algorithms is the shortest path algorithms, exemplified by Dijkstra's single-source shortest path algorithm [28, 16]. Dominance

* This work was carried out while the first author was visiting Aarhus University in the fall of 2011.

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 132–156, 2012.
© Springer-Verlag Berlin Heidelberg 2012

algorithms are another class of widespread use: for transforming programs into static single assignment form [5], for optimizing functional programs [30], for ownership typing [35], for information flow analysis [10], etc. In this paper we reconsider the calculational foundations for such algorithms in the context of fixed-point calculus and Galois connections.

In order to bridge two worlds, namely, calculational program development and semantics-based program analysis, we employ the toolset of both fixed-point calculus [1] and abstract interpretation [22], and show that solutions for finite path properties in graphs can be obtained by these means, yielding polynomial-time algorithms that are correct by construction. In doing so, we utilize Galois connections to extract properties (namely dominance and shortest path) from sets of paths in graphs similar to how Galois connections are used to extract properties from program executions in abstract interpretation.

2 Background

We first highlight the relevant mathematical preliminaries. Readers familiar with lattices and orders [27] as found in the fixed-point calculus [1], basic abstract interpretation [26], and relational algebra [7] may wish to proceed directly to Section 3.

2.1 Notation

We use the standard notation $\wp(X)$ for the powerset of X. When working with sets and relations, we will make use of Eindhoven notation for quantified expressions [40]. The general pattern is $\langle Q\ x : p(x) : t(x)\rangle$, where Q is some quantifier (e.g., "$\forall$" or "$\exists$"), x is a sequence of free variables (also called *dummies*), $p(x)$ is a predicate, which must be satisfied by the dummies, and $t(x)$ is an expression, defined in terms of the dummies. For instance, for cases "$\forall$" or "$\exists$", we have the following relations with the standard notation:

$$\langle \forall x : p(x) : q(x)\rangle \iff \forall x.(p(x) \Rightarrow q(x))$$
$$\langle \exists x : p(x) : q(x)\rangle \iff \exists x.(p(x) \wedge q(x))$$

Following the same notation, we use set comprehensions $\{x : p(x) : q(x)\}$ as a shorthand for $\langle \cup x : p(x) : \{q(x)\}\rangle$, where x contains components to union over, p is a filtering condition and $\{q(x)\}$ is a yielded result for a particular combination from x.[1] The square brackets around a formula indicate a universal quantification over any free variables, not mentioned in the preamble. For instance, $[\ x \vee \neg x\ ]$.

In the proofs, we will overload the equality sign "=" to mean equivalence between two subsequent steps of a derivation, supplying a textual explanation in fancy brackets: $\wr\ \ldots\ \wr$.

[1] This notation is equivalent to the traditional notation $\{q(x) \mid p(x)\}$, which does not make explicit which variables are bound and which variables are free.

2.2 Basics of Domain Theory and Abstract Interpretation

A *complete lattice* $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a partial order $\langle C; \sqsubseteq \rangle$ such that there exists a least upper bound (or join) $\sqcup S$ and a greatest lower bound (or meet) $\sqcap S$ of all subsets $S \subseteq C$. In particular $\sqcup C = \top$ and $\sqcap C = \bot$.

A point x is a fixed point of a function f if $f(x) = x$. Given two partial orders, $\langle C, \sqsubseteq \rangle$ and $\langle A, \leq \rangle$, a function f of type $C \to A$ is monotone if $[\ x \sqsubseteq y \implies f(x) \leq f(y)\ ]$. By the Knaster-Tarski fixed-point theorem a monotone endo-function f over a complete lattice has a *least fixed point* $\mathsf{lfp}_{\sqsubseteq} f = \sqcap \{x \mid f(x) \sqsubseteq x\}$. Algorithmically the least fixed point of a monotone function f over a complete lattice of finite *height*[2] can be computed by Kleene iteration: $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq f^3(\bot) \sqsubseteq \ldots$ since $\mathsf{lfp}_{\sqsubseteq} f = \sqcup_{i \geq 0} f^i(\bot)$.

A *Galois connection* is a pair of functions α, γ (in the present case, between two partial orders $\langle C, \sqsubseteq \rangle$ and $\langle A, \leq \rangle$), such that $[\ \alpha(c) \leq a \iff c \sqsubseteq \gamma(a)\ ]$. The function α is referred to as the *lower adjoint* and the function γ is referred to as the *upper adjoint*.[3] We typeset Galois connections as:

$$\langle C, \sqsubseteq \rangle \xleftarrow[\alpha]{\gamma}\!\!\!\!\!\!\!\xrightarrow{} \langle A, \leq \rangle$$

sometimes with double arrow heads to stress that an adjoint is surjective. Galois connections enjoy a number of properties of which we only highlight a few. Both adjoints of a Galois connection are monotone. Furthermore, for a Galois connection between two complete lattices the lower adjoint distributes over the least upper bound: $[\ \alpha(\sqcup \mathcal{X}) = \bigvee \alpha(\mathcal{X})\ ]$. Finally if a function between two complete lattices distributes over the least upper bound, then it is the lower adjoint of a Galois connection with its corresponding upper adjoint uniquely determined by $\gamma(a) = \sqcup \{c \mid \alpha(c) \leq a\}$ [23].

Galois connections can be constructed compositionally: given $\langle C, \sqsubseteq \rangle \xleftarrow[\alpha_1]{\gamma_1}\!\!\!\!\!\!\!\xrightarrow{} \langle A_1, \leq_1 \rangle$ and $\langle A_1, \leq_1 \rangle \xleftarrow[\alpha_2]{\gamma_2}\!\!\!\!\!\!\!\xrightarrow{} \langle A_2, \leq_2 \rangle$, one has $\langle C, \sqsubseteq \rangle \xleftarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2}\!\!\!\!\!\!\!\xrightarrow{} \langle A_2, \leq_2 \rangle$.

Galois connections interact with least fixed points by *fixed-point fusion* [1]:

$$\alpha \circ F_c \mathrel{\dot{\leq}} F_a \circ \alpha \implies \alpha(\mathsf{lfp}\ F_c) \leq \mathsf{lfp}\ F_a \tag{1}$$

$$\alpha \circ F_c = F_a \circ \alpha \implies \alpha(\mathsf{lfp}\ F_c) = \mathsf{lfp}\ F_a \tag{2}$$

for monotone functions F_c and F_a (where we have written $f \mathrel{\dot{\leq}} g$ for the pointwise ordering $[\ f(x) \leq g(x)\ ]$).[4] Note that we overload the notation for a least fixed point lfp, using it for different domains and orders, without specifying them explicitly, when it is obvious from the context. For instance, in the definitions (1) and (2) we use lfp in both cases, assuming in fact $\mathsf{lfp}_{\sqsubseteq}$ for F_c and $\mathsf{lfp}_{\leq}$ for F_a, respectively.

---

[2] Traditionally, the *height* of a lattice $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ denotes the maximal length of a (possibly infinite) strictly increasing chain of elements $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_i \in C$.

[3] In the abstract interpretation literature where they are typically associated with some information loss they are known as the *abstraction* and *concretization* functions, respectively [23].

[4] The first implication is also referred to as the *fixed-point transfer theorem* [22] and the latter implication is known as a *complete abstraction* [23].

2.3 Elements of Relational Algebra

Composition is a well-known operation on relations. For given relations $R \subseteq A \times B$ and $S \subseteq B \times C$, their composition $R \circ S \subseteq A \times C$ is defined as follows:

$$R \circ S \equiv \{x, y, z : \langle x, y\rangle \in R \wedge \langle y, z\rangle \in S : \langle x, z\rangle\}. \tag{3}$$

A particular case of relation composition is function composition. In order for function composition to be consistent with the right-to-left $\circ$-notation, we also think of a function of type $A \to B$ as a relation over $B \times A$ [12].

Another important notion we are going to use is *factors* [7]. Given three relations $R \subseteq A \times B$, $S \subseteq B \times C$ and $T \subseteq A \times C$, the *left factor* $T/S \subseteq A \times B$ and the *right factor* $R\backslash T \subseteq B \times C$ are defined pointwise as follows:

$$[\ x\ T/S\ y \equiv \langle \forall z : y\ S\ z : x\ T\ z\rangle\ ] \tag{4}$$

$$[\ x\ R\backslash T\ y \equiv \langle \forall z : z\ R\ x : z\ T\ y\rangle\ ] \tag{5}$$

Both the notions of composition and factors are helpful for reasoning in *point-free style*: while composition eliminates existential quantifications, the factor operations eliminate universal quantification. It is also notable that

$$[\ R \circ S \subseteq T \iff S \subseteq R\backslash T\ ] \tag{6}$$

$$[\ R \circ S \subseteq T \iff R \subseteq T/S\ ] \tag{7}$$

so we have

$$[\ T/S \supseteq R \iff S \subseteq R\backslash T\ ] \tag{8}$$

which makes it possible to consider the eta-expanded factors $(T/) = \lambda\mathcal{X}.T/\mathcal{X}$ and $(\backslash T) = \lambda\mathcal{X}.\mathcal{X}\backslash T$ as the adjoints of a Galois connection:

$$\langle \wp(B \times C), \subseteq\rangle \underset{(T/)}{\overset{(\backslash T)}{\rightleftarrows}} \langle \wp(A \times B), \supseteq\rangle \tag{9}$$

3 Calculating a Dominance Algorithm

In this section we derive an algorithm to compute a dominance relation of a directed graph. We first express dominance as a lower adjoint over a set of finite paths. We then calculate the dominance computation algorithm using fixed-point fusion with a least fixed point expressing all finite paths through a graph.

3.1 Graphs and Finite Paths

Definition 1 (Directed Graph). *A* rooted *directed graph is a triple $G = \langle V, E, v_0\rangle$, where V is a set of nodes, $E \subseteq V \times V$ is a set of edges and $v_0 \in V$ is a designated* initial *node.*

We use the notation $(u \to v)$ to indicate the edge $\langle u, v\rangle \in E$. A non-empty ***path*** $\sigma \in V^+$ in a graph G is a sequence of nodes $\sigma = u_0 \dots u_n$, such that for all $i \in 1 \dots n$, $(u_{i-1} \to u_i)$. Given a graph $G = \langle V, E, v_0\rangle$, all finite paths starting from v_0 can be obtained by "walking through the set of edges", which leads to the following definition:

Definition 2 (Finite path functional). [5] *Given a fixed graph $G = \langle V, E, v_0\rangle$, a finite path functional $p_G : \wp(V^+) \to \wp(V^+)$ is defined as follows:*

$$p_G(\mathcal{X}) = \{\sigma, v : \sigma \in \mathcal{X} \wedge (\texttt{last}(\sigma) \to v) : \sigma v\}, \tag{10}$$

where the function `last` *of type $V^+ \to V$ on non-empty paths is defined by*

$$\texttt{last}(\sigma u) = u. \tag{11}$$

In some cases, we will also consider `last` as a relation (i.e., $\texttt{last} \subseteq V \times V^+$) in order to compose it with other relations.

Using the well-known observation [7,26] that $\langle \wp(V^+), \subseteq\rangle$ is a complete lattice with $\sqcup = \cup$, $\sqcap = \cap$, $\bot = \emptyset$ and $\top = V^+$, and the fact that p_G is monotone, one can express the set of finite paths through a graph G, starting in v_0 as the following *least fixed point*:

$$\mathtt{P}_G = \mathsf{lfp}(\lambda \mathcal{X}.\{v_0\} \cup p_G(\mathcal{X})) \tag{12}$$

By a simple inductive argument any finite path through G starting in v_0 belongs to $\mathtt{P}_G$. In the remainder of this section we consider a fixed graph $G = \langle V, E, v_0\rangle$.

3.2 Dominance in Finite Paths

A classical definition of dominance in a graph is stated as follows [36]:

> *A node u **dominates** node v if u belongs to every path from the initial node v_0 of the graph to v.*

Our goal is to derive an algorithm for computing dominators directly from the definition above. Clearly, the set of all finite paths cannot be examined in general, since it is infinite in the presence of cycles in the graph. Nevertheless, we start from the definition of dominance in a set of paths.

Definition 3. *The function* `dom` *of type $\wp(V^+) \to \wp(V \times V)$ is defined for all $\mathcal{X} \subseteq V^+$ as follows:*

$$[\ u\ \texttt{dom}(\mathcal{X})\ v = \langle \forall \sigma : \sigma \in \mathcal{X} \wedge \texttt{last}(\sigma) = v : u\ \texttt{in}\ \sigma\rangle\ ], \tag{13}$$

[5] The same functional is traditionally used within the *partial trace collecting semantics* [26].

where the relation $\mathtt{in} \subseteq V \times V^+$ *is defined by:*

$$\mathtt{in} = \mathsf{lfp}(\lambda \mathcal{X}.\mathtt{last} \cup \mathcal{X} \circ \mathtt{pre}) \tag{14}$$

and $\mathtt{pre} \subseteq V^+ \times V^+$ *is* $\{\sigma, v : \sigma \in V^+ \wedge \sigma v \in V^+ : \langle \sigma, \sigma v \rangle\}$

In words, Definition 3 says that u antecedes v for all paths in $\mathcal{X}$ trailed by v. One can notice that if there are no paths in $\mathcal{X}$ that end with v, then all nodes u dominate v. Also, a node u dominates itself for any $\mathcal{X}$.

3.3 A Galois Connection between Sets of Finite Paths and Dominance Relations

We proceed by building a Galois connection between the lattice of finite paths through a graph and a lattice of relations on the nodes, using the dominance function dom from Definition 3 as a lower adjoint:

$$\langle \wp(V^+), \subseteq \rangle \underset{\mathtt{dom}}{\overset{\overline{\mathtt{dom}}}{\leftrightarrows}} \langle \wp(V \times V), \supseteq \rangle \tag{15}$$

In order to do so, we first reformulate dominance in point-free style using factors (see Section 2). The new equivalent definition is established by the following lemma:

Lemma 1

$$\mathtt{dom} = (\mathtt{in}/) \circ f \tag{16}$$

where

$$f(\mathcal{X}) = \{\sigma : \sigma \in \mathcal{X} : \langle \mathtt{last}(\sigma), \sigma \rangle\} \tag{17}$$

If we consider last as a relation, we can construct its powerset, $\wp(\mathtt{last})$. If we furthermore view the latter as a type, f's signature is $\wp(V^+) \to \wp(\mathtt{last})$.

Proof. For all $u, v \in V$

$$
\begin{aligned}
& u \ \mathtt{dom}(\mathcal{X}) \ v \\
= & \ \wr \text{ by definition (13) } \wr \\
& \langle \forall \sigma : \sigma \in \mathcal{X} \wedge \mathtt{last}(\sigma) = v : u \ \mathtt{in} \ \sigma \rangle \\
= & \ \wr \text{ by definition of } f \text{ (17), definition of } / \text{ (4) } \wr \\
& u \ \mathtt{in}/f(\mathcal{X}) \ v
\end{aligned}
$$

□

Second, we show that dom is indeed a lower adjoint of the desired Galois connection. Since dom is equivalent to a composition of in/ and f and we already know that in/ is a lower adjoint from $\langle \wp(\mathtt{last}), \subseteq \rangle$ to $\langle \wp(V \times V), \supseteq \rangle$ (see Section 2.3), we only need to show that f is a lower adjoint from $\langle \wp(V^+), \subseteq \rangle$ to $\langle \wp(\mathtt{last}), \subseteq \rangle$. The following lemma delivers this result.

Lemma 2. *f is a lower adjoint in a Galois connection*

$$\langle \wp(V^+), \subseteq \rangle \underset{f}{\overset{\overline{f}}{\leftrightarrows}} \langle \wp(\mathtt{last}), \subseteq \rangle.$$

Proof. Suppose, $\mathcal{X} \subseteq V^+$ and $T \subseteq \mathtt{last}$ (i.e., $[\ u\ T\ \sigma \Rightarrow u = \mathtt{last}(\sigma)\ ]$). Then

$$\begin{aligned}
&f(\mathcal{X}) \subseteq T \\
= &\wr \text{ by definition (17) } \wr \\
&\langle \forall \sigma : \sigma \in \mathcal{X} : \mathtt{last}(\sigma)\ T\ \sigma \rangle \\
= &\wr \text{ by definition of } \subseteq \wr \\
&\mathcal{X} \subseteq \{\sigma : \mathtt{last}(\sigma)\ T\ \sigma : \sigma\} \\
= &\wr\ T \subseteq \mathtt{last}, \text{ taking } T' = \{v, \sigma : v\ T\ \sigma : \sigma\} \wr \\
&\mathcal{X} \subseteq T'
\end{aligned}$$

Therefore, f is a lower adjoint with upper adjoint the function that maps a relation T which is a subset of `last` to its right component:

$$\overline{f}(T) = \{v, \sigma : v\ T\ \sigma : \sigma\}$$

□

The following corollary states the fixed-point fusion property (2) with respect to `dom`:

Corollary 1. *The function* `dom` *is a lower adjoint in a Galois connections between sets of paths ordered by the $\subseteq$ relation and subsets of $V \times V$ ordered by the $\supseteq$ relation. Furthermore, if h is a monotone function of type $\langle \wp(V^+), \subseteq \rangle \rightarrow \langle \wp(V^+), \subseteq \rangle$ and g is a monotone function of type $\langle \wp(V \times V), \supseteq \rangle \rightarrow \langle \wp(V \times V), \supseteq \rangle$, then*

$$\mathtt{dom} \circ h = g \circ \mathtt{dom} \Rightarrow \mathtt{dom}(\mathsf{lfp}_{\subseteq}\ h) = \mathsf{lfp}_{\supseteq}\ g,$$

where the least fixed points are computed with respect to the appropriate order relations: $\subseteq$ for h and $\supseteq$ for g.

Proof. Follows from the fact that $\mathtt{dom} = (\mathtt{in}/) \circ f$ (Lemma 1), the composition property of Galois connections applied to (2) and Lemma 2. □

3.4 A Dominance Computation Functional

Having a Galois connection between the lattice of sets of finite paths and the lattice of dominance relations enable us to derive a functional for computing the dominance relation, induced by the set of *all* paths, which we defined as $\mathtt{P}_G$. In this section, we extract an algorithm to compute the actual dominance relation corresponding to all finite paths in the graph.

We construct the ***dominance computation functional*** $\mathcal{F}_{\mathscr{D}}$ from the finite path functional p_G and the lower adjoint $\mathtt{dom}$, such that:

$$\mathtt{dom} \circ p_G = \mathcal{F}_{\mathscr{D}} \circ \mathtt{dom} \tag{18}$$

Then we can just apply Corollary 1, taking $h = p_G$ and $g = \mathcal{F}_{\mathscr{D}}$.

We derive $\mathcal{F}_{\mathscr{D}}$ by a two-staged derivation. First, we find a function k, such that

$$f \circ p_G = k \circ f \tag{19}$$

Second, we find $\mathcal{F}_{\mathscr{D}}$ such that

$$(\mathtt{in}/) \circ k = \mathcal{F}_{\mathscr{D}} \circ (\mathtt{in}/) \tag{20}$$

One can then see that for $\mathcal{F}_{\mathscr{D}}$ defined in such a way we have:

$$
\begin{aligned}
& \mathtt{dom} \circ p_G \\
= & \wr \text{ by Lemma 1 } \int \\
& (\mathtt{in}/) \circ f \circ p_G \\
= & \wr \text{ by (19) } \int \\
& (\mathtt{in}/) \circ k \circ f \\
= & \wr \text{ by (20) } \int \\
& \mathcal{F}_{\mathscr{D}} \circ (\mathtt{in}/) \circ f \\
= & \wr \text{ by Lemma 1 } \int \\
& \mathcal{F}_{\mathscr{D}} \circ \mathtt{dom}
\end{aligned}
$$

and hence satisfy the requirement from equation (18).

Informally, we obtain the function k from (19) by "pushing" the lower adjoint f under the function definition p_G, a well-known "recipe" within the abstract interpretation community [24]:

$$
\begin{aligned}
& f(p_G(\mathcal{X})) \\
= & \wr \text{ by definition (17) } \int \\
& \{\sigma : \sigma \in p_G(\mathcal{X}) : \langle \mathtt{last}(\sigma), \sigma \rangle\} \\
= & \wr \text{ by definition (10) } \int \\
& \{\sigma, v : \sigma \in \mathcal{X} \wedge (\mathtt{last}(\sigma) \to v) : \langle \mathtt{last}(\sigma v), \sigma v \rangle\} \\
= & \wr \text{ one-point rule, definition of } \mathtt{last} \int \\
& \{\sigma, u, v : \sigma \in \mathcal{X} \wedge \mathtt{last}(\sigma) = u \wedge (u \to v) : \langle v, \sigma v \rangle\} \\
= & \wr \text{ by definition (17) } \int \\
& \{\sigma, u, v : \langle u, \sigma \rangle \in f(\mathcal{X}) \wedge (u \to v) : \langle v, \sigma v \rangle\} \\
= & \wr \text{ taking } k(\mathcal{X}) = \{\sigma, u, v : \langle u, \sigma \rangle \in \mathcal{X} \wedge (u \to v) : \langle v, \sigma v \rangle\} \int \\
& k(f(\mathcal{X}))
\end{aligned}
$$

Hence the following lemma:

Lemma 3

$$f \circ p_G = k \circ f,$$

where $k : \langle \wp(\mathtt{last}), \subseteq \rangle \to \langle \wp(\mathtt{last}), \subseteq \rangle$ *is defined as follows:*

$$k(\mathcal{X}) = \{\sigma, u, v : \langle u, \sigma \rangle \in \mathcal{X} \wedge (u \to v) : \langle v, \sigma v \rangle\} \tag{21}$$

Now, we obtain $\mathcal{F}_{\mathscr{D}}$ using the same technique as in the previous derivation. Assume $R \in \wp(\mathtt{last})$, then for all $u, v \in V$,

$$\begin{aligned}
& u\ (\mathtt{in}/k(R))\ v \\
= & \wr \text{ by definition (4) } \int \\
& \langle \forall \sigma : v\ k(R)\ \sigma : u\ \mathtt{in}\ \sigma \rangle \\
= & \wr \text{ by definition of } k \text{ (21) } \int \\
& \langle \forall \sigma : \langle \exists w : w \to v : w\ R\ \sigma \rangle : u\ \mathtt{in}\ \sigma v \rangle \\
= & \wr \text{ by definition of } \mathtt{in} \text{ (14) } \int \\
& \langle \forall \sigma : \langle \exists w : w \to v : w\ R\ \sigma \rangle : u = v \vee u\ \mathtt{in}\ \sigma \rangle \\
= & \wr \text{ by distributivity and range splitting } \int \\
& u = v \vee \langle \forall w : w \to v : \langle \forall \sigma : w\ R\ \sigma : u\ \mathtt{in}\ \sigma \rangle \rangle \\
= & \wr \text{ taking } [\ v\ \mathtt{pred}\ w \equiv w \to v\ ] \int \\
& u = v \vee u\ ((\mathtt{in}/R)/\mathtt{pred})\ v \\
= & \wr \text{ taking } \mathcal{F}_{\mathscr{D}}(\mathcal{X}) = \mathtt{id} \cup \mathcal{X}/\mathtt{pred} \int \\
& u\ \mathcal{F}_{\mathscr{D}}(\mathtt{in}/R)\ v
\end{aligned}$$

The presented derivation proves the following lemma:

Lemma 4

$$(\mathtt{in}/) \circ k = \mathcal{F}_{\mathscr{D}} \circ (\mathtt{in}/),$$

where $\mathcal{F}_{\mathscr{D}} : \langle \wp(V \times V), \supseteq \rangle \to \langle \wp(V \times V), \supseteq \rangle$ *is defined by*

$$\mathcal{F}_{\mathscr{D}}(\mathcal{X}) = \mathtt{id} \cup \mathcal{X}/\mathtt{pred} \tag{22}$$

with `id` *denoting the identity relation and* `pred` *defined as* $[\ v\ \mathtt{pred}\ u \equiv u \to v\ ]$.

We now have all the ingredients to express $\mathtt{dom}(\mathtt{P}_G)$ in terms of $\mathcal{F}_{\mathscr{D}}$:

Theorem 1

$$\mathtt{dom}(\mathtt{P}_G) = \mathsf{lfp}_{\supseteq}(\lambda \mathcal{X}.\mathtt{dom}(\{v_0\}) \cap (\mathtt{id} \cup \mathcal{X}/\mathtt{pred})) \tag{23}$$

where the least fixed point $\mathsf{lfp}_{\supseteq}$ *is computed with respect to the partial order* $\langle \wp(V \times V), \supseteq \rangle$.

Proof. First, we note that for all $\mathcal{X} \in \wp(V^+)$

$$\begin{aligned}
& \texttt{dom}(\{v_0\} \cup p_G(\mathcal{X})) \\
= & \wr \text{ by corollary 1 and distributivity of adjoints } \wr \\
& \texttt{dom}(\{v_0\}) \cap \texttt{dom}(p_G(\mathcal{X})) \\
= & \wr \text{ by the properties of } \mathcal{F}_{\mathscr{D}} \text{ (18) } \wr \\
& \texttt{dom}(\{v_0\}) \cap \mathcal{F}_{\mathscr{D}}(\texttt{dom}(\mathcal{X}))
\end{aligned}$$

Applying Corollary 1, one can see that

$$\texttt{dom}(\texttt{P}_G) = \mathsf{lfp}_{\supseteq}(\lambda \mathcal{X}.\texttt{dom}(\{v_0\}) \cap \mathcal{F}_{\mathscr{D}}(\mathcal{X})), \tag{24}$$

where the least fixed point $\mathsf{lfp}_{\supseteq}$ is computed with respect to the $\supseteq$ ordering. Finally, unfolding the definition of $\mathcal{F}_{\mathscr{D}}$ (22) concludes the proof of the theorem. □

3.5 Dominance Equations

From a practical point of view, one is usually more interested in computing a representation of the dominance relation as a map Dom, such that $\text{Dom}(v) = \{u : u \texttt{ dom}(\texttt{P}_G)\ v : u\}$. In this section we construct equivalent data-flow equations and iterative algorithms based on this representation, on the definition of the dominance functional $\mathcal{F}_{\mathscr{D}}$ (22), and on the result of Theorem 1. We thereby bridge the computation of dominance as a least fixed point of the path functional and the more traditional approaches [15].

First, we notice that

$$\begin{aligned}
& u \texttt{ dom}(\{v_0\})\ v \\
= & \wr \text{ definition (13) } \wr \\
& \langle \forall \sigma : \sigma \in \{v_0\} \wedge \texttt{last}(\sigma) = v : u \texttt{ in } \sigma \rangle \\
= & \wr \text{ since } \sigma \in \{v_0\} \iff \sigma = v_0 \text{ and } u \texttt{ in } v_0 \iff u = v_0 \wr \\
& v = v_0 \Rightarrow u = v_0
\end{aligned}$$

Therefore, we obtain

$$\begin{aligned}
& u \texttt{ dom}(\texttt{P}_G)\ v_0 \\
= & \wr \text{ definition (12) } \wr \\
& u \texttt{ dom}(\{v_0\} \cup p_G(\texttt{P}_G))\ v_0 \\
= & \wr \text{ since } \texttt{dom} \text{ is distributive } \wr \\
& u \texttt{ dom}(\{v_0\})\ v_0 \wedge u \texttt{ dom}(p_G(\texttt{P}_G))\ v_0 \\
= & \wr \text{ by the observation above, taking } v = v_0,\ [\ u \texttt{ dom}(\texttt{P}_G)\ u\ ] \wr \\
& u = v_0
\end{aligned}$$

So, we have an equivalence

$$[\ u\ \mathtt{dom}(\mathtt{P}_G)\ v_0 \iff u = v_0\ ] \tag{25}$$

Also, for $v \neq v_0$,

$$\begin{aligned}
& u\ \mathtt{dom}(\mathtt{P}_G)\ v \\
= & \wr \text{ by Theorem 1, since } \mathsf{lfp} \text{ is a fixed-point operator } \wr \\
& u\ (\mathtt{dom}(\{v_0\}) \cap (\mathtt{id} \cup \mathtt{dom}(\mathtt{P}_G)/\mathtt{pred}))\ v \\
= & \wr \text{ by assumption } v \neq v_0 \text{, so } u\ \mathtt{dom}(\{v_0\})\ v\ \wr \\
& u\ (\mathtt{id} \cup \mathtt{dom}(\mathtt{P}_G)/\mathtt{pred})\ v \\
= & \wr \text{ by definitions of } / \text{ (4) and } \mathtt{pred}\ \wr \\
& u = v \vee [\ \forall w : w \to v : u\ \mathtt{dom}(\mathtt{P}_G)\ w\ ]
\end{aligned}$$

So, we obtain the second equivalence

$$[\ u\ \mathtt{dom}(\mathtt{P}_G)\ v \iff u = v \vee \langle \forall w : w \to v : u\ \mathtt{dom}(\mathtt{P}_G)\ w \rangle\ ] \tag{26}$$

Taking Dom $= \mathtt{dom}(\mathtt{P}_G)$ not as a relation, but as a *function* of type $V \to \wp(V)$ defined as $[\ u \in \mathrm{Dom}(v) \equiv u\ \mathtt{dom}(\mathtt{P}_G)\ v\ ]$ and the equivalences (25) and (26), we discover the following equivalent data-flow equations for Dom [2]:[6]

$$\begin{aligned}
\mathrm{Dom}(v_0) &= \{v_0\} \\
\mathrm{Dom}(v) &= \bigcap_{w \in \mathtt{pred}(v)} \mathrm{Dom}(w) \cup \{v\}
\end{aligned} \tag{27}$$

The statement of Theorem 1 can also be exploited to obtain a simple iterative algorithm for computing the least fixed point of the functional $\mathcal{F}_{\mathscr{D}}$ using Kleene iteration. Figure 1 presents such an algorithm, writing $\mathtt{dom}(\{v_0\})$ for the map $\lambda v.(v = v_0\ ?\ \{v_0\} : V)$.

Initially, the dominance set for every node is the entire set of nodes, according to the lattice $\langle \wp(V \times V), \supseteq \rangle$ (i.e., $\bot = V \times V$). This dominance set is then being "shrunk", as the algorithm proceeds to consider more paths. In the output of

```
1: for v ∈ V do
2:     Dom[v] ← V
3: Dom' ← dom({v0}) ∩ F_D(Dom)
4: while Dom ≠ Dom' do
5:     Dom ← Dom'
6:     Dom' ← dom({v0}) ∩ F_D(Dom)
```

Fig. 1. A straightforward algorithm for computing dominance

[6] In order to mimic the traditional presentation [2], we consider `pred` as a function of type $V \to \wp(V)$ defined as $[\ w \in \mathtt{pred}(v) \equiv w \to v\ ]$.

```
1: for v ∈ V do
2:    Dom[v] ← V
3: Dom[v0] ← {v0}
4: Changed ← true
5: while Changed do
6:    Changed ← false
7:    for v ∈ V do
8:       newSet ← (⋂_{w∈pred(v)} Dom[w]) ∪ {v}
9:       if newSet ≠ Dom[v] then
10:         Dom[v] ← newSet
11:         Changed ← true
```

Fig. 2. An optimized iterative dominator algorithm [15]

the algorithm every node is dominated by itself. The initial node v_0 in particular is dominated only by itself. All disconnected nodes in the graph are dominated by all nodes.

This algorithm can be optimized further although we make no attempt to calculate our way to these changes. For example, rather than maintaining two dominance maps Dom and Dom' one can make do with a single map. In each iteration one then needs to keep track of stabilization by other means than map comparison, e.g., using a Boolean flag to signal changes to an entry. By unfolding $\mathcal{F}_{\mathscr{D}}$ and making these changes we arrive at the classic algorithm from Figure 2 (see Cooper et al. [15] for more details on the implementation).

3.6 Complexity

The complexity of the derived algorithm is polynomial: the *height* of the lattice of dominance functions is $\mathcal{O}(|V|^2)$, which is an upper bound on the number of iterations. Each iteration of the first algorithm in Figure 1 requires (1) an $\mathcal{O}(|V|^2)$-time equality test between two lattice elements and (2) computing an intersection for each node over all its predecessors in $\mathcal{F}_{\mathscr{D}}$ which takes $\mathcal{O}(|V| \times |E|)$ operations. As a consequence the algorithm takes $\mathcal{O}(|V|^2(|V|^2 + |V| \times |E|)) = \mathcal{O}(|V|^4 + |V|^3 \times |E|)$ time. The optimized algorithm in Figure 2 uses a constant time stabilization test, but still requires computing an intersection over all predecessors for each node. As a result it has $\mathcal{O}(|V|^3 \times |E|)$ worst case time complexity.

The bottleneck of the optimized algorithm is the strategy by which it chooses a node to process in line 7 of Figure 2. By instead iterating through the vertices in *reverse postorder* [3] (i.e., a node is visited before all its successor nodes have been visited), we can avoid a general fixed-point computation. By this strategy we can obtain a $\mathcal{O}(|V| \times |E|)$ time algorithm. By a clever choice of data structures, representing sets using dominator trees, this can be improved to $\mathcal{O}(|V|^2)$ [15].

Even linear time dominance algorithms exist [4], but the O-notation for these hide a non-negligible constant factor. For practical purposes they do not fare as well as a well-engineered iterative algorithm [32]. We refer to Cooper, Harvey, and Kennedy [15] for a historical account of dominator algorithms.

4 Calculating a Shortest Path Algorithm

In this section, we calculate an algorithm solving the single-source shortest path problem for a weighted graph with non-negative edge costs. We augment the definition of directed graphs from Section 3.1 with a function assigning weights to edges. The shortest distance from the source to a target node is then formulated for sets of finite weighted paths and an iterative algorithm is derived by fixed-point fusion. Finally, we modify the property to compute the actual shortest paths and not only the shortest distances.

4.1 Weighted Graphs and Paths

Definition 4 (Weighted rooted graph). *A* weighted *rooted graph* $G_w = \langle V, E, v_0, W\rangle$ *is a rooted directed graph* $\langle V, E, v_0\rangle$ *with a* **weight** *function* $W : E \to \mathbb{N}$.

For nodes $u, v \in V$, we use the notation $(u \xrightarrow{w} v)$ to indicate the edge $\langle u, v\rangle \in E$ and $W(\langle u, v\rangle) = w$. A ***weighted path*** $\tau \in V_w^+$ is a non-empty sequence of interleaving nodes and weights $\tau = u_0 w_1 \ldots u_{n-1} w_n u_n$, starting and ending by a node, such that for all $i \in 1 \ldots n$, $(u_{i-1} \xrightarrow{w_i} u_i)$.

Definition 5 (Weight of a path [16]). *The* ***weight*** *of a weighted path* $\tau = u_0 w_1 \ldots u_{n-1} w_n u_n$ *is the sum of the weights of its constituent edges:*

$$\|\tau\| = \sum_{i=1}^{n} w_i \tag{28}$$

In the remainder of this section we consider a *fixed* weighted graph $G_w = \langle V, E, v_0, W\rangle$.

4.2 The Single-Source Shortest Path Property for Finite Paths

In this section we will focus on the ***single-source shortest-path problem***, which can be defined as follows:

> *Given a node* u_0 *and a set of weighted paths*
> $\mathcal{X} = \{\tau : \tau = u_0 w_1 \ldots u_{n-1} w_n u_n : \tau\}$*, for each* v*, such that*
> $\tau_v = u_0 \ldots v \in \mathcal{X}$*, what is the minimum of* $\|\tau_v\|$*?*

Again, our goal is to compute an iterative algorithm for the defined property directly from its definition. In order to do so, we first define the *shortest-path weight* for a set of paths similarly to the canonical property by Cormen et al. [16].

Definition 6 (Shortest-path weight). *Given a set of weighted paths $\mathcal{X}$, then the* ***shortest-path weight*** *from u to v in $\mathcal{X}$ is*

$$\texttt{dist}(\mathcal{X})(u,v) = \min\{\tau : \tau \in \mathcal{X} \ \wedge \ \tau = u \ldots v : \|\tau\|\}, \ \textit{where}$$

$$\min(\emptyset) = \infty.$$

By overloading notation, the single-source shortest-path weight from v_0 to any other node in $\mathcal{X}$ is defined naturally using the function `last` (11) for weighted paths:

$$\texttt{dist}(\mathcal{X}) = \lambda v. \min\{\tau : \tau \in \mathcal{X} \ \wedge \ \texttt{last}(\tau) = v : \|\tau\|\}. \tag{29}$$

As in the canonical definition [16], we define the shortest-path weights as a function from a set of finite paths to natural numbers extended with infinity. Still, an arbitrary weighted graph can contain a possibly infinite number of paths from a node u to v. We connect the world of weighted graphs with sets of weighted paths by redefining the path functional from Section 2 for the single-source weighted paths of a weighted graph G_w.

Definition 7 (Weighted finite path functional). *Given a weighted graph $G_w = \langle V, E, v_0, W\rangle$, a weighted finite path functional $p_{G_w} : \wp(V_w^+) \to \wp(V_w^+)$ is defined as follows:*

$$p_{G_w}(\mathcal{X}) = \{\tau, w, v : \tau \in \mathcal{X} \wedge (\texttt{last}(\tau) \xrightarrow{w} v) : \tau w v\}. \tag{30}$$

Similarly to Section 3.1, $\langle \wp(V_w^+), \subseteq\rangle$ is a complete lattice, so the set of all weighted single-source finite paths in the graph is defined as the following least fixed point:

$$\mathsf{P}_{G_w} = \mathsf{lfp}(\lambda\mathcal{X}.\{v_0\} \cup p_{G_w}(\mathcal{X})) \tag{31}$$

Again by a simple inductive argument any finite weighted path starting in v_0 belongs to P_{G_w}. In this setting, $\texttt{dist}(\mathsf{P}_{G_w})$ specifies the single-source shortest path property for the whole graph. In the remainder of this section we will derive an algorithm to compute it using fixed-point fusion.

4.3 A Galois Connection between Sets of Finite Paths and the Shortest Path Weights

The function `dist` defined in Section 4.2 maps a set of paths to a function, mapping a node to a non-negative weight or infinity (in case a node is unreachable from v_0), so the codomain of `dist` is $\mathscr{E} = V \to \mathbb{N} \cup \{\infty\}$. In order to make it a complete lattice we extend natural arithmetic to infinity:

$$\forall n \in \mathbb{N} \ : \ n + \infty = \infty + n = \infty + \infty = \infty$$
$$\forall n \in \mathbb{N} \ : \ n < \infty$$
$$\infty \leq \infty$$

Next, we introduce a partial order and the least upper bound on elements δ of $\mathscr{E}$:

$$[\ \delta_1 \dot{\geq} \delta_2 \equiv \forall u \in V\ :\ \delta_1(u) \geq \delta_2(u)\ ] \tag{32}$$

$$[\ \delta_1 \sqcup \delta_2 = \lambda u.\min\{\delta_1(u), \delta_2(u)\}\ ] \tag{33}$$

Finally, one can observe that $\langle \mathscr{E}, \dot{\geq} \rangle$ is a complete lattice with the meet operation provided by (33), $\bot_{\mathscr{E}} = \lambda v.\infty$ and $\top_{\mathscr{E}} = \lambda v.0$. This follows, e.g, from realizing that $\langle \mathbb{N} \cup \{\infty\}, \geq \rangle$ is a complete lattice[7] that can be lifted into a complete lattice over functions with the above pointwise operations.

In order to build the Galois connection between $\langle \wp(V_w^+), \subseteq \rangle$ and $\langle \mathscr{E}, \dot{\geq} \rangle$ using `dist` as a lower adjoint, we need to show that `dist` is distributive with respect to $\sqcup$.

Lemma 5

$$\left[\bigsqcup_i \mathtt{dist}(\mathcal{X}_i) = \mathtt{dist}(\bigcup_i \mathcal{X}_i) \right]$$

Proof Let a sequence $\mathcal{X}_i \in \wp(V_w^+)$ be given

$$\begin{aligned}
& \bigsqcup_i \mathtt{dist}(\mathcal{X}_i) \\
= & \wr \text{ by definition of } \sqcup \int \\
& \lambda u.\min_i(\mathtt{dist}(\mathcal{X}_i)(u)) \\
= & \wr \text{ by definition of } \mathtt{dist} \int \\
& \lambda u.\min_i(\min\{\tau : \tau \in \mathcal{X}_i \ \wedge\ \mathtt{last}(\tau) = u : \|\tau\|\}) \\
= & \wr \text{ min is associative and commutative } \int \\
& \lambda u.\min\{\tau : \tau \in \bigcup_i \mathcal{X}_i \ \wedge\ \mathtt{last}(\tau) = u : \|\tau\|\} \\
= & \wr \text{ by definition of } \mathtt{dist} \int \\
& \mathtt{dist}(\bigcup_i \mathcal{X}_i)
\end{aligned}$$

□

Recall from Section 2.2 that Lemma 5 guarantees the existence of a Galois connection between the two complete lattices, including a unique upper adjoint $\overline{\mathtt{dist}}$:

$$\langle \wp(V_w^+), \subseteq \rangle \underset{\mathtt{dist}}{\overset{\overline{\mathtt{dist}}}{\leftrightarrows}} \langle \mathscr{E}, \dot{\geq} \rangle$$

[7] The construction corresponds roughly to half an *interval domain* formalized by Cousot and Cousot as a complete product lattice $(\{-\infty\} \cup \mathbb{Z}) \times (\mathbb{Z} \cup \{\infty\})$ [20].

4.4 A Shortest-Path Functional

In this section, we extract an algorithm to compute the shortest-path weight function corresponding to all finite paths in the graph. In order to do so, first, we derive the ***shortest-path functional*** $\mathcal{F}_\delta$ by the "pushing" the lower adjoint `dist` under p_{G_w}:

$$
\begin{aligned}
& \texttt{dist}(p_{G_w}(\mathcal{X})) \\
= & \wr \text{ by the definition of } p_{G_w} \text{ (30) } \wr \\
& \texttt{dist}(\{\tau, w, v : \tau \in \mathcal{X} \wedge (\texttt{last}(\tau) \xrightarrow{w} v) : \tau w v\}) \\
= & \wr \text{ by definition of } \texttt{dist} \text{ (29) } \wr \\
& \lambda v.\min\{\tau, w : \tau \in \mathcal{X} \wedge (\texttt{last}(\tau) \xrightarrow{w} v) : \|\tau w v\|\} \\
= & \wr \text{ by definition of } \|\tau w v\| \text{ (28) } \wr \\
& \lambda v.\min\{\tau, w : \tau \in \mathcal{X} \wedge (\texttt{last}(\tau) \xrightarrow{w} v) : \|\tau\| + w\} \\
= & \wr \text{ taking } u = \texttt{last}(\tau) \wr \\
& \lambda v.\min\{\tau, u, w : \tau \in \mathcal{X} \wedge (u \xrightarrow{w} v) \wedge \texttt{last}(\tau) = u : \|\tau\| + w\} \\
= & \wr \text{ by the property of min } \wr \\
& \lambda v.\min\{u, w : (u \xrightarrow{w} v) : \overbrace{\min\{\tau : \tau \in \mathcal{X} \wedge \texttt{last}(\tau) = u : \|\tau\|\}}^{\texttt{dist}(\mathcal{X})(u)} + w\} \\
= & \wr \text{ by folding definition of } \texttt{dist} \text{ (29) } \wr \\
& \lambda v.\min\{u, w : (u \xrightarrow{w} v) : \texttt{dist}(\mathcal{X})(u) + w\} \\
= & \wr \text{ taking } \texttt{pred}(v) = \{u : u \xrightarrow{w} v : u\} \text{ and } W(\langle u, v\rangle) = w \wr \\
& \lambda v.\min\{u : u \in \texttt{pred}(v) : \texttt{dist}(\mathcal{X})(u) + W(\langle u, v\rangle)\} \\
= & \wr \text{ defining } \mathcal{F}_\delta(\mathcal{Y}) = \lambda v.\min\{u : u \in \texttt{pred}(v) : \mathcal{Y}(u) + W(\langle u, v\rangle)\} \wr \\
& \mathcal{F}_\delta(\texttt{dist}(\mathcal{X}))
\end{aligned}
$$

The derivation above proves the following lemma:

Lemma 6

$$\texttt{dist} \circ p_{G_w} = \mathcal{F}_\delta \circ \texttt{dist}$$

where $\mathcal{F}_\delta$ *is of type* $\langle \mathscr{E}, \dot{\geq} \rangle \rightarrow \langle \mathscr{E}, \dot{\geq} \rangle$ *is defined for all* $\mathcal{X}$ *by*

$$\mathcal{F}_\delta(\mathcal{X}) = \lambda v.\min\{u : u \in \texttt{pred}(v) : \mathcal{X}(u) + W(\langle u, v\rangle)\} \tag{34}$$

We can now notice that $\texttt{dist}(\{v_0\}) = \lambda v.(v = v_0 \ ? \ 0 : \infty)$, so the following theorem follows naturally:

Theorem 2

$$\texttt{dist}(\mathsf{P}_{G_w}) = \mathsf{lfp}_{\dot{\geq}}\ (\lambda \mathcal{X}.(\lambda v.(v = v_0 \ ? \ 0 : \infty)) \sqcup \mathcal{F}_\delta(\mathcal{X})) \tag{35}$$

where the least fixed point $\mathsf{lfp}_{\dot{\geq}}$ *is computed with respect to the ordering* $\dot{\geq}$ *over* $\mathscr{E}$*, starting from* $\perp_{\mathscr{E}} = \lambda v.\infty$*.*

Proof. Similarly to the proof of Theorem 1, using distributivity of `dist`, Lemma 6, fixed-point fusion (2) and inlining $\texttt{dist}(\{v_0\})$. □

```
1: for v ∈ V do
2:    δ(v) ← ∞
3: δ′ ← dist({v0}) ⊔ F_δ(δ)
4: while δ′ ≠ δ do
5:    δ ← δ′
6:    δ′ ← dist({v0}) ⊔ F_δ(δ)
```

Fig. 3. A straightforward algorithm for single-source shortest paths

Figure 3 provides a first iterative algorithm for computing the least fixed point of the functional $\mathcal{F}_\delta$ using Kleene iteration. Again the algorithm has room for improvement.

```
 1: for u ∈ V do
 2:    δ[u] ← ∞
 3: δ[v0] ← 0
 4: Changed ← true
 5: while Changed do
 6:    Changed ← false
 7:    for v ∈ V do
 8:       for u ∈ pred(v) do
 9:          if δ[u] + W[u, v] < δ[v] then
10:             δ[v] ← δ[u] + W[u, v]
11:             Changed ← true
```

Fig. 4. An optimized imperative single-source shortest path algorithm

By unfolding $\mathcal{F}_\delta$ and maintaining only a single δ-map as in Section 3.5 we arrive at the single-source shortest-path algorithm in Figure 4. The resulting algorithm is strikingly similar to Bellman's iterative algorithm [11] for computing shortest paths: as Bellman's algorithm proceeds by computing a "*monotone sequence*" of "*successive approximations*" so does the derived algorithm. The algorithms differ in that Bellman assumes that all nodes are connected, which allows him initialize the distance to a node with the weight of the direct edge from the source node. For an account of the early history of shortest path algorithms we refer to Schrijver [37].

4.5 Complexity

As Bellman's algorithm [11] the derived algorithm has polynomial time complexity. One can see that the lattice $\langle \mathscr{E}, \dot{\geq} \rangle$ is *noetherian*, i.e., it satisfies the

ascending chain condition [29] (i.e., every strictly ascending chain $x_1 \dot{\geq} x_2 \dot{\geq} \ldots$ of elements eventually terminates), which guarantees termination of the iterative algorithm, since $\mathcal{F}_\delta$ is monotone. Now let the constant L be the maximal weight of an edge between any two nodes in a given graph. For each node an initial path from the source node cannot contain cycles. Moreover its distance from the source node cannot be improved more than $L \times |V|$ times by a strictly increasing chain. Therefore, for a fixed graph, the length of a corresponding ascending chain in $\langle \mathscr{E}, \dot{\geq} \rangle$ is $\mathcal{O}(|V|^2)$ which bounds the number of while-loop iterations.

Both the first algorithm in Figure 3 and the optimized algorithm in Figure 4 iterate through the predecessors of each node, which takes $\mathcal{O}(|V| + |E|)$ operations for each while-loop iteration. In addition the first algorithm requires an $\mathcal{O}(|V|)$ time stabilization test. Therefore, the worst-case time complexity of both algorithms is $\mathcal{O}(|V|^3 + |V|^2 \times |E|)$, or $\mathcal{O}(|V|^2 \times |E|)$ for a connected graph.

The bottleneck of the optimized algorithm is again the non-optimized iteration in lines 7–11. Since $u \in \texttt{pred}(v)$ if and only if $v \in \texttt{next}(u)$, looping through all nodes u, v such that $u \in \texttt{pred}(v)$ is equivalent to looping through all nodes u, v such that $v \in \texttt{next}(u)$. We can therefore rewrite the for-loops into:

---

for $u \in V$ **do**
  for $v \in \texttt{next}(u)$ **do**
    if $\delta[u] + W[u, v] < \delta[v]$ **then**
      $\delta[v] \leftarrow \delta[u] + W[u, v]$
      Changed $\leftarrow$ true

Using an observation from Dijkstra's algorithm, we can process the nodes with less distance from v_0 first. As a consequence *each edge* will be examined *only once*, which leads to the original complexity $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$. By further improving the algorithm to quickly locate the next node to process (employing a binary min-heap), we obtain the complexity $\mathcal{O}((|V| + |E|) \times \log(|V|))$ (which is an improvement for sparse graphs [16]).

4.6 Computing the Shortest Paths

Usually, one wants to compute not only shortest-path weights, but the vertices on shortest paths as well. Traditionally, the representation for shortest paths is implemented by a ***predecessor*** map π. In the canonical literature on graph algorithms [16], for a given graph node v, $\pi(v)$ is either another node or NIL, which means that the node is either the source or that it is unreachable. The shortest-paths algorithms traditionally set the values of π so that the chain of predecessors, originating at a vertex v, runs backwards along *some* shortest path from v_0 to v. In practice, it means that there might be several shortest paths from v_0 to v, however, the canonical algorithm chooses one of them arbitrarily [16].

In order to compute the predecessors for the shortest path, we will use the shortest-path weight property from Section 4.2. The shortest path predecessors of v with respect to the set of finite paths $\mathcal{X}$ are then defined as the predecessors of v on paths from v_0 with the minimal possible weight:

$$\texttt{dist}^{\pi}(\mathcal{X}) = \lambda v.\langle \texttt{dist}(\mathcal{X})(v), \{\tau, u, w : \tau uwv \in \mathcal{X} \ \wedge \ \|\tau\| = \texttt{dist}(\mathcal{X})(v) : u\}\rangle \tag{36}$$

where the codomain of $\texttt{dist}^{\pi}$ is

$$\mathscr{P} = V \to (\mathbb{N} \cup \{\infty\}) \times \wp(V) \tag{37}$$

To derive an algorithm to compute the shortest path predecessors for a given graph, we formulate $\mathscr{P}$ as a complete lattice with an order $\sqsubseteq$, build a Galois connection between $\langle \wp(V_w^+), \subseteq \rangle$ and $\langle \mathscr{P}, \sqsubseteq \rangle$, and employ fixed-point fusion.

In order to simplify the notation, in the remainder of this section we use $\downarrow_1$ and $\downarrow_2$ to refer to the first and second projections of a pair, respectively. The partial order and meet operations on elements π_1, π_2 of $\mathscr{P}$ use a function-lifted lexicographical ordering with respect to componentwise orders $\geq$ and $\subseteq$:

$$\left[\begin{array}{l} \pi_1 \sqsubseteq \pi_2 \equiv \forall u \in V : \pi_1(u)\downarrow_1 > \pi_2(u)\downarrow_1 \quad \vee \\ \qquad\qquad\qquad\qquad (\pi_1(u)\downarrow_1 = \pi_2(u)\downarrow_1 \quad \wedge \ \pi_1(u)\downarrow_2 \subseteq \pi_2(u)\downarrow_2) \end{array}\right] \tag{38}$$

$$[\ \pi_1 \sqcup \pi_2 = \lambda u.\phi(\pi_1(u), \pi_2(u))\ ], \text{ where}$$

$$\phi(\langle m_1, r_1\rangle, \langle m_2, r_2\rangle) = \begin{cases} \langle m_2, r_1\rangle & \text{if } m_1 > m_2 \\ \langle m_1, r_2\rangle & \text{if } m_2 > m_1 \\ \langle m_1, r_1 \cup r_2\rangle & \text{otherwise} \end{cases} \tag{39}$$

One can see, that $\langle \mathscr{P}, \sqsubseteq \rangle$ is a complete lattice with $\bot_{\mathscr{P}} = \lambda u.\langle \infty, \emptyset\rangle$. Similarly to Section 4.3, in order to build a Galois connection between $\langle \wp(V_w^+), \subseteq \rangle$ and $\langle \mathscr{P}, \sqsubseteq \rangle$, using $\texttt{dist}^{\pi}$ as a lower adjoint, we show again that $\texttt{dist}^{\pi}$ is distributive with respect to $\sqcup$:

Lemma 7

$$\left[\ \bigsqcup_i \texttt{dist}^{\pi}(\mathcal{X}_i) = \texttt{dist}^{\pi}(\bigcup_i \mathcal{X}_i)\ \right]$$

Proof. Similar to the proof of Lemma 5, using case analysis on the arguments to the helper function ϕ (39). □

The computation of the functional $\mathcal{F}_{\pi}$ for the shortest-path predecessors, such that

$$\texttt{dist}^{\pi} \circ p_{G_w} = \mathcal{F}_{\pi} \circ \texttt{dist}^{\pi} \tag{40}$$

is similar to the derivation from Section 4.3, using Lemma 7. The final result is stated by the following lemma:

Lemma 8

$$\texttt{dist}^{\pi} \circ p_{G_w} = \mathcal{F}_{\pi} \circ \texttt{dist}^{\pi}$$

where $\mathcal{F}_{\pi}$ is of type $\langle \mathscr{P}, \sqsubseteq \rangle \to \langle \mathscr{P}, \sqsubseteq \rangle$ is defined for all $\mathcal{X}$ by

$$\mathcal{F}_\pi(\mathcal{X}) = \lambda v.\langle m, r\rangle,\ \text{where } m = \min\{u : u \in \texttt{pred}(v) : \mathcal{X}(u)\downarrow_1 + W(\langle u, v\rangle)\}$$

$$r = \left\{ u \;\middle|\; \begin{array}{l} u \in \texttt{pred}(v) \\ \mathcal{X}(u)\downarrow_1 < \infty \\ \mathcal{X}(u)\downarrow_1 + W(\langle u, v\rangle) = m \end{array} \right\} \tag{41}$$

Thus, the sets of predecessors in the single-source shortest paths are then computed as a least fixed point according to the following theorem:

Theorem 3

$$\texttt{dist}^\pi(\mathsf{P}_{G_w}) = \mathsf{lfp}_{\sqsubseteq}\,(\lambda\mathcal{X}.(\lambda v.\langle(v = v_0\ ?\ 0 : \infty), \emptyset\rangle) \sqcup \mathcal{F}_\pi(\mathcal{X})) \tag{42}$$

where the least fixed point $\mathsf{lfp}_{\sqsubseteq}$ is computed with respect to the ordering $\sqsubseteq$ over $\mathscr{P}$, starting from $\bot_{\mathscr{P}} = \lambda u.\langle\infty, \emptyset\rangle$.

Proof. Similarly to the proof of Theorem 2, using distributivity of `dist`, Lemma 8, fixed-point fusion (2) and inlining $\texttt{dist}^\pi(\{v_0\}) = \lambda v.\langle(v = v_0\ ?\ 0 : \infty), \emptyset\rangle$ □

Note that unlike traditional algorithms for the single-source shortest path problem [11,28], our algorithm computes *all* possible shortest paths from the source node. The complexity of the algorithm is determined by the height of the lattice $\langle\mathscr{P}, \sqsubseteq\rangle$, which is $\mathcal{O}(|V|^3)$. However, updating the minimum and the set of predecessors can be performed within the same loop (lines 8–11 in Figure 4):

```
for v ∈ V do
  for u ∈ pred(v) do
    d ← δ[u] + W[u, v]
    if d ≤ δ[v] then
      δ[v] ← d
      if d < δ[v] then
        π[v] ← {u}
      else
        π[v] ← π[v] ∪ {u}
      Changed ← true
```

This gives the same complexity boundary as in Section 4.5: $\mathcal{O}(|V|^3 \times |E|)$ in the worst case. By rewriting the algorithm with `next`() instead of `pred`() and applying observations from Dijkstra's algorithm analysis, one can obtain the complexity bound $\mathcal{O}(|V|^2)$ for the optimized iteration through the set of nodes.

5 Related Work

Two different schools have been working in parallel for the last forty years: the school of program calculation and the school of static program analysis. The

intrinsic goal of the first school is to derive algorithms from the specification of properties of interest. The second school was historically interested in computing a *sound* approximation of a property of a program semantics. In this section we give a brief overview of these two lines of research which we have attempted to bridge in the present paper.

Calculational approaches to graph algorithms. A number of approaches have been applied to derive graph algorithms since the seventies, originating in formulating path problems in terms of linear algebra. Carré [14] presented an algebraic structure to solve extremal network routing problems, such that a function is minimized or maximized on a particular path in a graph. He showed how extremal problems from this class can be expressed in terms of matrix equations and solved using a toolset from linear algebra. Later, Backhouse and Carré [8] showed the correspondence of the algebra for extremal graph problems and the algebra of regular languages. The idea was later extended to derive the exact implementation of Dijkstra's shortest path algorithm [9].

In the beginning of the nineties ideas from domain theory were applied to compute extremal properties on paths of graphs using fixed-point computations: Van den Eijnde [39] considered computation of path properties in graphs using monotone operators, satisfying certain restrictions and called these operators *conservative*. Van den Eijnde formulated a generalized fixed-point theorem, stating computation of a least fixed point of a monotone functional as a Kleene iteration. The property of interest was then defined as an *under-approximation* of the monotone function. As an example, this approach was applied to the ascending reachability problem. In contrast to our work, Van den Eijnde did not apply the Galois connection machinery to define the properties and prove them appropriate for an algorithm derivation. All the used toolset was later formalized as the *fixed-point calculus* [1]. The interplay between Galois connections and fixed points has later been established by Backhouse [6].

Abstract interpretation and distributive frameworks. In parallel with the above line of research, Cousot and Cousot developed and refined the abstract interpretation framework [20,21]. In their 1979 paper [22], they mention various instances of distributive frameworks for imperative program analysis as particular cases of abstract interpretation, i.e., constant propagation, trace (or path) reachability properties, where Galois connections are defined appropriately [22]. In the same work, they prove a connection between properties, defined as *meet-over-all paths* and ones described by monotone functions: the former is generally more precise than the latter but the two are identical in a distributive framework. Ten years later, Cai and Paige describe a nondeterministic iterative schema that in the case of finite iteration generalizes the "chaotic iteration" of Cousot and Cousot for computing fixed points of monotone functions efficiently (in particular, *incrementally*) and show how to apply this technique to design fast non-numerical algorithms, such as variable reachability and cycle detection in a program flow graph [13]. Whereas the current paper illustrates how to get from a graph specification to a provably correct (but not necessarily O-optimal) algorithm, we

believe that such chaotic iteration techniques may be the key to derive optimized versions of our calculated graph algorithms in a more principled manner.

Cousot and Cousot [22] initially formalized programs as flow graphs, but the framework was later generalized to *transition systems* [17, 23] which are not limited to describing formal semantics. Since then the abstract interpretation framework has been used to formalize other concepts than static analyses, e.g., program transformations [25] and to connect various forms of formal semantics [19].

Cooper, Harvey and Kennedy [15] point out that the equations to compute dominance form a distributive framework [31]. This fact allows them to state that the iterative algorithm for dominance computation will discover the maximal fixed-point solution and halt. Notably, the equations for Dom, presented by Cooper, Harvey and Kennedy in [15] are given *as is*, i.e., with no connection to the definition of dominance in terms of paths. In contrast we justify these equations by deriving them and a corresponding algorithm directly from the definition.

Backhouse [6, Section 6.2] used shortest paths as a motivating example for introducing fixed-point fusion in his lecture notes. In a later work on the shortest-path problem, Backhouse applied the fixed-point fusion theorem to a set of all paths, considered as a context-free language [7, Example 57], which gave the same solution as we obtained. We have nevertheless chosen to include the detailed development along with our complexity boundary discussion, as a second example of the technique.

Future Work. A natural next step is to incorporate more benefits of point-free style, such as those provided by relational compositions and factors for the systematic calculation of program analyses, as well as make use of tool support [38] for deriving graph algorithms.

6 Conclusion

In this work we explored two classical graph problems, formulated in terms of finite paths through a graph: dominance and the single-source shortest paths. Applying the toolset traditional to fixed-point calculus and semantics-based program analysis, we derived iterative, polynomial-time algorithms for both properties. We formalized definitions of the properties as adjoints in appropriate Galois connections. By fusing these with a least fixed point of a monotone path functional, we obtained polynomial-time algorithms for computing the properties directly.

The derived algorithms obtained are strikingly similar to independently discovered algorithms from the literature. Their calculations constitute constructive correctness proofs in contrast to, e.g., an invariant argument for Dijkstra's algorithm by contradiction [16]. The derivations further witness the wide applicability of the toolset behind fixed-point calculus and abstract interpretation.

Acknowledgements. We are grateful to Olivier Danvy for comments, which helped to improve the presentation of the paper, and to Jeremy Gibbons for suggestions on both formalism and terminology. We sincerely acknowledge the MPC 2012 reviewers, who *all* provided *excellent* feedback on the submission. In particular, we want to thank Reviewer #1 for suggesting the idea of using factors and showing how to apply it to compute the dominance algorithm,[8] which drastically simplified the derivations in Section 3. Finally, we want to express our gratitude to Shin-Cheng Mu for his dedication to bring out the best of the paper.

References

1. Aarts, C., Backhouse, R.C., Boiten, E.A., Doornbos, H., van Gasteren, N., van Geldrop, R., Hoogendijk, P.F., Voermans, E., van der Woude, J.: Fixed-point calculus. Information Processing Letters 53, 131–136 (1995)
2. Allen, F.E.: Control flow analysis. SIGPLAN Not. 5, 1–19 (1970)
3. Allen, F.E., Cocke, J.: Graph theoretic constructs for program control flow analysis. Technical Report IBM Research Report RC 3923, Thomas J. Watson Research Center, Yorktown Heights, NY, USA (1972)
4. Alstrup, S., Harel, D., Lauridsen, P.W., Thorup, M.: Dominators in linear time. SIAM J. Comput. 28(6), 2117–2132 (1999)
5. Appel, A.W.: Modern Compiler Implementation in {C, Java, ML}. Cambridge University Press, New York (1998)
6. Backhouse, R.: Chapter 4: Galois Connections and Fixed Point Calculus. In: Backhouse, R., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 89–148. Springer, Heidelberg (2002)
7. Backhouse, R.C.: Regular algebra applied to language problems. J. Log. Algebr. Program. 66(2), 71–111 (2006)
8. Backhouse, R.C., Carré, B.A.: Regular algebra applied to path-finding problems. Journal of the Institute of Mathematics and Applications 15, 161–186 (1975)
9. Backhouse, R.C., van den Eijnde, J.P.H.W., van Gasteren, A.J.M.: Calculating path algorithms. Sci. Comput. Program. 22(1-2), 3–19 (1994)
10. Barbuti, R., Bernardeschi, C., De Francesco, N.: Checking security of Java bytecode by abstract interpretation. In: Proceedings of the 2002 ACM Symposium on Applied Computing, Madrid, Spain, pp. 229–236. ACM (March 2002)
11. Bellman, R.: On a routing problem. Quarterly of Applied Mathematics 16, 87–90 (1958)
12. Bird, R., de Moor, O.: The Algebra of Programming. Prentice-Hall (1996)
13. Cai, J., Paige, R.: Program derivation by fixed point computation. Sci. Comput. Program. 11(3), 197–261 (1989)
14. Carré, B.A.: An algebra for network routing problems. J. Inst. Maths Applics. 7, 273–294 (1971)
15. Cooper, K.D., Harvey, T.J., Kennedy, K.: A simple, fast dominance algorithm. Technical report, Rice University Houston, Texas, USA (2001)
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)

[8] The degree of elaboration of the review made us speechless for a while.

17. Cousot, P.: Semantic foundations of program analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, ch. 10, pp. 303–342. Prentice-Hall (1981)
18. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
19. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Comput. Sci. 277(1-2), 47–103 (2002)
20. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Dunod, Paris, France, pp. 106–130 (1976)
21. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Sethi, R. (ed.) Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252 (January 1977)
22. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Rosen, B.K. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (January 1979)
23. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2-3), 103–179 (1992)
24. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
25. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Mitchell, J.C. (ed.) Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages, Portland, Oregon, pp. 178–190 (January 2002)
26. Cousot, P., Cousot, R.: Basic concepts of abstract interpretation. In: Jacquart, R. (ed.) Building the Information Society, pp. 359–366. Kluwer Academic Publishers (2004)
27. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2002)
28. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
29. Dummit, D., Foote, R.: Abstract algebra. Prentice Hall (1999)
30. Fluet, M., Weeks, S.: Contification using dominators. In: Leroy, X. (ed.) Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), Firenze, Italy, pp. 2–13 (September 2001)
31. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. J. ACM 23, 158–171 (1976)
32. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1, 121–141 (1979)
33. Midtgaard, J., Jensen, T.: A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 347–362. Springer, Heidelberg (2008)
34. Might, M.: Abstract Interpreters for Free. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 407–421. Springer, Heidelberg (2010)
35. Milanova, A., Vitek, J.: Static Dominance Inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
36. Prosser, R.T.: Applications of boolean matrices to the analysis of flow diagrams. In: Proceeding of the Eastern Joint IRE-AIEE-ACM Computer Conference, pp. 133–138. ACM, Boston (1959)

37. Schrijver, A.: On the History of Combinatorial Optimization (till 1960). In: Aardal, K., Nemhauser, G.L., Weismantel, R. (eds.) Handbook of Discrete Optimization, pp. 1–68 (2005)
38. Silva, P.F., Oliveira, J.N.: 'Galculator': functional prototype of a Galois-connection based proof assistant. In: Antoy, S., Albert, E. (eds.) PPDP 2008: Proceedings of the 10th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 44–55 (July 2008)
39. van den Eijnde, J.P.H.W.: Conservative Fixpoint Functions on a graph. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) MPC 1992. LNCS, vol. 669, pp. 80–99. Springer, Heidelberg (1993)
40. van Gasteren, A.J.M.: On the shape of mathematical arguments. Springer-Verlag New York, Inc. (1990)

First-Past-the-Post Games

Roland Backhouse

School of Computer Science, University of Nottingham,
Nottingham NG8 1BB, England
roland.backhouse@nottingham.ac.uk

Abstract. Informally, a first-past-the-post game is a (probabilistic) game where the winner is the person who predicts the event that occurs first among a set of events. Examples of first-past-the-post games include so-called block and hidden patterns and the Penney-Ante game invented by Walter Penney. We formalise the abstract notion of a first-past-the-post game, and the process of extending a probability distribution on symbols of an alphabet to the plays of a game.

Analysis of first-past-the-post games depends on a collection of simultaneous (non-linear) equations in languages. Essentially, the equations are due to Guibas and Odlyzko but they did not formulate them as equations in languages but as equations in generating functions detailing lengths of words.

Penney-Ante games are two-player games characterised by a collection of regular, prefix-free languages. For such two-player games, we show how to use the equations in languages to calculate the probability of winning. The formula generalises a formula due to John H. Conway for the original Penney-Ante game. At no point in our analysis do we use generating functions. Even so, we are able to calculate probabilities and expected values. Generating functions do appear to become necessary when higher-order cumulatives (for example, the standard deviation) are also required.

Keywords: algorithmic problem solving, regular language, generating function, probabilistic game, Penney-Ante, block pattern, hidden pattern.

Penney-Ante is the name of a game with pennies invented by Walter Penney [Pen74]. The two-player game is interesting because it is non-transitive; the game is also used to demonstrate the use of generating functions in the calculation of probability distributions [GO81, GKP94]. Our interest in the game began as a simple, (for us) introductory exercise in probability generating functions. It has turned out to be an exercise in applying the calculational method to the analysis of the game in the general case of an arbitrary number of players — an exercise with the surprising conclusion that generating functions are not needed for the calculation of probabilities and expected values.

Analysis of the game is substantially facilitated by a collection of simultaneous (non-linear) equations between languages. In the literature, either the equations

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 157–176, 2012.
© Springer-Verlag Berlin Heidelberg 2012

are stated without proof [GKP94] or the equations are not given explicitly but translated directly into generating functions detailing lengths of words [GO81]. The contribution of this paper is to record a derivation of the equations and the associated probability distributions in which naming of word length and the use of generating functions is avoided.

Our derivation has several novel features. We introduce the abstract notion of a first-past-the-post game, and we formalise the process of extending a probability distribution on symbols of an alphabet to the plays of such a game (section 2). (Multi-player) Penney-Ante games and so-called block and hidden patterns [FS09] are shown to be instances of first-past-the-post games. Such games are characterised by a collection of regular, prefix-free languages. We derive a collection of simultaneous non-linear equations in these languages and use these to show how to calculate the probability of winning (section 4).

The equations are essentially the basis for the equations in generating functions derived by Guibas and Odlyzko [GO81]. The formula we derive generalises a formula due to John Horton Conway for the original two-player Penney-Ante game. Another instance is the formula due to A.D.Solov'ev [Sol66] for the expected number of coin tosses until a given (contiguous) pattern appears. Like Guibas and Odlyzko [GO81], we also consider the generalisation of Penney-Ante games to an arbitrary number of players.

We show in section 5 that the equations in languages do not have a unique solution. This is surprising and demands further investigation.

1 Preliminaries

We assume familiarity with the use of regular expressions to denote languages. To avoid confusion with ordinary addition, the usual symbol "$\cup$" is used to denote set union, and not "+" (as often used in regular expressions). The symbol ε denotes the empty word and T denotes a finite set (which is fixed throughout the paper). In line with other literature on the Penney-Ante game, capital letters at the beginning of the alphabet (A, B, etc.) denote words and capital letters at the end of the alphabet (U,V, etc.) denote sets of words. The elements of T are called *symbols* and sets of words are called *languages*. Symbols are denoted by lower case letters (a, b, etc.). The length of word A is denoted by $\#A$. Concatenation of words and of languages is denoted by juxtaposition.

For any word A different from the empty word, $pre.A$ is the prefix of A obtained by discarding the last symbol in A. The function pre is extended to sets by the definition: for all languages V,

$$pre.V \;=\; \{A{,}a \;:\; A{\in}T^* \wedge a{\in}T \wedge Aa{\in}V \;:\; A\} \quad .$$

(We use the Eindhoven notation for quantifications [Bac86, GS93, Bac03]. The notation $\{vars{:}rng{:}term\}$ abbreviates $\langle\cup\, vars : rng : \{term\}\rangle$. In conventional notation, the dummy a in the definition of pre would be existentially quantified.)

Repeated application of pre one or more times is denoted by pre^+ and zero or more times by pre^*. Thus $pre^+.V$ is the set of all proper prefixes of words in V, and $pre^*.V$ is $V \cup pre^+.V$. Note that pre distributes through set union.

For calculational purposes the following property of pre^+ is used. For all words C and languages V,

$$C \in pre^+.V \quad \equiv \quad \{C\}T^+ \cap V \;\neq\; \emptyset \ .$$

2 First-Past-the-Post Games

Penney-Ante is an instance of a class of probabilistic games for which winning is characterised by the *first* occurrence of one of a set of events, and the events are words. We begin by formalising this class of games.

Definition 1. Suppose S is a subset of T^*. The set S is said to be a *first-past-the-post game* if

(a) $pre^+.S \cap S \;=\; \emptyset$.
In words, no proper prefix of a word in S is a word in S.

(b) $pre^*.S \;=\; \{\varepsilon\} \cup (pre^+.S)T$.
In words, appending an arbitrary symbol of the alphabet T to a proper prefix of a word in S gives a word that prefixes a word in S.
(This informal statement expresses only that the right side of the equation is included in the left side. The opposite inclusion is obvious from the definitions of pre^* and pre^+.)

A *play* of the game is an element of $pre^*.S$. A *complete play* of the game is an element of S. □

A play of the game can be thought of as repeatedly throwing a die with sides labelled by the elements of T. The play starts with the empty word and, as the die is thrown, the symbol that occurs is appended to the end of the play. The play is complete when the play is in S. Property (a) states that no proper prefix of a word in S is an element of S. That is, the game ends —the play is complete— immediately an element of S is recognised. Property (b) states that the plays are the empty word or arbitrary continuations of an incomplete play. It has the consequence that any throw of the die continues an incomplete play of the game. A second consequence is that S is non-empty (because the right side of the equation is a non-empty set).

Example 1. With $T = \{a,b\}$, the table below shows examples of languages and whether or not they fulfill properties (a) and (b) of definition 1.

Language	(a)	(b)
$\{a\}$	√	×
$\{a,ab\}$	×	×
$T^k\ (0 \leq k)$	√	√
$T^{\leq k}\ (0 < k)$	×	√
$\{a,ba,bb\}$	√	√
$\{b\}^*\{a\}$	√	√
$\{b\}^*\{a\}\{a\}^*\{b\}\{b\}^*\{a\}$	√	√

□

The set T^k, where k is some fixed natural number, exemplifies the set of complete plays in a first-past-the-post game. (See example 1.) It is the game where a die is thrown exactly k times.

Generally, the set S may be assumed to be split into disjoint sets each of which is owned by one of the players. When the play is complete, the owner of the play is the winner. The Penney-Ante game assumes that two players each choose one word. The reason for this assumption is that the game is then non-transitive: if one player chooses one word it is always possible for the second player to choose a word that gives a better than evens chance of winning. This, however, is not the focus of our investigation. For our purposes, the number of players can be arbitrary as can be the number of words each player chooses. There is no reason why games with fewer or more than two players should not be allowed, or why each player should choose just one word. "Games" with one player are associated with pattern-matching problems. See section 4.

We assume that the outcome of each single throw of the die is given by some probability distribution p. The outcomes of separate throws are assumed to be independent. This suggests the following definition.

Definition 2. Let p be a function with domain T and range the set of real numbers. We define the function h_p with domain T^* inductively by

(a) $h_p.\varepsilon \;=\; 1 \quad ,$
(b) $h_p.Ba \;=\; h_p.B \times p.a \quad$, for all $B \in T^*$ and $a \in T$.

The function h_p is extended to languages by defining, for all V, where $V \subseteq T^*$,

$$h_p.V \;=\; \langle \Sigma A : A \in V : h_p.A \rangle \quad .$$

The function e_p is defined on languages by, for all V, where $V \subseteq T^*$,

$$e_p.V \;=\; \langle \Sigma A \;:\; A \in V \;:\; h_p.A \times \# A \rangle \quad .$$

(Note: these definitions assume that the summations are well defined. In all the concrete examples discussed in this paper, this is indeed the case.) □

Theorem 1 shows that, if p is a probability distribution on T, h_p is a probability distribution on a first-past-the-post game S. The value of $e_p.S$ is then interpreted as the "expected" length of the game. It is important to note, however, that definition 2 does *not* assume that p is a probability distribution. We apply definition 2 just as often when p and/or h_p cannot be viewed as probability distributions.

Typically languages are defined syntactically — by a combination of regular expressions and equations (aka grammars). Unambiguity of syntactic definitions is useful in the evaluation of the functions h_p and e_p. This is made precise in the following definitions and lemmas.

Definition 3 (Unambiguous Expressions). Let U and V be expressions denoting languages $L.U$ and $L.V$, respectively. We say that the expression "$U \cup V$" is *unambiguous* if $L.U \cap L.V = \emptyset$ (i.e. the languages are disjoint). We say that the expression "UV" is *unambiguous* if, for all words A, A', B and B',

$$A{,}A' \in L.U \wedge B{,}B' \in L.V \wedge AB = A'B' \;\;\Rightarrow\;\; A = A' \wedge B = B' \quad .$$

We say that the expression "U^*" is *unambiguous* if, for all natural numbers k and k', and sequences of words A_i $(1 \leq i \leq k)$ and B_j $(1 \leq j \leq k')$ all of which are elements of $L.U$,

$$A_1 \ldots A_k = B_1 \ldots B_{k'} \;\Rightarrow\; k = k' \wedge \langle \forall i : 1 \leq i \leq k : A_i = B_i \rangle \quad . \qquad \square$$

Expressions and languages are, of course, different in the same way that names and people are different. ("Winston Churchill" is the name of a famous Englishman. The name consists of a forename and a surname, whilst the person has a mother and father, etc.) Definition 3 has been formulated in a way that makes the difference clear. Henceforth however, we are not so precise and we leave it to the reader to determine whether we are referring to the syntactic form of an expression or to the language that is denoted by the expression. So, for example, a less precise formulation of the first clause of definition 3 is

"the expression $U \cup V$ is *unambiguous* if $U \cap V = \emptyset$".

We trust that the reader will have no difficulty in understanding what is meant.

An example of unambiguity is the expression $\{\varepsilon\} \cup (pre^+.S)T$ in definition 1. Obviously $\{\varepsilon\} \cap (pre^+.S)T = \emptyset$ because $\{\varepsilon\}$ is the set of words of length zero whilst $(pre^+.S)T$ contains only words of length at least one. So the "$\cup$" operator is unambiguous. Also obvious on length considerations is that the (implicit) concatenation operator in the expression $(pre^+.S)T$ is unambiguous. In general, an expression denoting the concatenation of two languages of which one is a subset of T^k for some k (i.e. all the words in the language have the same length) is unambiguous. Deterministic finite-state machines also exemplify the use of unambiguous expressions in order to define a language. A deterministic finite-state machine corresponds to a system of equations in languages; the right sides of the equations are disjoint unions of expressions of the form ε or aU (where U denotes the language recognised by some state of the machine).

The following lemma is the key to evaluating probabilities and expected values in the context of first-past-the-post games. Note how the equations for e_p resemble the equations for calculating derivatives.

Lemma 1. If $U \cup V$ is an unambiguous expression,

$$h_p.(U \cup V) \;=\; h_p.U + h_p.V \quad \text{, and}$$

$$e_p.(U \cup V) \;=\; e_p.U + e_p.V \quad .$$

If UV is an unambiguous expression,

$$h_p.UV \;=\; h_p.U \times h_p.V \quad \text{, and}$$

$$e_p.UV \;=\; h_p.U \times e_p.V \;+\; e_p.U \times h_p.V \quad .$$

Proof. Straightforward manipulation of quantifier expressions. $\square$

We now consider the consequences of the function p being a probability distribution. Recall that we use S to denote a first-past-the-post game. Because it plays an important role in what follows, we use N throughout to denote $pre^{+}.S$. (The symbol "N" is the one used in [GKP94]; it may be read as a mnemonic for "N"ot complete.) With this notation, the two clauses in definition 1 of a first-past-the-post game become:

(1) $N \cap S \;=\; \emptyset$, and

(2) $N \cup S \;=\; \{\varepsilon\} \cup NT$.

From (2), it is easy to see that $h_p.T = 1 \Rightarrow h_p.S = 1$. See the calculation below.

$$
\begin{array}{ll}
 & h_p.S = 1 \\
= & \{ \quad \text{heading towards (2) in definition of a game,} \\
 & \quad\;\; \text{we add } h_p.N \text{ to both sides} \quad \} \\
 & h_p.N \;+\; h_p.S \;\;=\;\; h_p.N \;+\; 1 \\
= & \{ \quad \text{by definition, } 1 = h_p.\{\varepsilon\}\text{; assumption: } h_p.T = 1 \quad \} \\
 & h_p.N \;+\; h_p.S \;\;=\;\; h_p.N \times h_p.T \;+\; h_p.\{\varepsilon\} \\
= & \{ \quad \text{expressions } N{\cup}S \text{ and } NT \cup \{\varepsilon\} \text{ are unambiguous,} \\
 & \quad\;\; \text{lemma 1} \quad \} \\
 & h_p.(N \cup S) \;\;=\;\; h_p.(NT \cup \{\varepsilon\}) \\
= & \{ \quad \text{definition of a game: (2)} \quad \} \\
 & \mathsf{true} \;\; .
\end{array}
$$

This suggests that, if p is a probability distribution on T, h_p is a probability distribution on complete plays. This fact appears to be taken for granted in [GKP94] and [GO81]. (At least, we have been unable to find anything that we would recognise as a proof.) We think it is important to make the theorem explicit and provide a proof. (The proof is not calculational because it links the formal definitions with the informal notion of relative frequencies.)

Theorem 1. If p is a probability distribution on the alphabet T (i.e. $p.a$ is the relative frequency of the occurrence of symbol a when the die is thrown and, thus, $h_p.T = 1$) and throws of the die are independent, the function h_p is a probability distribution on complete plays of a first-past-the-post game S. Specifically, for an arbitrary word A in S, $h_p.A$ is the relative frequency that the word A is a complete play of the game. Moreover, h_p is a probability distribution on 2^S (the set of subsets of S); if $U \subseteq S$, then $h_p.U$ is the relative frequency with which a word in U occurs as a complete play.

Proof. Suppose $A \in pre^{*}.S$. We prove by induction on the length of A that $h_p.A$ is the relative frequency with which the word A occurs as a prefix of a complete play of the game.

When the length of A is zero, $A = \varepsilon$. The empty word occurs in every play of the game. That is, the relative frequency of ε as a prefix of a complete play of the game is 1, which equals $h_p.\varepsilon$ by definition. This proves the basis.

Now suppose the length of A is at least one. Suppose $A = Ba$ for some $B \in T^*$ and $a \in T$. Since $B \in pre^*.S$, and the length of B is less than the length of A, we may assume inductively that $h_p.B$ is the relative frequency with which the word B occurs as a prefix of a complete play of the game. But $B \in pre^+.S$ and so, by definition 1(b), $h_p.B$ is the relative frequency with which words of the form Bb, for some $b \in T$, occur as a prefix of a complete play. Since $p.a$ is the relative frequency that a occurs, the independence assumption implies that $h_p.B \times p.a$ is the relative frequency with which Ba occurs as a prefix of a complete play. But $h_p.A = h_p.B \times p.a$ by definition. In this way, the induction step is verified.

A corollary of this inductive argument and definition 1(a) is that, when A is a complete play, $h_p.A$ is the relative frequency of A among complete plays. (Because of definition 1(a), a complete play only occurs as a prefix of itself and no other plays.)

By the definition of a probability distribution, it is an immediate corollary that the extension of h_p to subsets of S is a probability distribution. □

Note that h_p is just a function on arbitrary languages. As shown above, it is a probability distribution on S and on 2^S whenever p is a probability distribution on T but we apply it elsewhere to arbitrary languages. An example of where h_p is used in this way is the following lemma.

Lemma 2. Suppose S is the set of complete plays in a first-past-the-post game and N is the set of incomplete plays. Suppose the symbols in T occur with probability distribution given by p. Then

$$e_p.S \;=\; h_p.N \quad .$$

Proof. First,

$$e_p.S \;=\; h_p.N$$

$=$ { heading towards (2) in definition of a game, we add $e_p.N$ to both sides }

$$e_p.N \;+\; e_p.S \;\;=\;\; e_p.N \;+\; h_p.N \quad .$$

But

$$e_p.N \;+\; e_p.S$$

$=$ { expression $N \cup S$ is unambiguous, lemma 1 }

$$e_p.(N \cup S)$$

$=$ { (2) }

$$e_p.(NT \cup \{\varepsilon\})$$

$= \quad \{ \quad$ expression $NT \cup \{\varepsilon\}$ is unambiguous, lemma 1 $\}$

$h_p.N \times e_p.T \ + \ e_p.N \times h_p.T \ + \ e_p.\{\varepsilon\}$

$= \quad \{ \quad$ p is a probability distribution on T, so $h_p.T = 1$;
also, for each $A \in T$, $\#A = 1$. So $e_p.T = 1$.
By definition, $e_p.\{\varepsilon\} = 0$. $\}$

$h_p.N \ + \ e_p.N \quad .$

The lemma follows by combining the two calculations (using symmetry of addition). □

Example 2. If $S = \{a, ba, bb\}$ and $p.a = q$ and $p.b = r$, where $q + r = 1$, then $N = \{\varepsilon, b\}$ and $e_p.S = 1 \times q + 2 \times r \times q + 2 \times r \times r = 1 + r = h_p.N$.

If $S = \{b\}^*\{a\}$ and $p.a = q$ and $p.b = r$, then $N = \{b\}^*$; so $e_p.S = (1-r)^{-1}$. (Note how much easier it is to use the lemma than to calculate $e_p.S$ directly from its definition.) □

3 Prefix-Free Languages

A requirement on games is that complete plays are prefix-free languages (definition 1(a)). Any language V can be reduced to a maximal, prefix-free language by selecting the words that have no proper prefixes in V. Specifically, if V is a language, the set $PF.V$, called the *prefix-free reduction* of V, is defined by

$$PF.V \quad = \quad V \cap \neg(VT^+) \quad .$$

The element-wise formulation of $PF.V$ is that, for all languages V and all words C,

$$C \in PF.V \quad \equiv \quad C \in V \ \wedge \ \neg \langle \exists D,E \ : \ D \in V \wedge E \in T^+ \ : \ DE = C \rangle \quad .$$

That is, $PF.V$ is the set of words in V that do not have a proper prefix in V.

Example 3. It is sometimes of interest to determine the expected length of a sequence of observations that culminates in a given "pattern". Patterns are classified as either *block* or *hidden* [FS09]. Formally, let A be an arbitrary word over the alphabet T. Then $PF.T^*\{A\}$ models the process of observing sequences of letters until the word A first occurs contiguously (i.e. as a "block" pattern). If $1 \le n$ and $A = a_1 a_2 \ldots a_n$, then $PF.T^*\{a_1\}T^*\{a_2\} \ldots T^*\{a_n\}$ models the process of observing sequences of letters until all the letters of A occur in order but not necessarily contiguously (i.e. as a "hidden" pattern).

Lemma 6 establishes that $PF.T^*W$ is a first-past-the-post game for arbitrary non-empty set W. Thus $PF.T^*\{A\}$ and $PF.T^*\{a_1\}T^*\{a_2\} \ldots T^*\{a_n\}$ are both first-past-the-post games.

(Of course, $PF.W$ is not a first-past-the-post game for arbitrary non-empty set W. A simple counter-example is $W = \{a\}$ since $PF.\{a\} = \{a\}$. When $T \neq \{a\}$ this is not a first-past-the-post game. See example 1.) □

The following lemma expresses formally the process of "reducing" V to $PF.V$.

Lemma 3. Every word in V has a unique prefix in $PF.V$.

Proof. Let C be a word in V. Consider a linear search of the prefixes of C, starting with the empty word and iteratively increasing the length of the prefix, to find a word that is an element of V. The search will eventually terminate successfully because C is itself such a word. An invariant of the algorithm is that the current prefix is an element of $\neg(VT^{+})$. The prefix that is found is thus an element of both V and $\neg(VT^{+})$. It is clearly unique because any other prefixes of C are either not in V or in VT^{+}. □

Several properties of the function PF will be used later.

Lemma 4. $PF.V$ is prefix-free. That is, for all V such that $V \subseteq T^{*}$,

$$pre^{+}.(PF.V) \cap PF.V \ = \ \emptyset \ .$$

Proof. This is, in fact, a corollary of lemma 3 but is proved directly as follows. We have, for all words C,

$$C \in pre^{+}.(PF.V) \cap PF.V$$

$$= \quad \{ \quad \text{definition of } pre^{+} \quad \}$$

$$\langle \exists E : E \in PF.V : E \in \{C\}T^{+} \rangle \wedge C \in PF.V$$

$$\Rightarrow \quad \{ \quad PF.V \subseteq V \quad \}$$

$$\langle \exists E : E \in PF.V : E \in VT^{+} \rangle$$

$$\Rightarrow \quad \{ \quad PF.V \subseteq \neg(VT^{+}) \quad \}$$

$$\mathsf{false} \ .$$

□

Remark: The prefix-free reduction of V is a maximal prefix-free reduction in the sense that it is *prefix-free* (lemma 4) and it is the *largest* prefix-free subset of V, i.e. for all languages U,

$$(U \subseteq V \ \equiv \ U \subseteq PF.V) \ \Leftarrow \ U \cap pre^{+}.U \ = \ \emptyset \ .$$

End of Remark

Lemma 5. For all languages V and U, the expression $(PF.V)U$ is unambiguous. That is, for all languages V and all words C, C', D and D',

$$CD = C'D' \wedge C \in PF.V \wedge C' \in PF.V \ \Rightarrow \ C = C' \wedge D = D' \ .$$

Proof. We begin with a simple property of words.

$$CD = C'D'$$

$$\Rightarrow \quad \{ \quad \text{case analysis on } \#C \text{ and } \#C' \text{, definition of } pre^{+} \quad \}$$

$$C = C' \ \vee \ C \in pre^{+}.C' \ \vee \ C' \in pre^{+}.C \ .$$

We now show that, assuming $C \in PF.V \wedge C' \in PF.V$, the second and third disjuncts are false.

$$C \in pre^+.C' \;\wedge\; C' \in PF.V$$

$\Rightarrow$ { definition of pre^+ }

$$C \in pre^+.(PF.V)$$

$\Rightarrow$ { lemma 4 }

$$\neg(C \in PF.V) \quad .$$

We conclude that

$$C \in pre^+.C' \;\wedge\; C \in PF.V \;\wedge\; C' \in PF.V \;\equiv\; \mathsf{false} \quad .$$

Interchanging the roles of C and C', the third disjunct is also false. The lemma follows straightforwardly. □

4 Block Patterns and Penney-Ante Games

We now specialise the analysis to block patterns and Penney-Ante-type games. In Penney-Ante games, each player chooses a word. A die (with $|T|$ faces each of which bears one of the elements of T, but not necessarily fair) is then thrown repeatedly until one of the chosen words occurs as a suffix of the play. The player who made the choice is declared the winner. For example, suppose the alphabet has two symbols a and b, one player chooses the word a and the second player chooses the word bb. There are just three complete plays of this game: the words a, ba and bb. The first player wins in the first two cases and the second player wins in the third case. Note that this is a first-past-the-post game — see example 1. Recognition of a block pattern (see example 3) is a special case of a Penney-Ante game with one player.

Consider a set W of words over an alphabet T. Note that we do not assume at this stage that W is finite.

The set S is defined to be the set of minimal-length words that end in a word in W. Formally, (in standard regular-language notation)

$$S \;=\; T^*W \cap \neg(T^*WT^+) \quad .$$

Equivalently, $S = PF.T^*W$.

Returning to the example above, taking W to be $\{a,bb\}$ we have:

$$S \;=\; \{a,b\}^*\{a,bb\} \cap \neg(\{a,b\}^*\{a,bb\}\{a,b\}^+) \;=\; \{a,ba,bb\} \quad .$$

In this very simple example, the set S is finite; this is not the case in general.

Lemma 6. For all W such that $W \subseteq T^*$ and $\emptyset \neq W$, $PF.T^*W$ is a first-past-the-post game.

Proof. Let S denote $PF.T^*W$ and let N denote $pre^+.S$. Then that S satisfies 1(a) in the definition of a first-past-the-post game,

$$(3) \quad N \cap S = \emptyset \quad ,$$

is immediate from lemma 4 by instantiating V to T^*W.

It remains to verify the property 1(b). Now,

$$pre^*.S \ = \ \{\varepsilon\} \cup (pre^+.S)T$$

$=$ { $pre^*.S \ = \ S \cup pre^+.S$, $N = pre^+.S$ }

$$S \cup N \ = \ \{\varepsilon\} \cup NT$$

$=$ { T is the alphabet }

$$(S \cup N) \cap T^* \ = \ (\{\varepsilon\} \cup NT) \cap T^*$$

$=$ { $T^* = \{\varepsilon\} \cup T^+$ }

$$(S \cup N) \ \cap \ (\{\varepsilon\} \cup T^+) \ = \ (\{\varepsilon\} \cup NT) \cap T^*$$

$=$ { distributivity of intersection over union,
assumption: $\emptyset \neq W$. So $\{\varepsilon\} \subseteq S \cup N$ }

$$\{\varepsilon\} \cup ((S \cup N) \cap T^+) \ = \ \{\varepsilon\} \cup (NT \cap T^*)$$

$=$ { $NT \subseteq T^+ \subseteq T^*$,
cancellation property of languages: ε has length 0
and words in T^+ have length at least 1 }

$$(S \cup N) \cap T^+ \ = \ NT$$

$=$ { definition of set concatenation and equality }

$$\langle \forall B,a \ : \ B \in T^* \wedge a \in T \ : \ Ba \in S \cup N \equiv B \in N \rangle \quad .$$

Now, for all $B \in T^*$ and $a \in T$, we have

$$Ba \in S \cup N$$

$\Rightarrow$ { definition of pre }

$$B \in pre.(S \cup N)$$

$=$ { $S \cup N = pre^*.S$ }

$$B \in pre^+.S$$

$=$ { $N = pre^+.S$ }

$$B \in N \quad .$$

For the opposite implication, choose an arbitrary word C in W. Then, for all $B \in T^*$ and $a \in T$, we have

$$B \in N$$

$=$ { $C \in W$ }

$$B \in N \ \wedge \ BaC \in T^*W$$

$\Rightarrow$ { lemma 3, definition of S }

$$B \in N \ \wedge \ \langle \exists k : 0 \leq k \leq \#(BaC) : pre^k.(BaC) \in S \rangle$$

$\Rightarrow$ { $pre^*.B \ \cap \ S$

$\subseteq$ { assume: $B{\in}N$ }

$pre^*.N \cap S$

$=$ { $pre^*.N = pre^*.(pre^+.S) = pre^+.S = N$ }

$N \cap S$

$=$ { (3) }

$\emptyset$.

That is, assuming $B{\in}N$,
$\langle\forall k : \#(aC) \le k \le \#(BaC) : \neg(pre^k.(BaC) \in S)\rangle$ }

$\langle\exists k : 0 \le k \le \#C : pre^k.(BaC) \in S\rangle$

$\Rightarrow$ { range splitting on $k = \#C$, definition of N }

$Ba{\in}S \vee Ba{\in}N$

$=$ { definition of set union }

$Ba \in S \cup N$. □

4.1 Equations in Languages

In this section, we show how to construct from a given language W a (non-linear) system of simultaneous equations in languages. The system has one equation for each word in W (which is not necessarily finite); as we show in section 5, these equations together with the equation 1(b) uniquely characterise $PF.T^*W$. Although W need not be finite, we do assume that it is "reduced", as defined below.

The set W is said to be *reduced* if, for all words A and B in W, A is a subword[1] of B equivales A equals B. The assumption that W is reduced is sensible because without it the game would be either unfair or ill-defined — if A is a proper suffix of B, the winner of complete play B is not well-defined, and if A is a proper subword of B and not a proper suffix, the player who chooses B can never win. For example the set $\{a,ba,bb\}$ in example 2 is not reduced. (If the complete play is ba, it is not clear whether the winner is the player choosing a or the player who chooses ba.) The need for the assumption also appears formally in our calculations.

If A is a word in W, S_A is defined by

$$S_A \quad = \quad T^*\{A\} \cap \neg(T^* W T^+) \quad .$$

Note that $S = \langle \cup A : A{\in}W : S_A\rangle$. The language S_A is the set of complete plays that end in the word A.

As in lemma 6, the set N is defined to be the set of all proper prefixes of S :

$$N \quad = \quad pre^+.S \quad .$$

[1] A is a subword of B equivales there are words C and D such that $B = CAD$.

(It is straightforward to show that $N = \neg(T^* W T^*)$. That is, N is the set of words of which no word in W is a subword. This is the definition of N used by Guibas and Odlyzko [GO81].)

The crucial properties of S and N are as follows. If W is reduced then, for all $A \in W$,

(4) $N\{A\} \;=\; \langle \cup B : B \in W : S_B(B \,\text{⌶}\, A) \rangle$

where

(5) $B \,\text{⌶}\, A \;=\; \{E,F \;:\; \#E < \#A \wedge \#F < \#B \wedge BE = FA \;:\; E\}$.

We pronounce $B \,\text{⌶}\, A$ as B *match* A. Note that, in spite of the symbol by which it is denoted, the match operator is not symmetric. See example 4 below for instances of the match operator and equations (4) and (5).

For the proof of (4), we first note that

$$N\{A\} \;=\; \langle \cup B : B \in W : S_B(B \,\text{⌶}\, A) \rangle$$

$$\equiv \;\; \langle \forall C :: C \in N \;\equiv\; \langle \exists B : B \in W : CA \in S_B(B \,\text{⌶}\, A) \rangle \rangle \quad .$$

(This is a simple application of the definition of equality of sets, set concatenation and set union.)

Now, for all words C and all words A in W,

$\langle \exists B \;:\; B \in W \;:\; CA \in S_B(B \,\text{⌶}\, A) \rangle$

$=$ { definition of $B \,\text{⌶}\, A$ }

$\langle \exists B,E,F \;:\; B \in W \wedge \#E < \#A \wedge \#F < \#B \wedge BE = FA \;:\; CA \in S_B\{E\} \rangle$

$=$ { word calculus, $\#(XY) = \#X + \#Y$ }

$\langle \exists B,D,E,F \;:\; B \in W \wedge 1 \le \#D \le \#B \wedge A = DE \wedge B = FD \;:\; CD \in S_B \rangle$

$=$ { $S_B \subseteq T^*\{B\}$ }

$\langle \exists B,D,E \;:\; B \in W \wedge 1 \le \#D \le \#B \wedge A = DE \;:\; CD \in S_B \rangle$

$=$ { $\#B < \#D \;\wedge\; A = DE \;\wedge\; CD \in S_B$

$\Rightarrow$ { $S_B \subseteq T^*\{B\}$ }

B is a proper subword of A

$\Rightarrow$ { W is reduced, $A \in W$ and $B \in W$ }

false }

$\langle \exists B,D,E \;:\; B \in W \wedge 1 \le \#D \wedge A = DE \;:\; CD \in S_B \rangle$

$=$ { $S \;=\; \langle \cup B : B \in W : S_B \rangle$ }

$\langle \exists D,E \;:\; 1 \le \#D \wedge DE = A \;:\; CD \in S \rangle$

$=$ { ($\Rightarrow$) definition of N,
($\Leftarrow$) $A \in W$, so $CA \in T^* W$; lemma 3 }

$C \in N$.

This completes the proof of (4).

Example 4. Suppose the alphabet has two symbols h and t. Suppose the set W has three elements hh, ht and th. The set S is $\{t\}^*\{hh,ht,th\}$ and the sets S_{hh}, S_{th} and S_{ht} are, respectively, $\{t\}^*\{hh\}$, $\{t\}^*\{th\}$ and $\{t\}^*\{ht\}$; the set N is $\{\varepsilon\} \cup \{t\}^*\{h,t\}$.

The following table shows $B \bowtie A$ for each of the 9 combinations of B and A. (Rows are indexed by B and columns by A.)

$\bowtie$	hh	ht	th
hh	$\{\varepsilon,h\}$	$\{t\}$	$\emptyset$
ht	$\emptyset$	$\{\varepsilon\}$	$\{h\}$
th	$\{h\}$	$\{t\}$	$\{\varepsilon\}$

The appropriate instances of (4) are thus as follows:

$$N\{hh\} = S_{hh}\{\varepsilon,h\} \cup S_{th}\{h\}$$
$$N\{ht\} = S_{hh}\{t\} \cup S_{ht} \cup S_{th}\{t\}$$
$$N\{th\} = S_{ht}\{h\} \cup S_{th}$$

(Some simplification has been applied to these equations. So, for example, in the first equation the term $S_{ht}\emptyset$ has been omitted and, in the second equation, $S_{ht}\{\varepsilon\}$ has been simplified to S_{ht}.)

These equations are complemented by the equations:

$$N \cup S = \{\varepsilon\} \cup N\{h,t\}$$
$$S = S_{hh} \cup S_{ht} \cup S_{th}$$

The combination of the two sets of equations is the basis for calculating the probabilities of winning a game with three players who each choose the three words hh, ht and th as the eventual outcome of the game, as we discuss in the next section. □

4.2 Solov'ev's Equation and Conway's Equation

Suppose we are given a probability distribution p on the elements of the alphabet T. Suppose W is a language and S equals $PF.T^*W$. Then, for each word A in W, $h_p.S_A$ is the relative frequency that a word ending in A is a complete play of the game (theorem 1). We show how to use (4) to evaluate $h_p.S_A$ for each A. In the case that W has one element, this gives Solov'ev's equation for the expected length of a sequence of observations culminating in (the "block pattern") A; see theorem 2. In the case that W has two elements, this gives Conway's formula for the probability that each person wins in a two-person Penney-Ante game; see theorem 3.

Lemma 7. Suppose V is a function from words in W to languages. Suppose W is reduced and finite. Then $\langle \cup B : B \in W : S_B V_B \rangle$ is unambiguous.

Proof. By lemma 5, each term $S_B V_B$ is unambiguous. Also, for all words D, D', E and E', and all words B and C in W,

$$DE = D'E' \ \wedge\ D \in S_B \ \wedge\ D' \in S_C$$

$$\Rightarrow \qquad \{ \qquad S_B \cup S_C \subseteq PF.T^*W, \text{ lemma 5} \quad \}$$

$$DE = D'E' \ \wedge\ D \in S_B \ \wedge\ D' \in S_C \ \wedge\ D = D'$$

$$\Rightarrow \qquad \{ \qquad W \text{ is reduced, } S_B \subseteq T^*\{B\},\ S_C \subseteq T^*\{C\} \quad \}$$

$$D = D' \ \wedge\ E = E' \ \wedge\ B = C \quad .$$

□

Corollary 1. For all A in W,

$$h_p.N \times h_p.\{A\} \quad = \quad \langle \Sigma B : B \in W : h_p.S_B \times h_p.(B \bowtie A) \rangle \quad .$$

Also, for all A and B in W,

$$h_p.(B \bowtie A) \quad = \quad \langle \Sigma\, E,F \ :\ \#E < \#A \wedge \#F < \#B \wedge BE = FA \ :\ h_p.E \rangle \quad .$$

Proof. The expression $N\{A\}$ is obviously unambiguous. So, by lemma 1, $h_p.N\{A\}$ is the product of $h_p.N$ and $h_p.\{A\}$. Applying h_p to both sides of (4), this gives the left side of the first equation above. The right side is immediate from lemma 1 and lemma 7.

The second equation is immediate from lemma 1. (Obviously the right side of (5) is unambiguous.) □

We are now in a position to formulate the theorems attributed to Sovol'ev and Conway. In the statement of the theorems, the binary operator ":" is defined on pairs of words by, for all C and D,

$$C : D \quad = \quad \frac{h_p.(C \bowtie D)}{h_p.D} \quad .$$

This operator generalises the one with the same name in [GKP94]. See theorem 3, below, for further explanation of the generalisation.

Note that $C : D$ has no interpretation as a probability. Indeed, $C : C$ is typically greater than 1; it is the expected length of the first occurrence of block pattern C, as shown in the next theorem.

Theorem 2 (Sovol'ev's formula). Suppose $S = PF.T^*\{A\}$. Then

$$e_p.S \ = \ A : A \quad .$$

Proof. We have:

$$e_p.S$$

$$= \qquad \{ \qquad \text{lemma 2} \quad \}$$

$$h_p.N$$

$$
\begin{array}{ll}
= & \{ \quad \text{corollary 1 with } W := \{A\} \text{ (using } h_p.S = 1 \\
 & \quad\ \ \text{and one-point rule to simplify the summation)} \quad \} \\
 & \dfrac{h_p.(A \between A)}{h_p.A} \\
= & \{ \quad \text{definition} \quad \} \\
 & A : A \quad .
\end{array}
$$

□

Example 5. Suppose the alphabet has two symbols h and t (for heads and tails). Suppose k is a natural number and A is the word $h^k t$ and B is the word h^{k+1}. Then

$$
\begin{aligned}
&A \between A = \{\varepsilon\} \\
&B \between B = \left\{ j : 0 \leq j \leq k : h^j \right\}
\end{aligned}
$$

Suppose further that $p.h = q$ and $p.t = r$, where $q + r = 1$. It follows that

$$
\begin{aligned}
&h_p.(A \between A) = 1 \\
&h_p.(B \between B) = \frac{1-q^{k+1}}{1-q}
\end{aligned}
$$

Since $h_p.A$ is $q^k \times r$ and $h_p.B$ is q^{k+1},

$$
A : A \quad = \quad \frac{1}{q^k \times r}
$$

and

$$
B : B \quad = \quad \frac{1-q^{k+1}}{(1-q) \times q^{k+1}} \quad .
$$

It follows from theorem 2 that

$$
\frac{e_p.(PF.T^*\{B\})}{e_p.(PF.T^*\{A\})} \quad = \quad \frac{1-q^{k+1}}{q} \quad .
$$

The expected number of coin-tosses before h^{k+1} is encountered is thus approximately $\frac{1}{q}$ times greater than the expected number of coin-tosses before $h^k t$ is encountered. In the words of [GKP94]: “patterns with no self-overlaps occur sooner than overlapping patterns do!” □

Theorem 3. Suppose $W = \{A,B\}$. Suppose W is reduced. Then

$$
\frac{h_p.S_A}{h_p.S_B} \quad = \quad \frac{B : B \ - \ B : A}{A : A \ - \ A : B} \quad .
$$

Proof. Straightforward instantiation of corollary 1. □

Corollary 2 (Conway’s formula). If A and B have equal length, and p assigns equal values to each element of T then

$$
\frac{h_p.S_A}{h_p.S_B} \quad = \quad \frac{h_p.(B \between B) - h_p.(B \between A)}{h_p.(A \between A) - h_p.(A \between B)} \quad .
$$

(The latter is equivalent to the formula attributed to John Horton Conway in [GKP94] for the odds of A winning against B in a Penney-Ante game where a coin is tossed and the probability of a head or tail occurring is $\frac{1}{2}$. In Conway's formula, the notation $B : A$ is used for $h_p.(B \between A) \times 2^{\#A\,-\,1}$. It is not clear from the published literature whether or not Conway derived the general formula given in theorem 3.) $\square$

The examples below test the use of theorem 3 on cases where it is easy to predict the relative frequency of occurrence of words in S_A and in S_B.

Example 6. Suppose the alphabet has two symbols h and t (for heads and tails). Suppose k is a natural number and A is the word $h^k t$ and B is the word h^{k+1}. Suppose further that $p.h = q$ and $p.t = r$, where $q+r = 1$. A simple argument establishes that the relative frequency of A compared to B in a Penney-Ante game is $\frac{r}{q}$. We can check that this is predicted by theorem 3 as follows. We first calculate that

$$A \between A = \{\varepsilon\}$$
$$A \between B = \emptyset$$
$$B \between A = \left\{ j : 0 \leq j < k : h^j t \right\} \quad .$$

Then.

$$h_p.(A \between B) = 0$$
$$h_p.(B \between A) = \frac{(q^k - 1) \times r}{q - 1} \quad .$$

Combining these with the calculations in example 5 and substituting in theorem 3 (top formula), we get, for example,

$$B : A \quad = \quad \frac{(q^k - 1) \times r}{(q - 1) \times (q^k \times r)}$$

Hence, applying theorem 3 (top formula) (and a lot of simplification!), we get

$$\frac{h_p.S_A}{h_p.S_B} \quad = \quad \frac{r}{q}$$

as expected. $\square$

Example 7. Suppose the alphabet has two symbols a and b . Suppose the set W has two elements, A and B, equal to a and bb, respectively. Suppose $p.a = q$ and $p.b = r$, where $q+r = 1$. As observed earlier, $PF.(\{a,b\}^* \{a,bb\}) = \{a,ba,bb\}$. If q and r model the relative frequency of occurrences of a and b, respectively, it is clear that the relative frequency of S_A, which equals $\{a,ba\}$, is $q + r \times q$ and the relative frequency of S_B, which equals $\{bb\}$, is r^2. Let us check that this is what is predicted by theorem 3.

We calculate that $A \between A$ equals $\{\varepsilon\}$, $B \between B$ equals $\{\varepsilon,b\}$ and both $A \between B$ and $B \between A$ equal the empty set. The h_p values are now easily calculated. Applying theorem 3, we get

$$\frac{h_p.S_A}{h_p.S_B} \quad = \quad \frac{(1+r) \times q - 0}{1 \times r^2 - 0}$$

which simplifies to $\frac{(1+r)\times q}{r^2}$. Since $q+r=1$ and $h_p.S_A + h_p.S_B = 1$, it follows that $h_p.S_A$ equals $1-r^2$ and $h_p.S_B$ equals r^2. □

The next example is of a game with an infinite number of players.

Example 8. Suppose the alphabet has three symbols a, b and c. Suppose $W=\{a\}\{b\}^*\{c\}$. (So each word in W is of the form ab^kc for some k, $0\leq k$. Note that W is not finite but it is reduced.) It is easy to verify that $ab^kc \between ab^kc = \{\varepsilon\}$ and, when $j\neq k$, $ab^jc \between ab^kc = \emptyset$. Thus:

$$N\{ab^kc\} = S_{ab^kc}$$
$$N\cup S = \{\varepsilon\} \cup N\{a,b,c\}$$
$$S = \langle \cup k : 0\leq k : S_{ab^kc}\rangle$$

It is immediate from these equations that $S=N\{a\}\{b\}^*\{c\}$. However, it is difficult to "solve" them in the sense of determining a regular expression defining N. Indeed, it is not even clear that there is a unique solution for N; see section 5.

Suppose now that $p.a=q$, $p.b=r$ and $p.c=s$, where $q+r+s=1$. Then, exploiting the above equation for S, we obtain:

$$h_p.N \times q \times r^k \times s = h_p.S_{ab^kc}$$
$$h_p.N + h_p.N \times q \times r^* \times s = 1 + h_p.N \times (q+r+s)$$

(where we write r^* for $\frac{1}{1-r}$). It follows that $h_p.N = \frac{1-r}{q\times s}$ and $h_p.S_{ab^kc} = (1-r)\times r^k$. So the expected length of a game is $\frac{1-r}{q\times s}$ (which equals $\frac{1}{q}+\frac{1}{s}$) and the probability that the recognised pattern is ab^kc is $(1-r)\times r^k$. □

5 Uniqueness

In the case that a Penney-Ante game has just two players, theorem 3 together with the equation $h_p.S_A + h_p.S_B = 1$ enables one to calculate both $h_p.S_A$ and $h_p.S_B$. In other words, it is possible to determine the probability that each of the players wins. This raises the question whether or not the system of equations (4) together with the equations

(6) $N\cup S = \{\varepsilon\} \cup NT$

(7) $S = \langle \cup A : A\in W : S_A\rangle$

(cf. definition 1(b)) viewed as equations in the unknowns N, S and S_A (for each $A\in W$), has a unique solution independently of the size of W.

The answer is no. A very simple example demonstrates this fact. Suppose $T=\{a\}=W$. Then, since $a \between a = \{\varepsilon\}$, we get just two equations (equation (7) is trivial):

$$N\cup S = \{\varepsilon\} \cup N\{a\}$$
$$N\{a\} = S$$

As is easily checked, one solution to these equations is $N = \{\varepsilon\}$ and $S = \{a\}$. (This is the desired solution.) A second solution is $N = \{a\}^*$ and $S = \{a\}^+$.

Note that, although these two equations do not have a unique solution, we can use them to determine $h_p.N$ and $h_p.S$. Specifically, since inevitably $h_p.a = 1$, we get the equations:

$$\begin{aligned} h_p.N + h_p.S &= 1 + h_p.N \\ h_p.N &= h_p.S \end{aligned}$$

Unsurprisingly, the expected length of a complete play is 1. (Apply lemma 2.) Note, however, that $h_p.\{a\}^*$ is undefined. (Recall that $\{a\}^*$ is a solution for N.)

It was a surprise to us that the equations in languages do not have a unique solution since Guibas and Odlyzko [GO81] claim that the derived equations in generating functions do have unique solutions. Their argument is based on the fact that, when $\{A,B\}$ is reduced, $\varepsilon \in A \between B \equiv A = B$ for all words A and B. We have as yet been unable to use this fact to show that the equations in h_p values have unique solutions and the question remains open. (Whether or not the equations have unique solutions when we add the equation $N \cap S = \emptyset$ is irrelevant since this property is not reflected in the generating functions.)

6 Conclusion

The purpose of this paper has been to fully understand the reasoning behind the derivation of Conway's formula for solving the Penney-Ante game. Our understanding has improved considerably. In [GKP94] equations are formulated for $h_p.N$ and $h_p.S_A$ (for each A) —albeit using a different notation— and it is claimed that $h_p.S_A$ is the probability that the event A occurs. However, this claim does not appear to be properly justified, as evidenced by the fact that no claim is made about the meaning of $h_p.N$. Similarly, we find the arguments given by Guibas and Odlyzko [GO81] somewhat difficult to understand because they involve events that can never occur in a first-past-the-post game: Guibas and Odlyzko appear to ascribe meaning to $h_p.S_B \times h_p.(B \between A)$ as a probability, whereas, for $B \neq A$, the frequency that a word in $S_B(B \between A)$ ever occurs is 0 — the game would be terminated before such an event occurs. Here we have made clear that h_p is a probability distribution on the sample space $PF.T^*W$. In the derivation of theorem 3, the function is also applied to languages not in this sample space, in which case it is typically not a probability distribution.

We make no use whatsoever of generating functions. Generating functions enable one to derive properties related to word length; our derivations show that word length is irrelevant to deriving Conway's formula (and also Solov'ev's formula). Even in the case of calculating the expected length of a complete play of a game, where word length is part of the definition, lemma 2 is all that is needed.

On the other hand, we have been unable to calculate a formula for the standard deviation of the length of complete games (or for other higher-order cumulants).

The conclusion would appear to be that the versatility of generating functions is best demonstrated by their use in determining higher-order cumulants.

The fact that we have been unable to establish uniqueness of the system of equations in languages whilst the system of equations in h_p values does have a unique solution (since the equations in generating functions have a unique solution) requires further investigation.

Acknowledgement. I am very grateful to the anonymous referees all five of whom gave very detailed and supportive comments on the submitted paper, including correcting some errors. I hope I have done justice to their efforts.

References

[Bac86] Backhouse, R.C.: Program Construction and Verification. Prentice-Hall International (1986)

[Bac03] Backhouse, R.: Program Construction. Calculating Implementations From Specifications. John Wiley & Sons, Ltd. (2003)

[FS09] Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)

[GKP94] Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: a Foundation for Computer Science, 2nd edn. Addison-Wesley Publishing Company (1994)

[GO81] Guibas, L.J., Odlyzko, A.M.: String overlaps, pattern matching and nontransitive games. Journal of Combinatorial Theory, Series A30, 183–208 (1981)

[GS93] Gries, D., Schneider, F.B.: A Logical Approach to Discrete Math. Springer (1993)

[Pen74] Penney, W.: Problem 95: Penney-Ante. Journal of Recreational Mathematics, 321 (1974)

[Sol66] Solov'ev, A.D.: A combinatorial identity and its application to the problem concerning the first occurrence of a rare event. Theory of Probability and its Applications 11, 276–282 (1966)

Reverse Exchange for Concurrency and Local Reasoning

Han-Hing Dang and Bernhard Möller

Institut für Informatik,
Universität Augsburg, D-86159 Augsburg, Germany
{h.dang,moeller}@informatik.uni-augsburg.de

Abstract. Recent research has pointed out the importance of the inequational exchange law $(P*Q)\,;(R*S) \leq (P\,;R)*(Q\,;S)$ for concurrent processes. In particular, it has been shown that this law is equivalent to validity of the concurrency rule for Hoare triples. Unfortunately, the law does not hold in the relationally based setting of algebraic separation logic. However, we show that under mild conditions the reverse inequation $(P\,;R) * (Q\,;S) \leq (P * Q)\,;(R * S)$ still holds there. Separating conjunction $*$ in that calculus can be interpreted as true concurrency on disjointly accessed resources. From the reverse exchange law we derive slightly restricted but still reasonably useful variants of the concurrency rule. Moreover, using a corresponding definition of locality, we obtain also a variant of the frame rule. By this, the relational setting can also be applied for modular and concurrency reasoning. Finally, we present several variations of the approach to further interpret the results.

Keywords: True concurrency, relational semantics, Hoare logic, concurrent separation logic, locality, frame rule.

1 Introduction

Algebraic techniques nowadays have found widespread application, especially in the area of program logics. In particular, *separation logic* [14] has proved to be very useful in the domain of modular and concurrency reasoning [1,12] — although originally it was only developed to facilitate reasoning about shared mutable data structures. For this logic there are already different abstract approaches that capture corresponding calculi [2,4]. Recent investigations on these topics resulted in a general algebraic structure called *Concurrent Kleene Algebra* [8]. A central concept of that algebra is that it allows easy soundness proofs of important rules like the concurrency and frame rules used in logics for concurrency and modular reasoning.

The *concurrency* and *frame* rules have the form

$$\frac{\{P_1\}\,Q_1\,\{R_1\} \qquad \{P_2\}\,Q_2\,\{R_2\}}{\{P_1 * P_2\}\,Q_1 * Q_2\,\{R_1 * R_2\}}\,(\mathit{conc}) \qquad \frac{\{P\}\,Q\,\{R\}}{\{P * S\}\,Q\,\{R * S\}}\,(\mathit{frame})\ .$$

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 177–197, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Here Q and Q_i denote programs while all other letters denote assertions. Now the *separating conjunction* $*$, as it is called in the literature, is used in the conclusion of these rules to ensure disjointness of states or resources characterised by assertions. When used on programs, such as the Q_i above, separating conjunction can be interpreted as concurrent execution of programs.

Interestingly, it has been shown in [7] that validity of the *exchange law*

$$(P_1 * P_2) \,;\, (Q_1 * Q_2) \;\leq\; (P_1 \,;\, Q_1) * (P_2 \,;\, Q_2) \;,$$

for programs P_i and Q_i and validity of the concurrency rule are equivalent. An analogous connection holds between the *small exchange law*

$$(P_1 * P_2) \,;\, Q_1 \leq (P_1 \,;\, Q_1) * P_2$$

and the frame rule. In these laws, semicolon denotes sequential composition, while $\leq$ denotes a partial ordering expressing refinement. The exchange laws can be seen as an abstract characterisation of the interplay between sequential and concurrent composition. Each of them expresses that the program on the right-hand side has fewer sequential dependences than the one on the left-hand side.

Several models for algebraic structures obeying those laws exist; details may be found in [8,7]. However, they either do not model concurrency adequately enough or fail to satisfy other important laws in connection with nondeterministic choice. The purpose of the present paper is to investigate an extension of the relational model of separation logic presented in [4] by a generalised separating conjunction. As a relational structure it copes well with nondeterminacy; moreover, it allows the re-use of a large and well studied body of algebraic laws in connection with assertion logic. Surprisingly, it turns out that, although the model satisfies neither of the mentioned exchange laws, it validates an exchange law with the reversed refinement order. Moreover, this entails variants of the concurrency and frame rules with similarly simple soundness proofs as in the original Concurrent Kleene Algebra approach. Also, we establish an analogous equivalence between the concurrency rule and the reverse exchange law as in [7]. Hence, the relational calculus can be applied in reasoning about programs involving true concurrency and modularity. To underpin this further, we also study a number of variations of our main relational model and discuss their adequacy and usefulness.

2 Basic Definitions and Properties

We start by repeating some basic definitions from [4] and some direct consequences. Summarised, the central concept of this paper is a relational structure enriched by an operator that ensures disjointness of program states or executions. Notationally, we follow [4,7].

Definition 2.1. A *separation algebra* is a partial commutative monoid $(\Sigma, \bullet, u)$; the elements of Σ are called *states* and denoted by $\sigma, \tau, \ldots$. The operator $\bullet$ denotes state combination and the *empty state* u is its unit. A partial commutative

monoid is given by a partial binary operation satisfying the unity, commutativity and associativity laws w.r.t. the equality that holds for two terms iff both are defined and equal or both are undefined. The induced *combinability* or *disjointness* relation $\#$ is defined by

$$\sigma_0 \# \sigma_1 \;\Leftrightarrow_{df}\; \sigma_0 \bullet \sigma_1 \text{ is defined} .$$

As a concrete example one can instantiate the states to heaps. For this we set $\Sigma =_{df} \mathbb{N} \rightsquigarrow \mathbb{N}$, i.e., the set of partial functions from naturals to naturals. Moreover $\bullet =_{df} \cup$ and $u =_{df} \emptyset$, the empty heap. A possible combinability relation for this domain would be $h_0 \# h_1 \Leftrightarrow_{df} dom(h_0) \cap dom(h_1) = \emptyset$ for heaps h_0, h_1. More concrete examples can be found in [2].

Definition 2.2. We assume a separation algebra $(\Sigma, \bullet, u)$. A *command* is a relation $P \subseteq \Sigma \times \Sigma$ between states. Relational composition is denoted by $;$. The command skip is the identity relation between states. A *test* is a subidentity, i.e., a command P with $P \subseteq \mathsf{skip}$. In the remainder we will denote tests by lower case letters $p, q, \ldots$. A particular test that characterises the empty state u is provided by $\mathsf{emp} =_{df} \{(u, u)\}$. Moreover, the domain of a command P, represented as a test, will also be denoted by $dom(P)$. It is characterised by the universal property

$$dom(P) \subseteq q \;\Leftrightarrow\; P \subseteq q \,;P .$$

In particular, $P \subseteq dom(P) \,; P$ and hence $P = dom(P) \,; P$.

Note that tests form a Boolean algebra with skip as its greatest and $\emptyset$ as its least element. Moreover, on tests $\cup$ coincides with join and $;$ with meet. In particular, tests are idempotent and commute under composition, i.e., $p \,; p = p$ and $p \,; q = q \,; p$.

Next we give some definitions to introduce separation relationally. Separating conjunction of commands can be interpreted as their parallel execution on disjoint portions of the state or, in the special case of tests, by asserting disjointness of certain resources.

Definition 2.3. We will frequently work with pairs of commands. Union, inclusion and composition of such pairs are defined componentwise. The *Cartesian product* $P \times Q$ of commands P, Q is given by

$$(\sigma_1, \sigma_2)\ (P \times Q)\ (\tau_1, \tau_2) \Leftrightarrow_{df} \sigma_1\ P\ \tau_1 \wedge \sigma_2\ Q\ \tau_2 .$$

We assume that $;$ binds tighter than $\times$. It is clear that $\mathsf{skip} \times \mathsf{skip}$ is the identity of composition. Note that $\times$ and $;$ satisfy an *equational* exchange law:

$$P \,; Q \;\times\; R \,; S \;=\; (P \times R) \;;\; (Q \times S) . \tag{1}$$

Definition 2.4. *Tests* in the set of product relations are again subidentities; as before they are idempotent and commute under $;$. The Cartesian product of tests is a test again. However, there are other tests, such as the *combinability check* $\#$ [4], on pairs of states:

$$(\sigma_1, \sigma_2)\ \#\ (\tau_1, \tau_2) \Leftrightarrow_{df} \sigma_1 \# \sigma_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 .$$

Definition 2.5. We define *split* $\lhd$ and its converse *join* $\rhd$ as in [4] by

$$\sigma \lhd (\sigma_1, \sigma_2) \;\Leftrightarrow_{df}\; (\sigma_1, \sigma_2) \rhd \sigma \;\Leftrightarrow_{df}\; \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \;.$$

Lemma 2.6. *We have* $\# = \rhd \,;\, \lhd \cap \mathsf{skip}$ *and hence* $\# \subseteq \rhd \,;\, \lhd$. *Moreover* $\# \,;\, \rhd = \rhd$ *and symmetrically* $\lhd \,;\, \# = \lhd$.

One might conjecture $\mathsf{skip} \times \mathsf{skip} \subseteq \rhd \,;\, \lhd$ at first. However, this is not true, since the left-hand side of the inequation also considers incombinable pairs of states which are not included in the right-hand side according to Lemma 2.6. We will see in the next section that this fact requires us to impose an additional compatibility condition on commands for proving soundness of the reverse exchange law, i.e., the exchange law with the inequation reversed.

Definition 2.7. Generalising [4], we define the *parallel composition (separating conjunction)* of commands as $P * Q \;=_{df}\; \lhd \,;\, (P \times Q) \,;\, \rhd$.

By this definition, a relation $\sigma\,(P{*}Q)\,\tau$ holds iff σ can be split as $\sigma = \sigma_1 \bullet \sigma_2$ with disjoint parts σ_1, σ_2 on which P and Q can act and produce results τ_1, τ_2 that are again disjoint and combine to $\tau = \tau_1 \bullet \tau_2$. Hence $P * Q$ may be viewed as a program that runs P and Q in a truly concurrent fashion as indivisible actions, at least conceptually. An actual implementation may still do this in an interleaved or even truly concurrent fashion, as long as non-interference is guaranteed. We note that for tests p, q the command $p{*}q$ is a test again. Moreover, $*$ is associative and commutative and emp is its unit. Finally, there is the following interplay between $*$ and the domain operator.

Lemma 2.8. *For commands* P, Q *we have* $dom(P * Q) \subseteq dom(P) * dom(Q)$.

The proof can be found in the Appendix.

3 Compatibility and the Reverse Exchange Law

According to the general results in [7], soundness of the concurrency rule in the relational setting would follow immediately if the exchange law

$$(P * Q) \,;\, (R * S) \;\subseteq\; (P \,;\, R) * (Q \,;\, S)$$

with relational inclusion $\subseteq$ as the refinement order were to hold there.

However, as also shown in [7], we have

Lemma 3.1. *The exchange law implies* $\mathsf{skip} \subseteq \mathsf{emp}$.

On the other hand, by definition $\mathsf{emp} \subseteq \mathsf{skip}$, so that by antisymmetry skip and emp would be equal, a contradiction.

Therefore the exchange law is not valid in the relational setting. Instead, and surprisingly, we were only able to show soundness of a restricted variant of the exchange law with the reversed inclusion order. The proof uses a restriction on pairs (P, Q) of commands: when P and Q start from combinable pairs of input states they produce combinable pairs of output states, or the other way around. This is formalised as follows.

Definition 3.2. Commands P and Q are *forward compatible* iff

$$\# \,;\, (P \times Q) \;\subseteq\; (P \times Q) \,;\, \# \;.$$

Symmetrically P and Q are *backward compatible* iff $(P{\times}Q);\# \subseteq \#;(P{\times}Q)$. Two commands are called *compatible* iff they are forward and backward compatible, i.e., $\# \,;\, (P \times Q) = (P \times Q) \,;\, \#$.

Again for a more concrete example of such commands we recapitulate our instantiation of states to heaps described in Section 2. Intuitively two compatible commands would work on disjoint portions of a heap, e.g. by only altering disjoint ranges of heap cells. Hence they ensure disjointness before and after their execution. In the following we list a few consequences of Definition 3.2.

Lemma 3.3. *All test commands are compatible with each other.*

Proof. For test commands p, q the relation $p{\times}q$ is a test in the algebra of relations on pairs. Since $\#$ is a test there, too, they commute, which means forward and backward compatibility of p and q. □

Since the combinability check $\#$ is a test on pairs of commands, it induces some useful closure properties.

Corollary 3.4. *If P, Q are forward compatible and $R \subseteq P$ then also R, Q are forward compatible. This result also holds for backward compatibility, hence compatibility is downward closed, too.*

Proof. We assume ; binds tighter than $\cap$. Now we show the following more general result: Let C, D, E be relations on pairs of states such that C is a test. If C is an invariant of D, i.e., $C\,;D \subseteq D\,;C$, and $E \subseteq D$ then C is also an invariant of E. For this we calculate

$$C \,;\, E \;=\; C \,;\, (D \,\cap\, E) \;=\; C \,;\, D \,\cap\, C \,;\, E \;\subseteq\; D \,;\, C \,\cap\, C \,;\, E \;=\; D \,\cap\, C \,;\, E \,;\, C \;=\; C \,;\, E \,;\, C \;\subseteq\; E \,;\, C \;.$$

The last but one step follows since C is a test. A proof can e.g. be found in [10]. Now the main claim follows by setting $C = \#$, $D = P \times Q$ and $E = R \times Q$. □

We note that this proof extends to arbitrary test semirings.

Corollary 3.5. *Let P, Q and R, S be forward compatible. Then also $P\,;R$ and $Q\,;S$ are forward compatible. Again the same holds for backward compatibility.*

Proof

$$\# \,;\, (P \,;\, R \times Q \,;\, S) = \# \,;\, (P \times Q) \,;\, (R \times S) \subseteq (P \times Q) \,;\, \# \,;\, (R \times S) \subseteq (P \times Q) \,;\, (R \times S) \,;\, \# = (P \,;\, R \times Q \,;\, S) \,;\, \# \;.$$

□

Now we are ready for the central result mentioned at the beginning of this section. For forward or backward compatible commands we are able to prove soundness of a variant of the reverse exchange law using the inclusion order. Note that validity of the exchange law in [7] is proved for arbitrary predicate transformers. In the next section we will see that specialising validity of the reversed law to compatible commands does not impose any restrictions on our treatment.

Lemma 3.6 (Reverse Exchange). *If P, Q are forward compatible or R, S are backward compatible then*

$$(P\,;R) * (Q\,;S) \;\subseteq\; (P * Q)\,;(R * S)\ .$$

In particular, if P, R or Q, S are tests the inequation holds.

Proof. We assume that P and Q are forward compatible.

$$\begin{array}{ll}
 & (P\,;R) * (Q\,;S) \\
= & \quad \{\!\mid \text{definition of } * \mid\!\} \\
 & \lhd\,;(P\,;R \times Q\,;S)\,;\rhd \\
= & \quad \{\!\mid\ ;/\times \text{ exchange (1)} \mid\!\} \\
 & \lhd\,;(P \times Q)\,;(R \times S)\,;\rhd \\
= & \quad \{\!\mid \text{Lemma 2.6} \mid\!\} \\
 & \lhd\,;\#\,;(P \times Q)\,;(R \times S)\,;\rhd \\
\subseteq & \quad \{\!\mid \text{forward compatibility} \mid\!\} \\
 & \lhd\,;(P \times Q)\,;\#\,;(R \times S)\,;\rhd \\
\subseteq & \quad \{\!\mid \text{Lemma 2.6} \mid\!\} \\
 & \lhd\,;(P \times Q)\,;\rhd\,;\lhd\,;(R \times S)\,;\rhd \\
= & \quad \{\!\mid \text{definition of } * \mid\!\} \\
 & (P * Q)\,;(R * S)\ .
\end{array}$$

The proof for backward compatibility and R, S is symmetric. □

The reverse exchange law expresses an increase in granularity: while in the left-hand side program $P\,;R$ and $Q\,;S$ are treated as indivisible, they are split in the right-hand side program, at the expense of a "global" synchronisation point marked by the semicolon (which is the reason for the compatibility requirement).

4 Hoare Triples and the Concurrency Rule

To prepare our variants of the concurrency rule we now define Hoare triples in our setting.

Definition 4.1. For general commands P, Q, R, the *general Hoare triple* [7] is defined as

$$P\,\{Q\}\,R \;\Leftrightarrow_{df}\; P\,;Q \subseteq R\ .$$

For tests p, r and arbitrary command Q the *standard* Hoare triple [9] $\{p\}\, Q\, \{r\}$ is defined by

$$\{p\}\, Q\, \{r\} \;\Leftrightarrow_{df}\; p\,;Q \subseteq Q\,;r\;.$$

General Hoare triples also admit programs as assertions, in contrast to the standard ones that only allow tests to denote pre- and postconditions. As shown in [4], we have the relationship

$$\{p\}\, Q\, \{r\} \;\Leftrightarrow\; (U\,; p)\, \boldsymbol{\{} Q \boldsymbol{\}}\, (U\,; r)$$

where U denotes the universal relation. Hence our results for standard Hoare triples can be immediately translated into ones for general triples. The composition $U\,;p$ maps a test p to a command that makes no assumption about its starting state. Intuitively, starting from an arbitrary state that command will end up in one satisfying p. Trivially, a symmetrical command $p\,;U$ makes no restriction on the ending state or codomain.

Next we turn to the definition of properties and conditions that will allow us to prove variants of the concurrency rule for standard Hoare triples using the reverse exchange law.

The following observation is trivial, but useful for our first variant of the concurrency rule.

Corollary 4.2. $\{p\}\, Q\, \{r\} \Leftrightarrow \{p\}\, p\,;Q\, \{r\}$.

Proof. By idempotence of test p,

$$\{p\}\, Q\, \{r\} \;\Leftrightarrow\; p\,;Q \subseteq Q\,;r \;\Leftrightarrow\; p\,;p\,;Q \subseteq Q\,;r \;\Leftrightarrow\; \{p\}\, p\,;Q\, \{r\}\;.$$

□

The command $p;Q$ can be viewed as asserting the precondition p before executing Q.

The condition we need for our first variant of the concurrency rule is that the commands Q_i enforce the preconditions p_i in that all their starting states satisfy the respective p_i. Algebraically this is expressed by the formula $Q_i \subseteq p_i\,;Q_i$, which is equivalent to $Q_i = p_i\,;Q_i$ and to $dom(Q_i) \subseteq p_i$. This restriction is not essential: by Cor. 4.2 and the idempotence of tests we can always replace Q_i by $Q_i' =_{df} p_i\,;Q_i$ to achieve this.

Lemma 4.3 (Concurrency Rule I)

$$\frac{\{p_1\}\, Q_1\, \{r_1\} \quad \{p_2\}\, Q_2\, \{r_2\} \quad dom(Q_1) \subseteq p_1 \quad dom(Q_2) \subseteq p_2}{\{p_1 * p_2\}\, Q_1 * Q_2\, \{r_1 * r_2\}}$$

Proof. $(p_1 * p_2)\,;(Q_1 * Q_2)$

$\subseteq$ $\quad \{\!\![\; p_1 * p_2$ a test, hence a subidentity $]\!\!\}$

$Q_1 * Q_2$

$$
\begin{array}{ll}
\subseteq & \{\![\ Q_i \subseteq p_i \,;\, Q_i\ ]\!\} \\
& (p_1 \,;\, Q_1) * (p_2 \,;\, Q_2) \\
\subseteq & \{\![\ \text{by } \{p_i\}\, Q_i\, \{r_i\}\ ]\!\} \\
& (Q_1 \,;\, r_1) * (Q_2 \,;\, r_2) \\
\subseteq & \{\![\ \text{reverse exchange law (Lemma 3.6), since } r_1 \text{ and } r_2 \text{ are tests} \\
& \quad \text{and hence compatible by Lemma 3.3}\ ]\!\} \\
& (Q_1 * Q_2) \,;\, (r_1 * r_2)\ .
\end{array}
$$

□

Note that compatibility of the commands Q_i is not needed.

The proof might suggest that the preconditions p_i do not really matter, since they are discarded in the first step. However, they are re-introduced in the next step and hence indeed *do* matter.

A brief discussion of the relevance and use of this rule can be found at the end of the next section.

To round off this section, we prove the following result which, together with Lemma 4.3, provides the analogue of the equivalence between the full exchange law and the concurrency rule shown in [7].

Lemma 4.4. *Validity of Concurrency Rule I implies a special case of the reverse exchange law: for arbitrary commands P_i and tests r_i,*

$$(P_1 \,;\, r_1) * (P_2 \,;\, r_2) \subseteq (P_1 * P_2) \,;\, (r_1 * r_2)\ .$$

Proof. In Concurrency Rule I we set $Q_i = P_i \,;\, r_i$ and $p_i = dom(Q_i)$. By this the premise of the rule becomes valid since

$$\{dom(Q_i)\}\, Q_i\, \{r_i\} \;\Leftrightarrow\; dom(Q_i) \,;\, Q_i \subseteq Q_i \,;\, r_i \;\Leftrightarrow\; Q_i \subseteq Q_i \,;\, r_i$$

and $Q_i \,;\, r_i = (P_i \,;\, r_i) \,;\, r_i = P_i \,;\, r_i = Q_i$. Hence, by the conclusion of the rule we have

$$(dom(Q_1) * dom(Q_1)) \,;\, (Q_1 * Q_2) \subseteq (Q_1 * Q_2) \,;\, (r_1 * r_2)\ . \qquad (\dagger)$$

Now we calculate:

$$
\begin{array}{ll}
& (P_1 \,;\, r_1) * (P_2 \,;\, r_2) \\
= & \{\![\ \text{definitions of } Q_i\ ]\!\} \\
& Q_1 * Q_2 \\
= & \{\![\ \text{property of domain}\ ]\!\} \\
& dom(Q_1 * Q_2) \,;\, (Q_1 * Q_2) \\
\subseteq & \{\![\ \text{by Lemma 2.8}\ ]\!\} \\
& (dom(Q_1) * dom(Q_2)) \,;\, (Q_1 * Q_2) \\
\subseteq & \{\![\ \text{by } (\dagger)\ ]\!\} \\
& (Q_1 * Q_2) \,;\, (r_1 * r_2) \\
= & \{\![\ \text{definitions of } Q_i\ ]\!\} \\
& ((P_1 \,;\, r_1) * (P_2 \,;\, r_2)) \,;\, (r_1 * r_2) \\
\subseteq & \{\![\ \text{by } r_i \subseteq \mathsf{skip}\ ]\!\} \\
& (P_1 * P_2) \,;\, (r_1 * r_2)\ .
\end{array}
$$

□

We conclude by showing that the symmetric special case already follows without assuming reverse exchange or Concurrency Rule I or even mentioning the notion of compatibility.

Lemma 4.5. *For arbitrary commands Q_i and tests p_i,*

$$(p_1 ; Q_1) * (p_2 ; Q_2) \subseteq (p_1 * p_2) ; (Q_1 * Q_2) .$$

Proof. We calculate:

$$\begin{array}{ll} & (p_1 ; Q_1) * (p_2 ; Q_2) \\ = & \quad \{\!\![\text{ property of domain }]\!\!\} \\ & dom((p_1 ; Q_1) * (p_2 ; Q_2)) ; ((p_1 ; Q_1) * (p_2 ; Q_2)) \\ \subseteq & \quad \{\!\![\text{ by Lemma 2.8 }]\!\!\} \\ & (dom(p_1 ; Q_1) * dom(p_2 ; Q_2)) ; ((p_1 ; Q_1) * (p_2 ; Q_2)) \\ = & \quad \{\!\![\text{ property of domain }]\!\!\} \\ & ((p_1 ; dom(Q_1)) * (p_2 ; dom(Q_2))) ; ((p_1 ; Q_1) * (p_2 ; Q_2)) \\ \subseteq & \quad \{\!\![\text{ by } dom(Q_i) \subseteq \mathsf{skip} \text{ and } p_i \subseteq \mathsf{skip}]\!\!\} \\ & (p_1 * p_2) ; (Q_1 * Q_2) . \end{array}$$

□

Since Lemma 2.8 holds analogously for the codomain operator, this proof could also be adapted to a direct proof of the property in Lemma 4.4.

Finally, the special case of reverse exchange mentioned in Lemma 4.5 in turn implies Lemma 2.8:

$$\begin{array}{ll} & dom(P * Q) \subseteq dom(P) * dom(Q) \\ \Leftrightarrow & \quad \{\!\![\text{ universal characterisation of domain }]\!\!\} \\ & P * Q \subseteq (dom(P) * dom(Q)) ; (P * Q) \\ \Leftarrow & \quad \{\!\![\text{ special case of reverse exchange }]\!\!\} \\ & P * Q \subseteq (dom(P) ; P) * (dom(Q) ; Q) \\ \Leftrightarrow & \quad \{\!\![\text{ property of domain }]\!\!\} \\ & P * Q \subseteq P * Q \\ \Leftrightarrow & \quad \{\!\![\text{ reflexivity of } \subseteq]\!\!\} \\ & \mathsf{TRUE} . \end{array}$$

5 Another Concurrency Rule

We now present a second variant of the concurrency rule. Its main idea is inspired by a more special property given in [4], which will also figure again in the next section.

Definition 5.1. Two commands Q_1, Q_2 have the *concurrency property* iff

$$(dom(Q_1) \times dom(Q_2)) ; \rhd ; Q_1 * Q_2 \subseteq (Q_1 \times Q_2) ; \rhd . \tag{2}$$

This property is "angelic" in the sense that whenever two combinable states σ_1 and σ_2 provide enough resource for the execution of the programs Q_i then each Q_i will be able to acquire its needed resource from the joined state $\sigma_1 \bullet \sigma_2$.

To see that this is not always possible, we present a concrete example again in the heap model (mentioned in Section 2) with two commands that do not satisfy the concurrency property. Consider

$$Q_1 =_{df} ([1] := 1) \cup ([2] =: 1) \quad \text{and} \quad Q_2 =_{df} ([1] := 2) \cup ([2] := 2)$$

where $[x] := y$ represents a command that changes the content of the heap cell x to y and $\cup$ denotes non-deterministic choice. Clearly, the commands show interference with each other since both may access the same heap locations.

To see that Q_1 and Q_2 do not satisfy the concurrency property, first note $dom(Q_1) = dom(Q_2) = \{(h, h) : 1 \in dom(h) \vee 2 \in dom(h)\}$. Next, we consider heaps $h_1 = \{(1, 0)\}$ and $h_2 = \{(2, 0)\}$ with $(h_i, h_i) \in dom(Q_i)$. Thus, using $h = h_1 \bullet h_2$ and $h_1 \# h_2$, we have $(h, h) \in dom(Q_1 * Q_2)$. Moreover, a possible execution of $Q_1 * Q_2$ is $(h, \{(1, 1), (2, 2)\})$. Hence, $((h_1, h_2), \{(1, 1), (2, 2)\})$ is included in the left hand side of the instantiated concurrency property but not in the right hand side since we only have $((h_1, h_2), \{(1, 2), (2, 1)\})$ there.

We are now interested in relating concurrency property to the exchange law for concurrent processes. It turns out that the property is sufficient for validating a special case of the exchange law which we use to prove soundness of the concurrency rule.

Definition 5.2. We call two commands Q_1, Q_2 *pre-concurrent* iff we have for all tests p_1, p_2

$$p_1 \subseteq dom(Q_1) \wedge p_2 \subseteq dom(Q_2) \Rightarrow (p_1 * p_2) ; (Q_1 * Q_2) \subseteq (p_1 ; Q_1) * (p_2 ; Q_2) \ .$$

Lemma 5.3. *If commands Q_1 and Q_2 have the concurrency property then they are pre-concurrent.*

Proof. Assume $p_i \subseteq dom(Q_i)$. Then

$$\begin{array}{ll}
 & (p_1 * p_2) ; (Q_1 * Q_2) \\
= & \quad \{\!|\ \text{definition of } *,\ p_i \subseteq dom(Q_i) \text{ for } i = 1, 2\ |\!\} \\
 & \lhd ; (p_1 ; dom(Q_1) \times p_2 ; dom(Q_2)) ; \rhd ; (Q_1 * Q_2) \\
= & \quad \{\!|\ ; / \times \text{ exchange (1) } |\!\} \\
 & \lhd ; (p_1 \times p_2) ; (dom(Q_1) \times dom(Q_2)) ; \rhd ; (Q_1 * Q_2) \\
\subseteq & \quad \{\!|\ \text{concurrency property (2) } |\!\} \\
 & \lhd ; (p_1 \times p_2) ; (Q_1 \times Q_2) ; \rhd \\
= & \quad \{\!|\ ; / \times \text{ exchange (1) } |\!\} \\
 & \lhd ; (p_1 ; Q_1 \times p_2 ; Q_2) ; \rhd \\
= & \quad \{\!|\ \text{definition of } * \ |\!\} \\
 & (p_1 ; Q_1) * (p_2 ; Q_2) \ .
\end{array}$$

□

Interestingly, this special case of the exchange law already suffices to prove our second variant of the concurrency rule although the complete exchange law is needed for the concurrency rule in [7]; note also that the inclusion relations between the preconditions and the domains of the commands are the reverses of the ones in Lemma 4.3.

Lemma 5.4 (Concurrency Rule II) . *Let Q_1 and Q_2 have the concurrency property. Then*

$$\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\} \quad p_1 \subseteq \mathit{dom}(Q_1) \quad p_2 \subseteq \mathit{dom}(Q_2)}{\{p_1 * p_2\}\,Q_1 * Q_2\,\{r_1 * r_2\}}\ .$$

Proof.

$$\begin{array}{ll} & (p_1 * p_2)\,;(Q_1 * Q_2) \\ \subseteq & \quad \{\!\![\ \text{Lemma 5.3}\ ]\!\!\} \\ & (p_1\,; Q_1) * (p_2\,; Q_2) \\ \subseteq & \quad \{\!\![\ \{p_i\}\,Q_i\,\{r_i\}\ ]\!\!\} \\ & (Q_1\,; r_1) * (Q_2\,; r_2) \\ \subseteq & \quad \{\!\![\ \text{reverse exchange law (Lemma 3.6)}, \\ & \quad\ \ \text{since } r_1, r_2 \text{ as tests are compatible}\ ]\!\!\} \\ & (Q_1 * Q_2)\,;(r_1 * r_2)\ . \end{array}$$

□

In summary, we have presented two variations of the concurrency rule in our relational calculus. An advantage of Lemma 4.3 is that it only requires that the domains of the commands Q_i coincide with the respective preconditions, but needs no connection between the Q_i. Contrarily, Lemma 5.4 is more liberal w.r.t. the preconditions but requires the Q_i to have the concurrency property.

Still, usually at least one of the concurrency rules can be applied. Consider, for instance, parallel mergesort `ms` [11]:

$$\frac{\begin{array}{c}\{\mathit{array}(a,i,m)\}\,\mathtt{ms}(a,i,m)\,\{\mathit{sorted}(a,i,m)\} \\ \{\mathit{array}(a,m+1,j)\}\,\mathtt{ms}(a,m+1,j)\,\{\mathit{sorted}(a,m+1,j)\}\end{array}}{\begin{array}{c}\{\mathit{array}(a,i,m) * \mathit{array}(a,m+1,j)\} \\ \mathtt{ms}(a,i,m) * \mathtt{ms}(a,m+1,j) \\ \{\mathit{sorted}(a,i,m) * \mathit{sorted}(a,m+1,j)\}\end{array}}$$

where $\mathit{array}(a,i,j)$, assuming $i < j$, asserts that the store range with addresses $a{+}i$ to $a{+}j$ forms an array, i.e., contains elements of equal type, and $\mathit{sorted}(a,i,j)$ ensures that the content in that range is sorted. It is easy to define $\mathtt{ms}(a,i,m)$ in such a way that its domain is characterised by $\mathit{array}(a,i,m)$. Moreover, in any reasonable implementation the commands $\mathtt{ms}(a,i,m)$ and $\mathtt{ms}(a,m+1,j)$ even satisfy the concurrency property.

Thus, the concurrency rules in our relational calculus represent a feasible approach to enable reasoning about disjoint true concurrency.

6 Locality and the Frame Rule

We now turn to another important proof rule for modular reasoning. Validity of that rule is based on the concept of *locality* which describes the behaviour of programs that only access certain subsets of the available resources. Hence locality allows embedding a program into a larger context so that any resource not accessed by that program remains unchanged. This fact is expressed by the *frame rule*

$$\frac{\{p\}\,Q\,\{q\}}{\{p * r\}\,Q\,\{q * r\}}\ .$$

To obtain a suitable version of the frame rule in our relational calculus we first remind the reader of a central result of [7]. A predicate transformer F in that model is called *local* iff it satisfies the equation

$$F * \mathsf{skip} = F\ .$$

This equation characterises exactly the above-mentioned modularity concept. Each execution of the program F can be replaced by one that only operates on the necessary and possible smaller part of the state while the rest of it remains unchanged (abstractly denoted by the program skip). In the following we derive the same compact characterisation for commands.

First remember that emp is the unit of $*$ and $\mathsf{emp} \subseteq \mathsf{skip}$.

Lemma 6.1. *For arbitrary commands Q we have $Q \subseteq Q * \mathsf{skip}$.*

Proof. $Q = Q * \mathsf{emp} \subseteq Q * \mathsf{skip}$. □

To get the other inclusion, i.e., $Q * \mathsf{skip} \subseteq Q$, we need an additional assumption about Q. Surprisingly, this inequation can be derived from a property given in [4] which was called test preservation and used there to prove soundness of the frame rule.

Definition 6.2. A command Q *preserves* a test r iff

$$\lhd\,;\,(Q \times r)\,;\,\# \;\subseteq\; Q\,;\,\lhd\,;\,(\mathsf{skip} \times r)\ . \tag{3}$$

We call a command Q *local* iff Q preserves all tests.

Formula (3) means that when running Q on a part of the state such that the remainder of the state satisfies r one might also run Q first on the complete state and will still find an r-part in the result state.

Preservation of r by Q is an abstraction of the property that Q does not modify the free variables of r; a more refined version of this definition was given in [4] and another one was studied in [3]. Locality as preservation of all tests, does not seem very realistic in that domain. Nevertheless, it turns out to be equivalent to the algebraic formulation of [7]:

Theorem 6.3 *A command Q is local iff $Q * \mathsf{skip} \subseteq Q$.*

The proof can be found in the Appendix.

By Theorem 6.3 we may, as in [7], define local commands as fixpoints of the localising operation $(\ \cdot\ ) * \mathsf{skip}$.

With this definition of locality we now prove our variant of the frame rule. We take a similar direction as in Section 4 by defining sufficient conditions needed for a soundness proof. First notice that in [7] the compact definition of locality and the full exchange law are used to get validity of the small exchange law for local predicate transformers. The small exchange law reads

$$(P * Q) \,;\, R \;\leq\; (P \,;\, R) * Q$$

for programs P, Q, R and the refinement order $\leq$ of a locality bimonoid. Moreover this law is equivalent to soundness of the frame rule in such a structure. In our approach locality has the same definition, but the small exchange law does not hold. Therefore again a further sufficient condition is needed to simulate the relevant part of the small exchange law.

Definition 6.4. Call command Q *pre-framed* iff for all test commands p, r

$$p \subseteq dom(Q) \quad \Rightarrow \quad (p * r) \,;\, Q \;\subseteq\; (p \,;\, Q) * r \ .$$

The premise $p \subseteq dom(Q)$ informally states that p already ensures enough resources for the execution of Q. We will see that this is a sufficient condition to prove soundness of the frame rule. Notice that the conclusion is only a special case of the small exchange law.

In [4] we used a relational variant of the frame property to prove the frame rule. We will use it in this paper in a simplified form.

Definition 6.5. A command Q has the *frame property* iff

$$(dom(Q) \times \mathsf{skip}) \,;\, \rhd \,;\, Q \;\;\subseteq\;\; (Q \times \mathsf{skip}) \,;\, \rhd \ .$$

This property can be derived as a special case of the concurrency property by setting $Q_2 = \mathsf{skip}$ and assuming locality for Q_1, i.e., $Q_1 * \mathsf{skip} = Q_1$. Again, this property is sufficient for pre-framedness.

Lemma 6.6. *If Q has the frame property then Q is pre-framed.*

Proof. Assume $p \subseteq dom(Q)$. Then

$$
\begin{array}{ll}
 & (p * r) \,;\, Q \\
= & \quad \{\!\![\ \text{assumption}\ ]\!\!\} \\
 & ((p \,;\, dom(Q)) * r) \,;\, Q \\
= & \quad \{\!\![\ \text{definition of } * \text{ and } ;/\!\times \text{ exchange (1)}\ ]\!\!\} \\
 & \lhd \,;\, (p \times r) \,;\, (dom(Q) \times \mathsf{skip}) \,;\, \rhd \,;\, Q \\
\subseteq & \quad \{\!\![\ \text{frame property}\ ]\!\!\} \\
 & \lhd \,;\, (p \times r) \,;\, (Q \times \mathsf{skip}) \,;\, \rhd \\
= & \quad \{\!\![\ ;/\!\times \text{ exchange (1) and definition of } *\ ]\!\!\} \\
 & (p \,;\, Q) * r \ .
\end{array}
$$

□

Now we can easily prove the frame rule.

Lemma 6.7 (Frame Rule) . *Let Q be local and have the frame property. Moreover, assume $p \subseteq dom(Q)$. Then*

$$\frac{\{p\}\,Q\,\{q\}}{\{p * r\}\,Q\,\{q * r\}}\ .$$

Proof.

$$
\begin{array}{ll}
 & (p * r)\,;Q \\
\subseteq & \{\![\ p \subseteq dom(Q) \text{ and } Q \text{ pre-framed by Lemma 6.6 }]\!\} \\
 & (p\,;Q) * r \\
\subseteq & \{\![\ \text{by } \{p\}\,Q\,\{q\}\ ]\!\} \\
 & (Q\,;q) * r \\
\subseteq & \{\![\ r = \mathsf{skip}\,;r \text{ and reverse exchange law (Lemma 3.6),} \\
 & \quad \text{since } r, \mathsf{skip} \text{ as tests are compatible }]\!\} \\
 & (Q * \mathsf{skip})\,;(q * r) \\
\subseteq & \{\![\ Q \text{ local }]\!\} \\
 & Q\,;(q * r)\ .
\end{array}
$$

□

Next, we compare the structure of our proofs with the corresponding ones in [7] to point out the main differences. Since the small exchange law is not valid in our relational setting (not even for local commands), it was necessary to constrain the set of commands considered in the frame rule by an additional assumption. It turned out in [4] that the relational version of the frame property was an adequate substitute. We have shown here that this property also relates to pre-framed commands which already ensure the relevant part of the small exchange law. Structurally, the proof of this frame rule becomes as simple as the one for the predicate transformer approach in [7]. Due to the angelic character of relations, the rule itself needs the additional premise $p \subseteq dom(Q)$.

As a further remark, the approach of [7] requires special functions for the semantics of Hoare triples. They are called *best predicate transformers* and are used as an adequate substitute for assertions. Intuitively these functions simulate the allocation of resources that are characterised by pre- and postconditions. In our calculus this can be handled by composing tests with the universal relation. However, since we have a non-trivial test algebra in the relational setting, tests by themselves already admit a suitable representation of pre- and postconditions.

7 Dual Correctness Triples

The previous sections presented an approach to include the concurrency and frame rules in the given relational approach to separation logic [4] by requiring additional assumptions and hence restricting the proof rules. In this section we present some further applications for the reverse exchange law. We link it with

the definitions of triples dual to the ones of Hoare. By this we will again see that the concurrency and frame rules can be easily derived using the reverse exchange law.

Definition 7.1. As in [6], for commands P, Q, R we define *Plotkin* triples by

$$\langle P, Q\rangle \to R \;\; \Leftrightarrow_{df} \;\; R \subseteq P\,;Q$$

and dual partial correctness triples by

$$P\,[Q]\,R \;\; \Leftrightarrow_{df} \;\; P \subseteq Q\,;R\ .$$

Intuitively, the former characterise possible states satisfying the postcondition R after the execution of Q starting from P while the latter symmetrically describes possible starting states of P that end in R after the execution of Q. The notation is inspired by Plotkin's structural operational semantics [13] in which $\langle C, s\rangle \to t$ means that evaluation of term C starting in state s may lead to term t. According to [6], dual partial correctness triples can e.g. be used as a method for the generation of test cases. Assuming R represents erroneous final states of Q then P characterises some conditions that will lead to such error situations. Plotkin triples can used for a dual application.

Using the relationship between tests and commands given in Section 4, in our calculus the dual partial correctness triples transform into

$$(p\,;U)\,[Q]\,(q\,;U) \;\Leftrightarrow\; p\,;U \subseteq Q\,;(q\,;U) \;\Leftrightarrow\; p \subseteq (Q\,;q)\,;U \;\Leftrightarrow\; p \subseteq \mathit{dom}(Q\,;q)$$

and, symmetrically, Plotkin triples into

$$\langle U\,;p\ ,Q\rangle \to U\,;q \;\Leftrightarrow\; U\,;q \subseteq (U\,;p)\,;Q \;\Leftrightarrow\; q \subseteq U\,;(p\,;Q) \;\Leftrightarrow\; q \subseteq \mathit{cod}(p\,;Q)\ .$$

We concentrate on dual partial correctness triples and use the abbreviation $p\;[\![\,Q\,]\!]\;q \Leftrightarrow_{df} (p\,;U)\,[Q]\,(q\,;U) \Leftrightarrow p \subseteq \mathit{dom}(Q\,;q)$. Dual results hold for Plotkin triples.

The central interest of these new triples lies in the following result.

Lemma 7.2. *The concurrency rule for dual partial correctness or Plotkin triples holds iff the reverse exchange law holds.*

A proof for this lemma can be derived dually to [7]. Unfortunately, in our setting the reverse exchange law does not hold unconditionally. However, we will see that under an assumption of compatibility the concurrency and frame rules can still be derived. Note that it was not needed to assume compatibility for the proof rules with Hoare triples since tests already come with that property. In contrast, the new triples do not need additional assumptions besides the compatibility condition.

We begin with an auxiliary result.

Lemma 7.3. *Assume P, Q are forward compatible. Then $\mathit{dom}(P) * \mathit{dom}(Q) = \mathit{dom}(P * Q)$, i.e., $*$ distributes over domain.*

A proof can be found in the Appendix.

Lemma 7.4. *If Q_1, Q_2 are forward compatible then the concurrency rule for dual partial correctness triples holds, i.e., for tests p_1, p_2, q_1, q_2*

$$\frac{p_1 \llbracket Q_1 \rrbracket\, q_1 \qquad p_2 \llbracket Q_2 \rrbracket\, q_2}{p_1 * p_2 \llbracket Q_1 * Q_2 \rrbracket\, q_1 * q_2} \,.$$

Again this holds also when Q_1 and Q_2 are backward compatible and Plotkin instead of dual partial correctness triples are used.

Proof. By assumption we have $p_1 \subseteq dom(Q_1 \,;\, q_1)$, $p_2 \subseteq dom(Q_2 \,;\, q_2)$ and the restricted variant of the reverse exchange law. Hence

$$\begin{array}{ll} & p_1 * p_2 \\ \subseteq & dom(Q_1 \,;\, q_1) * dom(Q_2 \,;\, q_2) \\ = & dom((Q_1 \,;\, q_1) * (Q_2 \,;\, q_2)) \\ \subseteq & dom((Q_1 * Q_2) \,;\, (R_1 * R_2)) \;. \end{array}$$

□

We characterised the behaviour of the triples "dual" on purpose since the calculations given above are symmetric to the algebraic approach of [7]. It is not hard to see that a further application of the compact characterisation of locality presented in Section 6 also gives the following result.

Lemma 7.5. *If Q is local and forward compatible with* skip *then the frame rule for dual partial correctness triples holds, i.e.,*

$$\frac{p \llbracket Q \rrbracket\, q}{p * r \llbracket Q \rrbracket\, q * r} \,.$$

(A dual result again holds for Plotkin triples).

Proof. Assume $p \subseteq dom(Q \,;\, q)$ for a command Q and test q. Hence

$$\begin{array}{ll} & p * r \\ \subseteq & dom(Q \,;\, q) * (\mathsf{skip} \,;\, r) \\ = & dom(Q \,;\, q) * dom(\mathsf{skip} \,;\, r) \\ = & dom((Q \,;\, q) * (\mathsf{skip} \,;\, r)) \\ \subseteq & dom((Q * \mathsf{skip}) \,;\, (q * r)) \\ \subseteq & dom(Q \,;\, (q * r)) \;. \end{array}$$

□

8 Further Variations

Both proof rules of the previous section have the restriction that compatible pairs of commands are needed. The reason for this is that, by Lemma 2.6, in the relational approach only $\# \subseteq \rhd \,;\, \lhd$ holds. If we would have $\mathsf{skip} \times \mathsf{skip} \subseteq \rhd \,;\, \lhd$

the proof of the reverse exchange law would not have any restrictions. However this requires an extension of the definition of $\rhd$ such that the composition $\rhd\,;\lhd$ has the same behaviour as skip on incombinable pairs of states.

An idea would be to lift commands to relations between sets containing at most a single state. The empty set is then the result of joining incombinable pairs of states. We define $\Sigma_s =_{df} \{\{\sigma\} : \sigma \in \Sigma\} \cup \{\emptyset\}$ and, for sets $X, Y \in \Sigma_s$,

$$X *_s Y \lhd (X, Y) \tag{4}$$

where $X *_s Y =_{df} \{\sigma_1 \bullet \sigma_2 : \sigma_1 \# \sigma_2, \sigma_1 \in X, \sigma_2 \in Y\}$. In the special case with $X = \{\sigma_1\} \neq \emptyset$ and $Y = \{\sigma_2\} \neq \emptyset$ we have that

$$\{\sigma_1\} *_s \{\sigma_2\} \lhd (\{\sigma_1\}, \{\sigma_2\}) .$$

We denote the lifting of this operation to relations by $*_s$ again.

This modification allows a relational model of the algebraic structure of a *locality bimonoid* defined in [7].

Definition 8.1. A *locality bimonoid* is an algebraic structure $(S, \leq, *, 1_*, ;, 1_;)$ where $(S, \leq)$ is partially ordered and $*$, ; are monotone operations on S. Moreover, $(S, *, 1_*)$ needs to be a commutative monoid and $(S, ;, 1_;)$ a monoid. Additionally, the structure has to satisfy the exchange law and $1 * 1 = 1$.

To obtain a relational model for this structure one may interpret the order $\leq$ as the reverse set inclusion order $\supseteq$. Of course, by this the mentioned reverse exchange law turns into the normal one and the relational approach into a refinement-based setting. Moreover, we have the following result.

Lemma 8.2. skip *is idempotent w.r.t.* $*$*, i.e.,* $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

Proof. Since $\mathsf{skip} * \mathsf{skip}$ is a test the $\subseteq$-direction is immediate. □

In summary, we summarise the following result.

Lemma 8.3. $(\mathcal{P}(\Sigma_s \times \Sigma_s), \supseteq, *_s, \mathsf{emp}, ;, \mathsf{skip})$ *forms a locality bimonoid.*

Note that by this modification $\sqcup$ and $\sqcap$ turn into $\cap$ and $\cup$. In particular, the test subalgebra is used as an algebraic counterpart to model assertions. Hence, the interpretation of the notion of a test becomes very unnatural, since e.g. $p \wedge q$ will be identified, unusually, in the algebra with $p \sqcup q$ and $p \vee q$ with $p \sqcap q$. Algebraically these modifications of the model entail simplifications. There are no additional constraints needed to validate the reverse exchange law and hence the original concurrency and frame rules hold. The reason for this is the inequation $\mathsf{skip} \times \mathsf{skip} \subseteq \rhd\,;\lhd$ that requires the introduction of an extra failure-state to capture the join of incombinable states. However, considering this extra failure-state makes the whole approach more complicated and artificial from the model-theoretical view.

9 Conclusion

Although neither the full nor small exchange law holds in the relational calculus, we were still able to obtain reasonable variants of the concurrency and frame rules. The proofs greatly benefit from the (restricted) reverse exchange law and hence are almost as simple as the ones in [7]. The advantage of the relational framework is that it admits choice and the corresponding distributivity laws without effort by using relational union.

Further work on this approach includes investigations on so-called interference relations [5]. The intention with such relations is to provide admissible behaviour of commands in a concurrent context so that interference between these commands is excluded. By this we hope to include more concrete models for the application domain of the presented relational approach.

Acknowledgements. We are grateful to Tony Hoare for fruitful discussions and comments and to Andreas Zelend for valuable remarks. Moreover, we thank all reviewers for their comments that helped to significantly improve the presentation of this paper. This research was partially funded by the DFG project *MO 690/9-1 AlgSep — Algebraic Calculi for Separation Logic.*

References

1. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science 375, 227–270 (2007)
2. Calcagno, C., O'Hearn, P.W., Yang, H.: Local Action and Abstract Separation Logic. In: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, pp. 366–378. IEEE Press (2007)
3. Dang, H.-H., Höfner, P.: Variable Side Conditions and Greatest Relations in Algebraic Separation Logic. In: de Swart, H. (ed.) RAMICS 2011. LNCS, vol. 6663, pp. 125–140. Springer, Heidelberg (2011)
4. Dang, H.-H., Höfner, P., Möller, B.: Algebraic separation logic. Journal of Logic and Algebraic Programming 80(6), 221–247 (2011)
5. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
6. Hoare, C.A.R.: An Algebra for Program Designs. Notes on Summer School in Software Engineering and Verification in Moscow (2011), http://research.microsoft.com/en-us/um/redmond/events/sssev2011/slides/tony_1.pptx
7. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On Locality and the Exchange Law for Concurrent Processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)
8. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra and its Foundations. Journal of Logic and Algebraic Programming 80(6), 266–296 (2011)

9. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19(3), 427–443 (1997)
10. Möller, B.: Kleene getting lazy. Science of Computer Programming 65, 195–214 (2007)
11. O'Hearn, P.W.: Resources, Concurrency, and Local Reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
12. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
13. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program., 60–61, 17–139 (2004)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society (2002)

10 Appendix: Deferred Proofs

Proof of Lemma 2.8.
For arbitrary σ we have

$$
\begin{array}{ll}
 & \sigma \; dom(P * Q) \; \sigma \\
\Leftrightarrow & \quad \{\!\![\text{ definitions of } * \text{ and domain }]\!\!\} \\
 & \exists\, \sigma_1, \sigma_2, \tau_1, \tau_2 \,.\, \sigma_1 \# \sigma_2 \;\wedge\; \sigma = \sigma_1 \bullet \sigma_2 \;\wedge\; \tau_1 \# \tau_2 \;\wedge\; \sigma_1 \; P \; \tau_1 \;\wedge\; \sigma_2 \; Q \; \tau_2 \\
\Rightarrow & \quad \{\!\![\text{ omitting conjunct } \tau_1 \# \tau_2 \text{ and shifting quantification over } \tau_1, \tau_2 \;]\!\!\} \\
 & \exists\, \sigma_1, \sigma_2 \,.\, \sigma_1 \# \sigma_2 \;\wedge\; \sigma = \sigma_1 \bullet \sigma_2 \;\wedge\; \exists\, \tau_1, \tau_2 \,.\, \sigma_1 \; P \; \tau_1 \;\wedge\; \sigma_2 \; Q \; \tau_2 \\
\Leftrightarrow & \quad \{\!\![\text{ definition of domain }]\!\!\} \\
 & \exists\, \sigma_1, \sigma_2 \,.\, \sigma_1 \# \sigma_2 \;\wedge\; \sigma = \sigma_1 \bullet \sigma_2 \;\wedge\; \sigma_1 \; dom(P) \; \sigma_1 \;\wedge\; \sigma_2 \; dom(Q) \; \sigma_2 \\
\Leftrightarrow & \quad \{\!\![\text{ definition of } * \;]\!\!\} \\
 & \sigma \; (dom(P) * dom(Q)) \; \sigma \; .
\end{array}
$$

$\square$

In Def. 6.2 we stated that a command Q *preserves* a test r iff

$$\lhd \,;\, (Q \times r) \,;\, \# \;\subseteq\; Q \,;\, \lhd \,;\, (\mathsf{skip} \times r)$$

and called a command Q *local* iff Q preserves all tests.

We first list a few useful properties in connection with these notions.

Lemma 10.1

1. skip *preserves* skip .
2. *For arbitrary Q and r we have*

$$\lhd \,;\, (Q \times r) \,;\, \# \quad\subseteq\quad (Q * \mathsf{skip}) \,;\, \lhd \,;\, (\mathsf{skip} \times r) \; .$$

3. *If Q preserves a test r then $Q * r \subseteq Q \,;\, (\mathsf{skip} * r)$.*
   In particular, $\mathsf{skip} * \mathsf{skip} \subseteq \mathsf{skip}$. *Hence if Q is local then $Q * \mathsf{skip} \subseteq Q$.*

Proof

1. The claim follows immediately by setting $Q = \mathsf{skip} = r$ in Definition 6.2.
2. We calculate:

$$
\begin{array}{ll}
 & \lhd\,;\,(Q \times r)\,;\,\# \\
= & \quad \{\!\![\text{ neutrality of } \mathsf{skip} \,]\!\!\} \\
 & \lhd\,;\,(Q\,;\mathsf{skip} \times \mathsf{skip}\,;r)\,;\,\# \\
= & \quad \{\!\![\;;/\times \text{ exchange (1) }]\!\!\} \\
 & \lhd\,;\,(Q \times \mathsf{skip})\,;\,(\mathsf{skip} \times r)\,;\,\# \\
= & \quad \{\!\![\text{ by Definition 2.4 }]\!\!\} \\
 & \lhd\,;\,(Q \times \mathsf{skip})\,;\,\#\,;\,(\mathsf{skip} \times r) \\
\subseteq & \quad \{\!\![\;\# \subseteq \rhd\,;\lhd \text{ and isotony }]\!\!\} \\
 & \lhd\,;\,(Q \times \mathsf{skip})\,;\,\rhd\,;\,\lhd\,;\,(\mathsf{skip} \times r) \\
= & \quad \{\!\![\text{ definition of } * \,]\!\!\} \\
 & (Q * \mathsf{skip})\,;\,\lhd\,;\,(\mathsf{skip} \times r)\ .
\end{array}
$$

3. The first claim is immediate from the definition of locality by right-composing both sides of the inclusion with $\rhd$, isotony and the definition of $*$. Hence the second claim is trivial by isotony. The third claim follows by setting $r = \mathsf{skip}$ and using $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

□

We can now give the

Proof of Theorem 6.3

The direction (⇒) is just Lemma 10.1.3. For (⇐) we obtain by Lemma 10.1.3 and the assumption, for arbitrary test r,

$$\lhd\,;\,(Q \times r)\,;\,\# \;\subseteq\; (Q * \mathsf{skip})\,;\,\lhd\,;\,(\mathsf{skip} \times r) \;\subseteq\; Q\,;\,\lhd\,;\,(\mathsf{skip} \times r)\ .$$

□

Corollary 10.2. $Q * \mathsf{skip} \subseteq Q \;\Leftrightarrow\; \lhd\,;(Q \times \mathsf{skip})\,;\# \subseteq Q\,;\lhd\,.$

Proof The direction (⇐) follows from isotony. For the other direction we immediately get by definition and isotony $\lhd\,;(Q \times \mathsf{skip})\,;\rhd\,;\lhd \subseteq Q\,;\lhd$ since $Q * \mathsf{skip} \subseteq Q$. Now the claim follows from Lemma 2.6 using $\# \subseteq \rhd\,;\lhd$. □

Next we turn to Section 7. To prove Lemma 7.3 we first sum up a few results.

Corollary 10.3. $\#\,;(U \times U)\,;\rhd = \rhd\,;U.$

For a proof we refer to [4].

Lemma 10.4 *If commands P, Q are forward compatible then $(P\,;U) * (Q\,;U) = (P * Q)\,;U$.*

Proof. We calculate

$$
\begin{array}{ll}
 & (P\,;U) * (Q\,;U) \\
= & \quad \{\!|\ \text{definition of } *\ |\!\} \\
 & \lhd\,;(P\,;U \times Q\,;U)\,;\rhd \\
= & \quad \{\!|\ \text{Lemma 2.6, Equation (1)}\ |\!\} \\
 & \lhd\,;\#\,;(P \times Q)\,;(U \times U)\,;\rhd \\
\subseteq & \quad \{\!|\ P, Q \text{ forward compatible}\ |\!\} \\
 & \lhd\,;(P \times Q)\,;\#\,;(U \times U)\,;\rhd \\
= & \quad \{\!|\ \text{Corollary 10.3}\ |\!\} \\
 & \lhd\,;(P \times Q)\,;\rhd\,;U \\
= & \quad \{\!|\ \text{definition of } *\ |\!\} \\
 & (P * Q)\,;U\ .
\end{array}
$$

The reverse inequation follows similarly from Corollary 10.3 and isotony. □

Finally we are able to prove Lemma 7.3.

Proof of Lemma 7.3

First note that $dom(P) = P\,;U \cap \mathsf{skip}$. The same holds for Q. By this we calculate

$$dom(P) * dom(Q) = (P;U \cap \mathsf{skip}) * (Q;U \cap \mathsf{skip}) \subseteq (P;U) * (Q;U) = (P * Q);U\ .$$

Moreover $dom(P) * dom(Q) \subseteq \mathsf{skip}$ since both are tests. Hence we can conclude $dom(P) * dom(Q) \subseteq (P * Q)\,;U \cap \mathsf{skip} = dom(P * Q)$.

The reverse inclusion was shown in Lemma 2.8. □

Unifying Correctness Statements

Walter Guttmann

Institut für Programmiermethodik und Compilerbau, Universität Ulm
walter.guttmann@uni-ulm.de

Abstract. Partial, total and general correctness and further models of sequential computations differ in their treatment of finite, infinite and aborting executions. Algebras structure this diversity of models to avoid the repeated development of similar theories and to clarify their range of application. We introduce algebras that uniformly describe correctness statements, correctness calculi, pre-post specifications and loop refinement rules in five kinds of computation models. This extends previous work that unifies iteration, recursion and program transformations for some of these models. Our new description includes a relativised domain operation, which ignores parts of a computation, and represents bound functions for claims of termination by sequences of tests. We verify all results in Isabelle heavily using its automated theorem provers.

1 Introduction

Sequential computations have many different models with varying degrees of precision as regards their ability to distinguish finite, infinite and aborting executions (which terminate due to an error; 'finite' means 'normally terminating'). Partial correctness models [10, 23, 29, 33] ignore infinite and aborting executions, general correctness models [2–4, 11, 13, 27, 32, 34, 38] represent but do not distinguish infinite and aborting executions, and total correctness models [7, 10, 18, 25, 31, 39] ignore finite executions in the presence of infinite and aborting ones. Yet other models ignore finite and infinite executions when aborting ones are present [17, 22] or represent finite, infinite and aborting executions independently [16].

Having a variety of models is useful: better precision is not always desired as it entails more details (which might be unnecessary or distracting for some applications) and typically a more complex theory (which might hinder comprehension or automation). However, this diversity of models should be structured to avoid the repeated development of similar theories for similar models, to improve our understanding of their connections and characterising properties and to encourage a systematic exploration with an eye to discovering new models.

Algebra provides the required structure. Key aspects of the models, such as the semantics of iteration or the infinite executions of a computation, are described by operations and axioms which are general enough to capture various computation models, yet powerful enough for the derivation of results known from particular models. These results are then recognised to hold in all models satisfying the common axioms. Examples include complex program transformations, separation rules and refinement laws, for which one common proof establishes

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 198–219, 2012.
© Springer-Verlag Berlin Heidelberg 2012

validity across several models [14, 16, 17]. Individual models are characterised by adding specific axioms to the common ones. Moreover, the axioms are suited to support by automated theorem provers and SMT solvers [16, 19].

The present paper extends this unifying approach to correctness statements and their calculi. Correctness statements similar to Hoare triples claim that a computation has only restricted kinds of executions, for example, that there are no aborting executions or that all finite executions end in a given set of states. Clearly, such guarantees can be given only in a computation model which is precise enough to talk about the kinds of executions involved. But in general there are several computation models capable of expressing a particular statement. At the level of concrete models, we therefore distinguish between the computation models and the kinds of correctness statements each supports. At the algebraic level, our approach is unifying in both dimensions: one statement applies in various models to various correctness claims.

We give a propositional correctness calculus for the unified correctness statements and show its soundness and completeness. The latter presumes boundedly non-deterministic programs. The calculus, too, unifies different computation models and correctness claims. We furthermore extend our unifying approach to pre-post specifications and loop refinement rules useful for program construction. Innovations which facilitate our development are a relativised domain operation and the representation of bound functions by sequences of tests.

Thus the contributions of the present paper are as follows:

- A generalisation of domain semirings [8, 9]: the new operation uniformly describes the domain of a part of a computation, such as its aborting, infinite or finite executions or combinations thereof. This is a form of relativisation: the operation gives the domain up to certain executions which are ignored. Technically, an element Z is singled out and the domain axioms are relaxed so as to ignore parts of elements contained in Z. Most of the theory of domain and antidomain semirings is relativised this way, including modal semirings with their diamond and box operators.
- An algebraic description of termination arguments known, for example, from the total correctness while-loop rule of the Hoare calculus. This is done by capturing the loop variant or bound function by a sequence of tests.
- An extension of algebraic accounts of correctness statements and their calculi [13, 14, 19, 29, 33, 34], which unifies existing propositional Hoare calculi in two ways. First, it applies to several computation models, which vary in their ability to describe aborting and infinite executions. Second, for each model, it applies to several kinds of correctness statements, which vary in their claims about aborting, infinite and finite executions or combinations thereof, up to the precision allowed by the model. In particular, this uniformly describes claims of partial, total and general correctness.
- An algebraic description of pre-post specifications and loop refinement rules based on the above correctness statements. This generalises previous works [12–14, 39] again by uniformly applying to several computation models and several kinds of correctness claims.

Together they achieve the main contribution: a framework that unifies correctness reasoning for various computation models and correctness claims.

All results are verified in Isabelle making heavy use of its integrated automated theorem provers. The proofs can be found in the theory files, which are available at http://www.uni-ulm.de/en/in/pm/staff/guttmann/algebra/.

We deal with computation models at various levels of abstraction. Concrete models appear in the literature, for example, in terms of relations over extended state spaces, predicates or predicate transformers. In Section 2 we describe a number of these models in a uniform setting, namely matrices of relations, and discuss the different kinds of correctness statements they feature. Abstracting from the matrix representation, in Section 3 we introduce relative domain semirings, which allow us to express the correctness statements in a uniform way. At the same level we express iteration in Section 4. Finally, in Section 5 we axiomatise properties of preconditions and while-programs, based on which we introduce the unified correctness calculus, pre-post specifications and loop refinement laws.

2 Models

This section gives an overview of the models and various kinds of correctness statements which will be algebraically captured in the remainder of this paper. We distinguish between the computation models and the correctness statements applicable for each model. The models vary according to precision as regards their ability to describe infinite and aborting executions in addition to finite ones. Each model may support different kinds of correctness statements about the executions of a computation limited by its precision.

All of the following models describe sequential, non-deterministic computations and are based on relations over a state space given by the possible values of program variables. The program

$$R = (\mathsf{while}\ x \leq 1\ \mathsf{do}\ x := x/x)$$

is our running example. Its variable x has values in $\mathbb{N}$ and x/x is integer division. Execution of R leaves x unchanged unless $x = 1$, in which case the execution does not terminate, or $x = 0$, in which case it aborts due to division by zero. However, not all computation models can represent aborting or infinite executions.

2.1 Partial Correctness

In the first model, the program R is a binary relation over the state space $\mathbb{N}$. A pair (x, x') in R specifies that there is an execution of R which starts in state x and terminates in x'. Hence

$$R = \{(x, x) \mid x \geq 2\}$$

comprises the finite executions of the program. Because it is deterministic, R is a partial function; in general there may be more than one final state x' related to a

single initial state x. This simple model has no provision for representing aborting or infinite executions. The partiality of R indicates 'missing' finite executions, but is otherwise unrelated to the presence of aborting or infinite executions; see also [38].

Therefore the typical correctness claim in this model is about partial correctness. For conditions or sets of states p and q, the Hoare triple $p\{R\}q$ expresses that every execution of R which starts in a state in p and terminates normally, does so in a state in q. This triple claims nothing about infinite or aborting executions.

The Hoare triple $p\{R\}q$ is algebraically formalised by $p \cdot R \cdot q' \leq 0$ using tests p and q [29]. Tests correspond to subsets of the identity relation and act as filters in a sequential composition. The test q' is the complement of q, the operation $\cdot$ is relational composition, $\leq$ is subset and 0 is the empty relation. According to the inequality there is no execution in R which starts in p and ends in a state not in q.

2.2 Total Correctness

The second model augments the above relational model to represent executions that do not terminate normally [18, 31]. It is an abstraction of the 'designs' of the Unifying Theories of Programming [25]. The program R is a 2×2 matrix whose entries are relations over $\mathbb{N}$, namely

$$R = \begin{pmatrix} \top & \top \\ W & X \end{pmatrix}, \qquad \begin{aligned} W &= \{(x, x') \mid x \leq 1\} \\ X &= \{(x, x) \mid x \geq 2\} \cup W \end{aligned}$$

For every program in this model, both entries in the top row are the universal relation $\top = \mathbb{N} \times \mathbb{N}$. They are chosen so as to appropriately propagate the information captured in W through a sequential composition. Subsequent models feature matrices with other combinations of 0 and $\top$ entries providing a characteristic, constant structure for each model.

The entry W represents the states from which executions exist that do not terminate normally. It is a vector, that is, a relation in which every state is related either to all states or to none, and therefore corresponds to a set of states. No distinction is made between infinite and aborting executions: $(x, x') \in W$ means that there is an execution starting in x which does not terminate normally.

The entry X represents the finite executions of the program with the proviso $W \subseteq X$. This requirement leads to a demonic non-deterministic choice: for example, the endless loop is the matrix with four $\top$ entries, which is an annihilator of the non-deterministic choice given by componentwise union. Because of this, computations with both finite and infinite or aborting executions starting in the same state cannot be represented properly. For example, consider $R + \mathsf{skip}$ using the non-deterministic choice $+$ and the program skip that does not change the state. The matrix representation of skip has the empty relation as W and the identity relation as X. In the initial state $x = 1$ there is both an infinite and a finite execution in the computation $R + \mathsf{skip}$, but in the current model $R = R + \mathsf{skip}$. Thus the finite execution of skip is ignored in the presence of R.

This model supports total correctness claims. The Hoare triple $p\{R\}q$ now expresses that every execution of R which starts in p terminates normally in q. Again this is formalised by $p \cdot R \cdot q' \leq 0$ [39]. The order $\leq$ is the subset relation lifted componentwise to matrices. The test p is represented just as a program by a 2×2 matrix of the above form, using $W = 0$ and a subset U of the identity relation as X; a similar representation is used for the test q'. The computation 0 with no executions is represented by a matrix with $W = X = 0$. The relational operations are lifted to matrices in the standard way, so that $p \cdot R \cdot q' \leq 0$ elaborates as

$$\begin{pmatrix} \top & \top \\ 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & \top \\ W & X \end{pmatrix} \cdot \begin{pmatrix} \top & \top \\ 0 & Y' \end{pmatrix} = \begin{pmatrix} \top & \top \\ UW & UW{+}UXY' \end{pmatrix} \leq \begin{pmatrix} \top & \top \\ 0 & 0 \end{pmatrix}$$

which is equivalent to $UW \subseteq 0 \wedge UXY' \subseteq 0$. As here, we frequently omit the operator $\cdot$ for relational composition, and we contrast $\subseteq$ on the components with its lifted counterpart $\leq$. The first term $UW \subseteq 0$ expresses that all executions starting in p terminate normally. The second term $UXY' \subseteq 0$ claims partial correctness: no execution starting in p terminates in q'. Their conjunction holds, for example, using $U = \{(x, x) \mid 2 \leq x \leq 5\}$ and $Y = \{(x, x) \mid x \leq 5\}$.

However, partial correctness cannot be claimed alone. In particular, the 'weak correctness' claim $p \cdot R = p \cdot R \cdot q$ of [39] reduces to $UXY' \subseteq UW$. This expresses that no execution starting in a state in p terminates in q', provided all executions starting in the same state terminate normally (whence $UW = 0$).

2.3 General Correctness

The third model removes the restriction imposed by the previous one, so that finite executions can be represented independently of executions which do not terminate normally [13, 31]. It is an abstraction of the 'prescriptions' of the Unifying Theories of Programming [11]. The independence is achieved by modifying the structure of the 2×2 matrices. The program R becomes

$$R = \begin{pmatrix} \top & 0 \\ W & X \end{pmatrix}, \qquad \begin{aligned} W &= \{(x, x') \mid x \leq 1\} \\ X &= \{(x, x) \mid x \geq 2\} \end{aligned}$$

Now the top-right entry is the empty relation 0 instead of $\top$ and the restriction $W \subseteq X$ is abandoned. Otherwise the interpretations of W and X remain as in the total correctness model. In the current model, $R \neq R + \mathsf{skip}$ and $R + \mathsf{skip}$ has precisely the two expected executions starting in the state $x = 1$. Still there is no distinction between infinite and aborting executions.

This model supports both partial and total correctness claims, and is typically called 'general correctness' [27]. First, observe that

$$\begin{pmatrix} \top & 0 \\ 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & 0 \\ W & X \end{pmatrix} \cdot \begin{pmatrix} \top & 0 \\ 0 & Y' \end{pmatrix} = \begin{pmatrix} \top & 0 \\ UW & UXY' \end{pmatrix} \leq \begin{pmatrix} \top & 0 \\ \top & 0 \end{pmatrix}$$

is equivalent to the partial correctness claim $UXY' \subseteq 0$. The matrix on the right-hand side of the inequality represents the program loop which has only

infinite executions. On the level of programs the partial correctness claim is thus formalised by $p \cdot R \cdot q' \leq \mathsf{loop}$ [13]. This is equivalent to $p \cdot R = p \cdot R \cdot q$ in the current model. Second, set $Y' = 0$ and observe that

$$\begin{pmatrix} \top & 0 \\ 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & 0 \\ W & X \end{pmatrix} \cdot \begin{pmatrix} \top & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} \top & 0 \\ UW & 0 \end{pmatrix} \leq \begin{pmatrix} \top & 0 \\ 0 & 0 \end{pmatrix}$$

is equivalent to $UW \subseteq 0$, which expresses that all executions starting in p terminate normally. On the level of programs this is formalised by $p \cdot R \cdot 0 \leq 0$.

The conjunction of the two inequalities amounts to a total correctness claim, but it is not required to use the same precondition p in both inequalities. This makes it possible to state claims about termination independently from claims about finite executions. For example, for the computation $R + \mathsf{skip}$ the partial correctness claim holds using $U = Y = \{(x, x) \mid x \leq 5\}$ and the claim about normal termination holds using $U = \{(x, x) \mid x \geq 2\}$.

2.4 Extended Designs

The fourth model called 'extended designs' distinguishes aborting and infinite executions [17, 22]. In this model the program R is the 3×3 matrix

$$R = \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ V & W & X \end{pmatrix}, \qquad \begin{aligned} V &= \{(x, x') \mid x = 0\} \\ W &= \{(x, x') \mid x = 1\} \cup V \\ X &= \{(x, x) \mid x \geq 2\} \cup V \end{aligned}$$

The extra row/column stores the main extension with respect to the previously discussed models, namely the entry V which is a vector similar to W. The vector V represents the states from which aborting executions exist, while W analogously represents the infinite executions. Like designs, extended designs make the restrictions $V \subseteq W$ and $V \subseteq X$. Because of them, in the presence of an aborting execution there is no way to distinguish finite or infinite executions. For example, consider the program $S = (\mathsf{if}\ x = 1\ \mathsf{then\ loop\ else\ skip})$. In the current model $R = R + S$ and the finite execution of S in the state $x = 0$ is ignored in the presence of R.

Several kinds of correctness claims are possible in this model. For the first, observe that $V \subseteq W$ implies $UV \subseteq UW$ and therefore

$$\begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ V & W & X \end{pmatrix} \cdot \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & 0 & Y' \end{pmatrix} = \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ UV & UW & UV{+}UXY' \end{pmatrix} \leq \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

is equivalent to $UW \subseteq 0 \wedge UXY' \subseteq 0$. Hence $p \cdot R \cdot q' \leq 0$ formalises a total correctness claim, namely that all executions starting in p terminate in q. In particular, no infinite executions start in p and therefore also no aborting ones. Another correctness claim is obtained by

$$\begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ V & W & X \end{pmatrix} \cdot \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & 0 & Y' \end{pmatrix} = \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ UV & UW & UV{+}UXY' \end{pmatrix} \leq \begin{pmatrix} \top & \top & \top \\ 0 & \top & 0 \\ 0 & \top & 0 \end{pmatrix}$$

which is equivalent to $UV \subseteq 0 \wedge UXY' \subseteq 0$. The matrix on the right-hand side of the inequality represents loop in this model. Hence the claim is formalised by $p \cdot R \cdot q' \leq \mathsf{loop}$ and expresses that no aborting executions start in p and all finite executions starting there end in q. It states nothing about the infinite executions.

Two further claims are obtained by setting $Y' = 0$ again. First, $p \cdot R \cdot 0 \leq 0$ expresses that no infinite and therefore no aborting executions start in p. Second, $p \cdot R \cdot 0 \leq \mathsf{loop}$ expresses that no aborting executions start in p. It is thus possible to make statements about aborting and infinite executions, but partial correctness cannot be claimed alone. In particular, $p \cdot R = p \cdot R \cdot q$ reduces to $UXY' \subseteq UV$, which expresses that no execution starting in a state in p terminates in q', provided no execution starting in the same state aborts.

2.5 Finite, Infinite and Aborting Executions

The fifth model treats finite, infinite and aborting executions independently [16]. The independence is achieved by modifying the structure of the 3×3 matrices of extended designs. The program R becomes

$$R = \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ V & W & X \end{pmatrix}, \qquad \begin{aligned} V &= \{(x, x') \mid x = 0\} \\ W &= \{(x, x') \mid x = 1\} \\ X &= \{(x, x) \mid x \geq 2\} \end{aligned}$$

Now the second and third entries in the top row are the empty relation 0 instead of $\top$ and the restrictions $V \subseteq W$ and $V \subseteq X$ are abandoned. Otherwise the interpretations of V, W and X remain as for extended designs. In the current model, $R \neq R + S$ using $S = (\mathsf{if}\ x = 1\ \mathsf{then\ loop\ else\ skip})$ again. Moreover, the computation $R + S$ has an aborting and a finite execution starting in the state $x = 0$, and the computation $R + S + \mathsf{loop}$ additionally has an infinite one. Thus finite, infinite and aborting executions may occur independently.

Claims about finite, infinite and aborting executions can be stated independently, too (see [20] for a derivation from different execution methods based on computation trees). Observe that

$$\begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ 0 & 0 & U \end{pmatrix} \cdot \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ V & W & X \end{pmatrix} \cdot \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ 0 & 0 & Y' \end{pmatrix} = \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ UV & UW & UXY' \end{pmatrix} .$$

An inequality may be formed with either 0, loop, abort or $\mathsf{loop} + \mathsf{abort}$ on the right-hand side, where the computations 0, loop and abort are represented by the matrices

$$0 = \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ 0 & 0 & 0 \end{pmatrix}, \qquad \mathsf{loop} = \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ 0 & \top & 0 \end{pmatrix}, \qquad \mathsf{abort} = \begin{pmatrix} \top & 0 & 0 \\ 0 & \top & 0 \\ \top & 0 & 0 \end{pmatrix} .$$

Using $p \cdot R \cdot q'$ or $p \cdot R \cdot 0$ on the left-hand side, we obtain the following seven kinds of claims:

$$\begin{array}{lllll}
(7) & p \cdot R \cdot q' \leq 0 & \Leftrightarrow UV \subseteq 0 \wedge & UW \subseteq 0 \wedge & UXY' \subseteq 0 \\
(6) & p \cdot R \cdot q' \leq \mathsf{abort} & \Leftrightarrow & UW \subseteq 0 \wedge & UXY' \subseteq 0 \\
(5) & p \cdot R \cdot q' \leq \mathsf{loop} & \Leftrightarrow UV \subseteq 0 \wedge & & UXY' \subseteq 0 \\
(4) & p \cdot R \cdot q' \leq \mathsf{loop} + \mathsf{abort} & \Leftrightarrow & & UXY' \subseteq 0 \\
(3) & p \cdot R \cdot 0 \ \leq 0 & \Leftrightarrow UV \subseteq 0 \wedge & UW \subseteq 0 & \\
(2) & p \cdot R \cdot 0 \ \leq \mathsf{abort} & \Leftrightarrow & UW \subseteq 0 & \\
(1) & p \cdot R \cdot 0 \ \leq \mathsf{loop} & \Leftrightarrow UV \subseteq 0 & &
\end{array}$$

Of particular interest are claims (1), (2) and (4) since the other ones are obtained as their conjunctions. Claim (1) expresses the absence of aborting executions from states in p, and claim (2) expresses the absence of infinite executions. Claim (4) is partial correctness and equivalent to $p \cdot R = p \cdot R \cdot q$ in the current model. For another example, the total correctness claim (6) expresses the absence of infinite executions in addition to partial correctness.

2.6 Summary

We have discussed five computation models that vary in the precision with which they can describe finite, infinite and aborting executions. These models support various kinds of correctness claims limited by their precision. Most claims take the form $p \cdot R \cdot q' \leq \mathsf{Z}$ for a constant Z depending on the model. The computation Z is either 0, loop, abort or $\mathsf{loop} + \mathsf{abort}$. The test p is the precondition, and the test q' is the complement of the postcondition or 0 if nothing is claimed about finite executions.

In the following we give a uniform algebraic description of all of these models and all of these correctness statements.

3 Relative Domain Semirings

Correctness statements in our models have the form $p \cdot R \cdot q' \leq \mathsf{Z}$ for a constant Z. In the partial and total correctness models $\mathsf{Z} = 0$ holds, and claims of this special form $p \cdot R \cdot q' \leq 0$ are well known in semirings with tests [29] or with a domain operation [8, 33]. In this section we generalise domain semirings to be able to encode claims of the form $p \cdot R \cdot q' \leq \mathsf{Z}$ for various values of Z.

3.1 Relative Domain

In relational computation models the domain $d(x)$ of the computation x is a test representing the set of states from which x has executions. Its Boolean complement, the antidomain $a(x)$, represents the set of states from which x has no executions. These operations satisfy the characteristic properties

$$x \leq d(y) \cdot x \Leftrightarrow d(x) \leq d(y)$$
$$a(y) \cdot x \leq 0 \Leftrightarrow a(y) \leq a(x)$$

which are at the centre of our generalisation. By Boolean algebra $d(x) \leq d(y)$ holds if and only if $a(y) \leq a(x)$ does. According to the first equivalence, $d(x)$ is the least test p such that $x \leq p \cdot x$, that is, all executions of x start in p. According to the second equivalence, $a(x)$ is the greatest test p such that $p \cdot x \leq 0$, that is, x to has no executions starting in p. We generalise these properties as follows:

$$x \leq d(y) \cdot x + \mathsf{Z} \Leftrightarrow d(x) \leq d(y)$$
$$a(y) \cdot x \leq \mathsf{Z} \Leftrightarrow a(y) \leq a(x)$$

Hence $d(x)$ is the least test p such that all executions of x start in p, except those executions that are in Z. This means that the executions of x that are in Z are ignored in the calculation of the domain. Similarly, $a(x)$ is the greatest test p such that x has no executions starting in p, except perhaps executions in Z. Setting $\mathsf{Z} = 0$ gives the characterisations of the usual domain and antidomain operations.

In the following we axiomatise the domain and antidomain operations relative to an element Z, which captures executions that are to be ignored. First, an idempotent semiring without right zero – simply called *semiring* in the remainder of this paper – is an algebraic structure $(S, +, \cdot, 0, 1)$ satisfying the axioms

$$\begin{array}{ccc} x + (y + z) = (x + y) + z & x(y + z) = xy + xz & x(yz) = (xy)z \\ x + y = y + x & (x + y)z = xz + yz & 1x = x \\ x + x = x & 0x = 0 & x1 = x \\ 0 + x = x & & \end{array}$$

where $x \cdot y$ is conventionally abbreviated as xy. In particular, the operation $+$ is idempotent and $x0 = 0$ is not an axiom. The *semilattice order* $x \leq y \Leftrightarrow x + y = y$ has least element 0, least upper bound $+$ and isotone operations $+$ and $\cdot$. A semiring is *bounded* if it has a greatest element $\top$ satisfying $x + \top = \top$.

In computation models, the operation $+$ represents non-deterministic choice, the operation $\cdot$ sequential composition, 0 the computation with no executions, 1 the program which does not change the state, $\top$ the computation with all possible executions, and $\leq$ the refinement relation. All computation models of Section 2 are semirings, and most of them do not satisfy the law $x0 = 0$.

A *relative domain semiring* is an algebraic structure $(S, +, \cdot, d, 0, 1, \mathsf{Z})$ such that the reduct $(S, +, \cdot, 0, 1)$ is a semiring and the axioms

$$\begin{array}{ccc} d(\mathsf{Z}) = 0 & d(x + y) = d(x) + d(y) & x \leq d(x)x + \mathsf{Z} \\ d(x) \leq 1 & d(d(x)y) = d(x)d(y) & d(xy) = d(xd(y)) \end{array}$$

are satisfied. Counterexamples generated by Mace4 show that none of these axioms follows from the remaining ones and the semiring axioms. Setting $\mathsf{Z} = 0$ gives the domain semiring axioms of [9], in which case $d(d(x)y) = d(x)d(y)$ follows from the remaining axioms.

Theorem 1. *Let S be a relative domain semiring and $x, y \in S$. Then*

- $(d(S), +, \cdot, 0, d(1))$ *is a bounded distributive lattice,*
- d *is isotone,*
- $d(0) = 0$*,*
- $d(d(x)) = d(x)$*,*
- $d(xy) \leq d(x) \leq d(1)$*,*
- $\mathsf{Z}x \leq \mathsf{Z}$*,*
- $x + \mathsf{Z} = d(x)x + \mathsf{Z}$*,*
- $d(x) = 0 \Leftrightarrow x \leq \mathsf{Z}$*,*
- $xy \leq \mathsf{Z} \Leftrightarrow xd(y) \leq \mathsf{Z}$*,*
- $x \leq d(y)x + \mathsf{Z} \Leftrightarrow d(x) \leq d(y)$*.*

In particular, $1+\mathsf{Z} = d(1)+\mathsf{Z}$ holds, but $d(1) = 1$ does not hold in general. Each computation model of Section 2 is a relative domain semiring where Z is any one of the values 0, loop, abort or $\mathsf{loop} + \mathsf{abort}$ available in the model. In these models the relative domain $d(x)$ is given by first omitting the executions of x that are in Z and then taking the usual domain. The element Z cannot be chosen arbitrarily; for example, $\mathsf{Z} = 1$ implies $d(x) \leq d(1) = d(\mathsf{Z}) = 0$ and therefore $x \leq d(x)x + \mathsf{Z} = 1$ which does not hold in any model given in Section 2.

A *relative antidomain semiring* is an algebraic structure $(S, +, \cdot, a, d, 0, 1, \mathsf{Z})$ such that the reduct $(S, +, \cdot, 0, 1)$ is a semiring, $d(x) = a(a(x))$ and the axioms

$$a(\mathsf{Z}) = 1 \qquad a(x + y) = a(x)a(y) \qquad a(x)x \leq \mathsf{Z}$$
$$a(x)d(x) = 0 \qquad a(d(x)y) = a(x) + a(y) \qquad a(xy) = a(xd(y))$$

are satisfied. Counterexamples generated by Mace4 show that none of these axioms follows from the remaining ones and the semiring axioms. Setting $\mathsf{Z} = 0$ gives axioms which are equivalent to the antidomain axioms of Boolean domain semirings [9].

Theorem 2. *Let S be a relative antidomain semiring and $x, y, z \in S$. Then*

- $(S, +, \cdot, d, 0, 1, \mathsf{Z})$ *is a relative domain semiring,*
- $(a(S), +, \cdot, a, 0, 1)$ *is a Boolean algebra with complement a,*
- $a(S) = d(S)$*,*
- $d(a(x)) = a(d(x)) = a(x)$*,*
- $a(x) \leq a(xy)$*,*
- $a(x) = 1 \Leftrightarrow x \leq \mathsf{Z}$*,*
- $d(x)y \leq z \Leftrightarrow d(x)y \leq d(x)z$*,*
- $x \leq y + \mathsf{Z} \Leftrightarrow x \leq d(x)y + \mathsf{Z}$*,*
- $ya(z) \leq a(x)y \Leftrightarrow ya(z) = a(x)ya(z) \Leftrightarrow d(x)ya(z) = 0$*,*
- $d(x)y \leq yd(z) \Leftrightarrow d(x)y = d(x)yd(z) \Leftrightarrow d(x)ya(z) = d(x)y0 \Leftrightarrow$ $d(x)ya(z) \leq y0$*,*
- $a(y)x \leq \mathsf{Z} \Leftrightarrow a(y) \leq a(x)$*.*

In particular, a is antitone, $a(1) = 0$ and $d(1) = a(0) = 1$. Using the Boolean complement of the relative domain, each computation model of Section 2 is a relative antidomain semiring where Z is any of the values 0, loop, abort or $\mathsf{loop}+\mathsf{abort}$ available in the model and different from $\top$. We can therefore represent tests and their Boolean complements by domain elements $d(x)$ and their antidomain $a(x)$. The correctness claim $p \cdot R \cdot q' \leq \mathsf{Z}$ is thus expressed as $d(x)ya(z) \leq \mathsf{Z}$ in a relative antidomain semiring. By Theorem 2 this is equivalent to $d(x) \leq a(ya(z))$.

3.2 Relative Modal Operators

Domain and antidomain semirings give rise to modal diamond and box operators. We generalise them to relative (anti)domain semirings.

In a relative domain semiring the binary *diamond* operator is defined by $|x\rangle y = d(xy)$, which is the same as $d(xd(y))$. This means that its second argument is effectively a test p, and $|x\rangle p$ represents the states from which there is an execution of x that is not in Z and that terminates in p if it terminates normally. In particular, $|x\rangle p$ contains the starting states of all infinite and aborting executions of x if $\mathsf{Z} = 0$, but these executions are filtered out if $\mathsf{Z} = \mathsf{loop} + \mathsf{abort}$. The diamond operator satisfies many properties known from the unrelativised setting.

Theorem 3. *Let S be a relative domain semiring and $x, y, z \in S$ and $p, q \in d(S)$. Then*

- $|\cdot\rangle\cdot$ *is isotone,*
- $|x + y\rangle z = |x\rangle z + |y\rangle z$,
- $|x\rangle y + z = |x\rangle y + |x\rangle z$,
- $|xy\rangle z = |x\rangle(yz) = |x\rangle|y\rangle z$,
- $|x\rangle(pq) = |x\rangle p \cdot |x\rangle q$,
- $|px\rangle y = p|x\rangle y$,
- $p|x\rangle q \leq \mathsf{Z} \Leftrightarrow pxq \leq \mathsf{Z}$,
- $|x\rangle q \leq p \Leftrightarrow xq \leq px + \mathsf{Z}$.

In a relative antidomain semiring the dual *box* operator is defined by $|x]y = a(xa(y))$. Again its second argument is effectively a test p, and $|x]p$ represents the states from which all executions are in Z or terminate in p. Also the box operator satisfies many properties known from the unrelativised setting.

Theorem 4. *Let S be a relative antidomain semiring and $x, y, z \in S$ and $p, q \in d(S)$. Then*

- $|x]y = a(|x\rangle a(y))$,
- $|x\rangle y = a(|x]a(y))$,
- $|x]\cdot$ *is isotone,*
- $|\cdot]x$ *is antitone,*
- $|x + y]z = |x]z \cdot |y]z$,
- $|xy]z = |x]|y]z$,
- $|x](pq) = |x]p \cdot |x]q$,
- $|px]y = a(p) + |x]y$,
- $|x\rangle q \leq p \Leftrightarrow a(p)xq \leq \mathsf{Z}$,
- $p \leq |x]q \Leftrightarrow pxa(q) \leq \mathsf{Z} \Rightarrow px \leq xq + \mathsf{Z}$.

Consequently, the correctness claim $p \cdot x \cdot q' \leq \mathsf{Z}$ with tests p and q is formalised by $p \leq |x]q$.

It is known that the box operator corresponds to wlp in partial correctness models [33] and to wp in general correctness [34] and total correctness models [31]. These cases are captured by using $\mathsf{Z} = 0$ in our setting. We furthermore find that

- with $\mathsf{Z} = 0$ box corresponds to a variant of wp, which avoids aborting executions in addition to infinite ones, for extended designs and the model of Section 2.5,
- with $\mathsf{Z} = \mathsf{loop}$ box corresponds to wlp in general correctness models, and to a variant of wlp, which avoids aborting executions, for extended designs and the model of Section 2.5,
- with $\mathsf{Z} = \mathsf{abort}$ box corresponds to wp in the model of Section 2.5,
- with $\mathsf{Z} = \mathsf{loop} + \mathsf{abort}$ box corresponds to wlp in the model of Section 2.5.

Similar variants of wp are observed in [20, 21] and related to different execution methods without a unified treatment; see [37] for variants of wlp.

4 Iteration

In this section we give axioms for operations that describe iteration in various computation models. They facilitate a unified semantics of while-programs as we show in Section 5.3.

A *Kleene algebra* [28] is a semiring expanded by an operation * satisfying the axioms

$$\begin{array}{ll} 1 + yy^* \leq y^* & z + yx \leq x \Rightarrow y^*z \leq x \\ 1 + y^*y \leq y^* & z + xy \leq x \Rightarrow zy^* \leq x \end{array}$$

It follows that y^*z is the least fixpoint of $\lambda x.yx + z$ and that zy^* is the least fixpoint of $\lambda x.xy + z$. The Kleene star describes finite iteration, but is not appropriate for models with infinite executions which require other fixpoints. We therefore use the following, more general structure.

An *itering* [16] is a semiring expanded by an operation $^\circ$ satisfying the axioms

$$\begin{array}{ll} (x+y)^\circ = (x^\circ y)^\circ x^\circ & zx \leq yy^\circ z + w \Rightarrow zx^\circ \leq y^\circ(z + wx^\circ) \\ (xy)^\circ = 1 + x(yx)^\circ y & xz \leq zy^\circ + w \Rightarrow x^\circ z \leq (z + x^\circ w)y^\circ \end{array}$$

The equations are the sumstar and productstar axioms of [6]. The other two axioms generalise simulation properties such as $zx \leq yz \Rightarrow zx^\circ \leq y^\circ z$, which is known in Kleene algebra and omega algebra [5]. Its dual $xz \leq zy \Rightarrow x^\circ z \leq zy^\circ$ holds in Kleene algebras, but not in other target models, whence we weaken its consequent. Properties of the operation $^\circ$ are shown in the following result.

Theorem 5. *Let S be an itering and $x, y, z \in S$. Then $^\circ$ is isotone and*

- $0^\circ = 1 \leq (x0)^\circ = 1 + x0 \leq x^\circ$,
- $x^\circ = x^\circ x^\circ = (x^\circ x)^\circ = 1 + xx^\circ = 1 + x^\circ x$,
- $x \leq xx^\circ = x^\circ x \leq x^\circ$,
- $x^\circ \leq x^\circ 1^\circ = 1^\circ x^\circ = (1 + x)^\circ = x^{\circ\circ} = x^{\circ\circ\circ}$,
- $x^\circ y^\circ \leq (x + y)^\circ \leq (x^\circ y^\circ)^\circ = (y^\circ x^\circ)^\circ = x^\circ(y^\circ x^\circ)^\circ$,
- $(yx^\circ)^\circ = y^\circ + y^\circ yxx^\circ(yx^\circ)^\circ = (yy^\circ x^\circ)^\circ$,
- $x(yx)^\circ = (xy)^\circ x$.

Moreover, $y^\circ z$ is a fixpoint of $\lambda x.yx + z$ and zy° is a fixpoint of $\lambda x.xy + z$.

The following result gives six models of iterings which cover all computation models of Section 2 [16].

Theorem 6. *Iterings have the following models:*

1. *Every Kleene algebra is an itering using* $x^\circ = x^*$*.*
2. *Every omega algebra [5] is an itering using* $x^\circ = x^\omega 0 + x^*$*.*
3. *Every omega algebra with* $\top x = \top$ *is an itering using* $x^\circ = x^\omega + x^*$*.*
4. *Every demonic refinement algebra [39] is an itering using* $x^\circ = x^\omega$*.*
5. *Extended designs [17, 22] form an itering using* $x^\circ = d(x^\omega)\mathsf{loop} + x^*$*.*
6. *The model of Section 2.5 forms an itering using* $x^\circ = n(x^\omega)\mathsf{loop} + x^*$*, where* $n(x)$ *captures the infinite executions of* x *as a test [16].*

A *modal itering* is a structure $(S, +, \cdot, a, d, ^*, ^\circ, 0, 1, \mathsf{Z})$ such that the reduct $(S, +, \cdot, a, d, 0, 1, \mathsf{Z})$ is a relative antidomain semiring, the reduct $(S, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and the reduct $(S, +, \cdot, ^\circ, 0, 1)$ is an itering. The operations * and $^\circ$ may be identical as in Kleene algebras or different as in the other models.

5 Correctness Statements

The box operator of Section 3.2 expresses various kinds of preconditions depending on the model and the value of the constant Z. In this section we give an axiomatic description of such preconditions suitable, in particular, for total correctness claims. This extends our previous work on preconditions for partial correctness [19]. We then use the preconditions to obtain a correctness calculus, pre-post specifications and loop refinement rules, all of which uniformly apply to the computation models and correctness statements of Section 2.

5.1 Tests

Preconditions are represented as tests, which we introduce first. A *test algebra* [16, 19] is a structure $(S, \cdot, ')$ satisfying the axioms

$$x'(y'z') = (x'y')z' \qquad x' = (x''y')'(x''y'')'$$
$$x'y' = y'x' \qquad x'y' = (x'y')''$$

They are derived from Huntington's axioms and make $S' = \{x' \mid x \in S\}$ a Boolean algebra with meet $\cdot$, complement $'$, order $x' \leq y' \Leftrightarrow x'y' = x'$, least element $0 = x'x''$ for any x, and greatest element $1 = 0'$. The operation $x' + y' = (x''y'')'$ is the join in S'. The extension $(S, +, \cdot, ', 0, 1)$ is also called a test algebra; elements of S' are *tests*. This axiomatisation imposes fewer constraints than antidomain semirings, which induce tests as the following result shows, without introducing a separate sort for tests.

Theorem 7. *Let* S *be a relative antidomain semiring. Then* $(S, +, \cdot, a, 0, 1)$ *is a test algebra with* $S' = d(S)$*.*

A test algebra is *complete* if S' is a complete Boolean algebra. Then every set of tests has a supremum in S' and the meet operation $\cdot$ on S' distributes over suprema.

5.2 Preconditions

A *precondition algebra* $(S, \cdot, «, ')$ is a test algebra $(S, \cdot, ')$ expanded with a binary operation « satisfying the axioms

$$x«q = (x«q)'' \qquad xy«q = x«(y«q)$$
$$p«q = (pq')' \qquad x«pq = (x«p)(x«q)$$

for $x, y \in S$ and $p, q \in S'$.

The first axiom states that the result of « is a test, making « an operation which takes an element and a test and yields a test. The axiom $p«q = (pq')' = p' + q$ reduces the precondition of tests to an implication; it is slightly stronger than our original in [19]. The remaining axioms express the effect of « on the sequential composition of elements and the conjunction of postconditions.

Theorem 8. *Let S be a precondition algebra and $x, y \in S$ and $p, q, r \in S'$. Then*

- $x«\cdot$ *is isotone,*
- $p(x«q) = p(px«q)$,
- $px«q = p' + (x«q)$,
- $xp«q = xp«pq$,
- $p(p«q) = pq$,
- $p'(p«q) = p'$,
- $0«q = 1$,
- $1«q = q$,
- $xy«1 \leq x«1$,
- $x«q \leq x«1 \leq 1$,
- $p \leq x«q \wedge q \leq y«r \Rightarrow p \leq xy«r$.

As the following result shows, the box operator expresses preconditions. Thus wp, wlp and their variants discussed in Section 3.2 are instances of «.

Theorem 9. *Let S be a relative antidomain semiring and $x, y \in S$ and $p, q \in d(S)$. Then S is a precondition algebra with $x«q = |x]q$. Moreover,*

- $(x + y)«q = (x«q) \cdot (y«q)$,
- $\cdot«q$ *is antitone,*
- $(x«q)x + \mathsf{Z} = (x«q)xq + \mathsf{Z}$,
- $(x«q)xq' \leq \mathsf{Z}$,
- $p \leq x«q \Leftrightarrow pxq' \leq \mathsf{Z}$,
- $x«1 = 1 \Leftrightarrow x0 \leq \mathsf{Z}$.

In a complete precondition algebra S, the *progressively bounded states* of an element $x \in S$ are given by $b(x) = \sup\{x^n«0 \mid n \in \mathbb{N}\}$. The test $b(x)$ describes the states which have an upper bound on the lengths of the emerging x-transition paths. This means that for every state in $b(x)$ there is a bound n such that x can be iterated at most n times starting from the state; the bound n may depend on the state.

This should be contrasted with the progressively finite states characterised, for example, by the convergence operation of [34]. These require the absence of infinite transition paths without giving bounds on the lengths of finite paths, and coincide with the progressively bounded states for deterministic programs.

5.3 While-Programs

A *while algebra* $(S, \lhd\rhd, \cdot, «, \star, ')$ is a precondition algebra $(S, \cdot, «, ')$ expanded with a ternary operation $\lhd\rhd$ and a binary operation $\star$ satisfying the axioms

$$(x \lhd p \rhd y)«q = p(x«q) + p'(y«q)$$
$$(p \star x)«q = (x(p \star x) \lhd p \rhd 1)«q$$

for $x, y \in S$ and $p, q \in S'$. The element $x \lhd p \rhd y$ represents the conditional statement if p then x else y and the corresponding axiom characterises the two branches under a postcondition; see [24, 26] for more comprehensive axiomatisations. The element $p \star x$ represents the while loop while p do x and the corresponding axiom describes its fixpoint unfolding, again under a postcondition. Both axioms are equations of tests, weakening our original axioms in [19]. As we show below, they hold in a wide range of computation models.

Theorem 10. *Let S be a while algebra and $x, y \in S$ and $p, q, r \in S'$. Then*

- $p((x \lhd p \rhd y)«q) = p(x«q)$,
- $p'((x \lhd p \rhd y)«q) = p'(y«q)$,
- $pq \leq x«r \wedge p'q \leq y«r \Rightarrow q \leq (x \lhd p \rhd y)«r$,
- $p((p \star x)«q) = p(x«(p \star x)q)$,
- $p'((p \star x)«q) = p'q$,
- $p' \leq (p \star x)«p' \leq (p \star x)«1$,
- $q \leq (p \star x)«1 \Leftrightarrow pq \leq (p \star x)«1$.

In Section 5.4, the test $\ell = (1 \star 1)«1$ helps us to treat total correctness claims and claims which do not involve termination in a uniform way. The element $1 \star 1$ represents the endless loop while true do skip. It establishes the postcondition true if and only if the infinite executions are ignored. Claims which do not involve termination are thus obtained in instances with $\ell = 1$, whereas instances with $\ell = 0$ yield total correctness. In particular, a convenient way to obtain partial correctness is to add the axiom $x«1 = 1$, a characteristic property of wlp [10].

As the following result shows, the operations $\lhd\rhd$ and $\star$ can be defined in modal iterings, hence in all models given in Section 2. This gives a unified semantics of while-programs.

Theorem 11. *Let S be a modal itering and $x, y \in S$ and $p \in d(S)$. Then S is a while algebra with $x \lhd p \rhd y = px + a(p)y$ and $p \star x = (px)^\circ a(p)$. In particular, $\ell = a(1^\circ 0)$.*

Consider a while algebra S, a subset $A \subseteq S$ of atomic programs and a subset $T \subseteq S'$ of atomic tests. We assume that $1 \in A$ and $0 \in T$, that is, skip is an atomic program and false is an atomic test. There are no further requirements on A and T; in concrete models they typically contain basic statements such as assignments and basic conditions.

Test expressions are constructed from atomic tests by the operations $'$ for negation and $\cdot$ for conjunction. Hence they are tests and closed under 0, 1,

finite sums and finite products. *While-programs* are constructed from atomic programs and test expressions by the operations $\cdot$ for sequential composition, $\lhd\rhd$ for conditionals and $\star$ for while loops. Hence they are closed under 1 and finite products. *Assertions* are test expressions extended by preconditions; they are constructed from test expressions and while-programs by the operations $'$ for negation, $\cdot$ for conjunction and $\ll$ for preconditions. Hence they are tests and closed under 0, 1, ℓ, finite sums and finite products.

5.4 Correctness Calculus

A *correctness algebra* is a complete while algebra satisfying the additional axiom

$$pq \leq x \ll q \Rightarrow q\ell \leq (p \star x) \ll p'q$$

for $x \in S$ and $p, q \in S'$. It expresses soundness of the partial correctness rule for while loops in the correctness calculus.

A *correctness statement* $p\{x\}q$ is composed of a while-program x and two assertions p and q. The statement $p\{x\}q$ is *valid* if and only if $p \leq x \ll q$. Its meaning depends on the model and the interpretation of the precondition operation $\ll$. In some models, $p \leq x \ll q$ amounts to partial correctness, that is, all finite executions of x starting in p establish the postcondition q. In other models, $p \leq x \ll q$ amounts to total correctness, which additionally requires that all executions of x starting in p are finite. In yet other models, $p \leq x \ll q$ requires that no execution of x starting in p aborts.

To *derive* correctness statements, we use a calculus with the following rules, for atomic program z, while-programs x and y, test expression p, assertions q, r, s and t, and tests t_i:

$$\text{(atom)} \quad \frac{}{z \ll q\{z\}q}$$

$$\text{(seq)} \quad \frac{q\{x\}r \quad r\{y\}s}{q\{xy\}s}$$

$$\text{(cond)} \quad \frac{pq\{x\}r \quad p'q\{y\}r}{q\{x \lhd p \rhd y\}r}$$

$$\text{(while)} \quad \frac{pq\{x\}q \quad q \leq \ell + t_{<\infty} \quad \forall n \in \mathbb{N} : t_n pq\{x\}\ell + t_{<n}}{q\{p \star x\}p'q}$$

$$\text{(cons)} \quad \frac{q \leq r \quad r\{x\}s \quad s \leq t}{q\{x\}t}$$

The rule for while loops is abstracted from the Hoare calculus for total correctness [1]. The test $t_{<n}$ is defined by $t_{<n} = \sup\{t_i \mid 0 \leq i < n\}$ for $n \in \mathbb{N} \cup \{\infty\}$. If $\ell = 0$, the sequence of tests t_i describes the bound function; each test t_n represents a set of states from which the loop terminates after at most n iterations, the inequality $q \leq \ell + t_{<\infty}$ expresses that the bound is non-negative while the invariant q holds, and $t_n pq\{x\}\ell + t_{<n}$ expresses that every iteration decreases the

bound. If $\ell = 1$, the premises simplify to $pq\{x\}q$, which expresses that the loop invariant q is preserved by the loop body. The rule concludes that the invariant is preserved by the while loop.

Theorem 12. *The calculus is sound, that is, only valid correctness statements can be derived. Let $(p \star x)\text{«}1 \leq \ell + b(px)$ for every test expression p and while-program x. Then the calculus is complete, that is, every valid correctness statement can be derived.*

The condition for completeness is satisfied if the body px of a while loop is boundedly non-deterministic in total correctness models. Namely, $(p \star x)\text{«}1$ contains the states from which the loop $p \star x$ has only finite executions, and they have to be among the progressively bounded states $b(px)$ if $\ell = 0$.

As usual, completeness is relative to having all true inequalities $p \leq q$ available in the calculus. The bound function $t_i = (px)^i\text{«}0$ is used in the completeness proof. Termination after a given number of iterations is described by domain elements in [18].

Our calculus unifies and generalises previous algebraic calculi for partial, total and general correctness [13, 14, 19, 29, 33, 34]. In particular, it applies to further computation models and facilitates total correctness claims by algebraically representing the bound function.

Example 13. We prove correctness of a program for integer division along the lines of [1], namely

$$(q, r := 0, x) \;;\; \mathsf{while}\ (r \geq y)\ \mathsf{do}\ (q, r := q + 1, r - y)$$

with four variables q, r, x, y ranging over $\mathbb{N}$. It computes the quotient q and the remainder r of the division of x by y. Using tests p_1, p_2, p_3, t_n and assignments z_1, z_2 defined by

$$\begin{array}{lll} p_1 = (y > 0) & p_3 = (r \geq y) & z_1 = (q, r := 0, x) \\ p_2 = (x = q \times y + r) & t_n = (r = n) & z_2 = (q, r := q + 1, r - y) \end{array}$$

the program is abstractly expressed as $z_1(p_3 \star z_2)$ and the following correctness statements hold:

- $p_1\{z_1\}p_1p_2$ since z_1 does not affect y and $r = 0 \times y + r$ holds,
- $p_3p_1p_2\{z_2\}p_1p_2$ since z_2 does not affect y and $x = q \times y + r$ implies $x = (q + 1) \times y + (r - y)$, and
- $t_np_1p_3\{z_2\}t_{<n}$ for each $n \in \mathbb{N}$ since $n = r \geq y > 0$ implies $r - y = n - y < n$; more precisely, t_{n-y} is established.

Furthermore $t_{<\infty} = 1$, whence we derive

$$\dfrac{\dfrac{p_1\{z_1\}p_1p_2 \qquad \dfrac{p_3p_1p_2\{z_2\}p_1p_2 \qquad p_1p_2 \leq \ell + t_{<\infty} \qquad \dfrac{\forall n \in \mathbb{N} : t_np_1p_3\{z_2\}t_{<n}}{\forall n \in \mathbb{N} : t_np_3p_1p_2\{z_2\}\ell + t_{<n}}}{p_1p_2\{p_3 \star z_2\}p_3'p_1p_2}}{p_1\{z_1(p_3 \star z_2)\}p_3'p_1p_2}}{p_1\{z_1(p_3 \star z_2)\}p_3'p_2}$$

using the loop invariant $p_1 p_2$ and the bound function t_n. Hence the precondition $y > 0$ suffices to establish the postcondition $x = q \times y + r$ and $r < y$, by which q is the quotient and r is the remainder of the division of x by y.

At the same time, this derivation establishes total correctness: the program terminates when started in a state with $y > 0$. Moreover, it establishes that the program does not abort when started in such a state. These consequences hold because the assumed correctness statements and the derivation are valid in all our computation models and for any $\mathsf{Z} \in \{0, \mathsf{loop}, \mathsf{abort}, \mathsf{loop} + \mathsf{abort}\} \setminus \{\top\}$.

The following result shows how to define correctness statements in modal iterings subjected to two additional axioms.

Theorem 14. *Let S be a modal itering – which is a test algebra, a precondition algebra and a while algebra according to Theorems 7, 9 and 11 – such that the test algebra is complete and*

$$x\mathsf{Z} \leq x0 + \mathsf{Z} \qquad |x^*]y \leq |\ell x^\circ]y$$

for each $x, y \in S$. Then S is a correctness algebra. Moreover, the induction laws

$$p \leq |x]p \Rightarrow p \leq |x^*]p$$
$$p \leq |x]p \Rightarrow \ell p \leq |x^\circ]p$$

hold for $x \in S$ and $p \in d(S)$.

In particular, $p\{x\}q$ is valid if and only if $p \leq |x]q$. The axiom $x\mathsf{Z} \leq x0 + \mathsf{Z}$ separates the executions of x in the composition $x\mathsf{Z}$. All finite executions of x reach Z; this part is subsumed by Z. The executions of x which do not reach Z because they are infinite or abort are subsumed by $x0$. The axiom $|x^*]p \leq |\ell x^\circ]p$ expresses that for claims not involving termination, whence $\ell = 1$, iteration reduces to finite iteration as infinite executions are ignored.

5.5 Pre-post Specifications

Consider the correctness statement $p\{x\}q$. For given x and q, the test $x«q$ is the greatest precondition that suffices to establish the postcondition q; all tests p with $p \leq x«q$ are sufficient, too. Another viewpoint is obtained for given p and q: the pre-post specification $p \dashv q$ [30, 35, 36, 39] is the greatest computation for which p suffices to establish q; all computations x with $x \leq p \dashv q$ satisfy $p \leq x«q$ as well. Pre-post specifications can therefore be introduced by a Galois connection.

A *pre-post algebra* is an algebraic structure $(S, +, \cdot, «, \dashv, ', 0, 1, \top)$ such that the reduct $(S, +, \cdot, 0, 1, \top)$ is a bounded semiring, the reduct $(S, +, \cdot, «, ', 0, 1)$ is a precondition algebra, and the operation $\dashv$ satisfies

$$x \leq p \dashv q \Leftrightarrow p \leq x«q$$

for $x \in S$ and $p, q \in S'$. This axiom is an order-reversing Galois connection between S and S'.

Theorem 15. *Let S be a pre-post algebra and $x, y \in S$ and $p, q, r, s \in S'$. Then*

- $\cdot \text{«} q$ *is antitone,*
- $\cdot \dashv q$ *is antitone,*
- $p \dashv \cdot$ *is isotone,*
- $(x + y) \text{«} q = (x \text{«} q) \cdot (y \text{«} q)$,
- $pq \dashv r = (p \dashv r) + (q \dashv r)$,
- $p \dashv (q + r) = (p \dashv q) + (p \dashv r)$,
- $x \le (x \text{«} q) \dashv q$,
- $p \le (p \dashv q) \text{«} q$,
- $(p \dashv q) r = (p \dashv qr) r = (p \dashv (q + r')) r$,
- $r(p \dashv q) = r(rp \dashv q) = r((r' + p) \dashv q)$,
- $p \dashv q = (1 \dashv q) + p' \top$,
- $p(p \dashv q) = p(1 \dashv q)$,
- $p'(p \dashv q) = p' \top$,
- $(1 \dashv q) \text{«} q = 1 \le p \dashv p$,
- $0 \dashv q = \top$,
- $q \le r \Rightarrow (p \dashv q)(r \dashv s) \le p \dashv s$,
- $(p \dashv q)(q \dashv r) \le p \dashv r$,
- $(p \dashv p)(p \dashv q) = (p \dashv q)(q \dashv q) = p \dashv q$,
- $(p \dashv p)(p \dashv p) = p \dashv p$,
- $x \text{«} 1 = 1 \Leftrightarrow x \le 1 \dashv 1$,
- $x \le pq \dashv r \Leftrightarrow px \le q \dashv r$.

Example 16. In a structure which is both a pre-post algebra and a correctness algebra, such as every model in Section 2, the correctness rule for while loops translates as

$$x \le pq \dashv q \ \wedge \ q \le \ell + t_{<\infty} \ \wedge \ (\forall n \in \mathbb{N} : x \le t_n pq \dashv \ell + t_{<n}) \ \Rightarrow \ p \star x \le q \dashv p'q \ .$$

This rule introduces a while loop by refining a pre-post specification. We give two instances in the computation model of Section 2.5. The first instance is a partial correctness rule obtained by setting $\mathsf{Z} = \top 0$, whence $\ell = 1$ by Theorem 9 and therefore

$$x \le pq \dashv q \ \Rightarrow \ p \star x \le q \dashv p'q$$

because $pq \dashv q \le t_n pq \dashv 1$ by Theorem 15. The second instance is a total correctness rule obtained by setting $\mathsf{Z} = 0$, whence $\ell = 0$. Using $q = t_{<\infty}$, a consequence of the rule is

$$r \le t_{<\infty} \ \wedge \ (\forall n \in \mathbb{N} : x \le t_n p \dashv t_{<n}) \ \Rightarrow \ p \star x \le r \dashv 1 \ .$$

Both instances are obtained in the same model, with different values of Z, and therefore apply to the same while loop $p \star x$. This achieves a separation of the invariant q and the termination condition r as advocated by [12, 13]. Moreover, by using $\mathsf{Z} = \mathsf{loop}$ a further separation can be obtained to specify the states from which the execution of the loop does not abort independently of q and r.

The following result shows how to define pre-post specifications in antidomain semirings subjected to two additional axioms taken from [15]. They are equivalent to $x0 \leq y \Leftrightarrow x \leq y + \mathsf{H}$ and introduce the element H representing the program havoc, which is the greatest program that has only finite executions.

Theorem 17. *Let S be a bounded relative antidomain semiring, which is a precondition algebra according to Theorem 9. Let $\mathsf{H} \in S$ such that*

$$\mathsf{H}0 = 0 \qquad x \leq x0 + \mathsf{H}$$

for each $x \in S$. Then S is a pre-post algebra with $p \dashv q = \mathsf{Z} + a(p)\top + \mathsf{H}q$.

6 Conclusion

Five computation models with varying representations of finite, infinite and aborting executions support similar correctness statements. These are captured uniformly by a relativisation of domain semirings and, more generally, by an algebra for preconditions. It facilitates the definition of correctness claims, their calculus and pre-post specifications in a unified way for various models and correctness statements.

Future work concerns a question raised by a referee, namely whether the approach can be extended to general refinement algebra [39] which models dual non-determinism.

Acknowledgement. I thank Jeremy Gibbons and the anonymous referees for providing helpful comments.

References

1. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd edn. Springer (2009)
2. de Bakker, J.W.: Semantics and termination of nondeterministic recursive programs. In: Michaelson, S., Milner, R. (eds.) Automata, Languages and Programming: Third International Colloquium, pp. 435–477. Edinburgh University Press (1976)
3. Berghammer, R., Zierer, H.: Relational algebraic semantics of deterministic and nondeterministic programs. Theor. Comput. Sci. 43, 123–147 (1986)
4. Broy, M., Gnatz, R., Wirsing, M.: Semantics of Nondeterministic and Noncontinuous Constructs. In: Bauer, F.L., Broy, M. (eds.) Program Construction. LNCS, vol. 69, pp. 553–592. Springer, Heidelberg (1979)
5. Cohen, E.: Separation and Reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
6. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
7. De Carufel, J.-L., Desharnais, J.: Demonic Algebra with Domain. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 120–134. Springer, Heidelberg (2006)

8. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Transactions on Computational Logic 7(4), 798–833 (2006)
9. Desharnais, J., Struth, G.: Internal axioms for domain semirings. Sci. Comput. Program. 76(3), 181–203 (2011)
10. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
11. Dunne, S.: Recasting Hoare and He's Unifying Theory of Programs in the context of general correctness. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods. Electronic Workshops in Computing. The British Computer Society (2001)
12. Dunne, S.E., Hayes, I.J., Galloway, A.J.: Reasoning about Loops in Total and General Correctness. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 62–81. Springer, Heidelberg (2010)
13. Guttmann, W.: General Correctness Algebra. In: Berghammer, R., Jaoua, A.M., Möller, B. (eds.) RelMiCS/AKA 2009. LNCS, vol. 5827, pp. 150–165. Springer, Heidelberg (2009)
14. Guttmann, W.: Partial, Total and General Correctness. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 157–177. Springer, Heidelberg (2010)
15. Guttmann, W.: Unifying Recursion in Partial, Total and General Correctness. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 207–225. Springer, Heidelberg (2010)
16. Guttmann, W.: Algebras for iteration and infinite computations (submitted, 2011)
17. Guttmann, W.: Extended designs algebraically. Sci. Comput. Program. (to appear, 2012)
18. Guttmann, W., Möller, B.: Normal design algebra. Journal of Logic and Algebraic Programming 79(2), 144–173 (2010)
19. Guttmann, W., Struth, G., Weber, T.: Automating Algebraic Methods in Isabelle. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 617–632. Springer, Heidelberg (2011)
20. Harel, D.: First-Order Dynamic Logic. LNCS, vol. 68. Springer, Heidelberg (1979)
21. Harel, D.: On the total correctness of nondeterministic programs. Theor. Comput. Sci. 13(2), 175–192 (1981)
22. Hayes, I.J., Dunne, S.E., Meinicke, L.: Unifying Theories of Programming That Distinguish Nontermination and Abort. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 178–194. Springer, Heidelberg (2010)
23. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580/583 (1969)
24. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Commun. ACM 30(8), 672–686 (1987)
25. Hoare, C.A.R., He, J.: Unifying theories of programming. Prentice Hall Europe (1998)
26. Jackson, M., Stokes, T.: Semigroups with if-then-else and halting programs. International Journal of Algebra and Computation 19(7), 937–961 (2009)
27. Jacobs, D., Gries, D.: General correctness: A unification of partial and total correctness. Acta Inf. 22(1), 67–83 (1985)
28. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
29. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Transactions on Computational Logic 1(1), 60–76 (2000)

30. Meertens, L.: Abstracto 84: The next generation. In: Martin, A.L., Elshoff, J.L. (eds.) ACM 1979: Proceedings of the 1979 Annual Conference, pp. 33–39. ACM Press (1979)
31. Möller, B.: The Linear Algebra of UTP. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 338–358. Springer, Heidelberg (2006)
32. Möller, B.: Kleene getting lazy. Sci. Comput. Program. 65(2), 195–214 (2007)
33. Möller, B., Struth, G.: Algebras of modal operators and partial correctness. Theor. Comput. Sci. 351(2), 221–239 (2006)
34. Möller, B., Struth, G.: wp Is wlp. In: MacCaull, W., Winter, M., Düntsch, I. (eds.) RelMiCS 2005. LNCS, vol. 3929, pp. 200–211. Springer, Heidelberg (2006)
35. Morgan, C.: The specification statement. ACM Trans. Progr. Lang. Syst. 10(3), 403–419 (1988)
36. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Program. 9(3), 287–306 (1987)
37. Morris, J.M.: Varieties of weakest liberal preconditions. Inf. Process. Lett. 25(3), 207–210 (1987)
38. Nelson, G.: A generalization of Dijkstra's calculus. ACM Trans. Progr. Lang. Syst. 11(4), 517–561 (1989)
39. von Wright, J.: Towards a refinement algebra. Sci. Comput. Program. 51(1-2), 23–45 (2004)

Dependently Typed Programming Based on Automated Theorem Proving

Alasdair Armstrong, Simon Foster, and Georg Struth

Department of Computer Science,
University of Sheffield, UK
{a.armstrong,s.foster,g.struth}@dcs.shef.ac.uk

Abstract. Mella is a minimalistic dependently typed programming language and interactive theorem prover implemented in Haskell. Its main purpose is to investigate the effective integration of automated theorem provers in this pure and simple setting. Such integrations are essential for supporting program development in dependently typed languages. We integrate the equational theorem prover Waldmeister and test it on more than 800 proof goals from the TPTP library. In contrast to previous approaches, the reconstruction of Waldmeister proofs within Mella is quite robust and does not generate a significant overhead to proof search. Mella thus yields a template for integrating more expressive theorem provers in more sophisticated languages.

1 Introduction

Dependently typed programming languages (DTPLs) such as Adga [11] or Epigram [22] are currently receiving considerable attention. By combining the elegance of functional programming with more expressive type systems, they introduce a new mathematically principled style of program development. In contrast to traditional functional programming, types are powerful enough to support detailed specifications of program properties. This however requires type-level reasoning that is no longer decidable. DTPLs are at the same time interactive theorem proving (ITP) systems similar to Nuprl [17] or Coq [9]. On the one hand this supports developing programs that are correct by construction. On the other hand it puts an additional burden on programmers.

To support program development at an appropriate level of abstraction, it is essential that programmers can focus on more creative aspects of proofs, whereas trivial and routine proof tasks are automated. Yet how can this be achieved?

Traditionally, automation is obtained in ITP systems by implementing large libraries of tactics, internally verified solvers and sophisticated simplification techniques, or by using external solvers as oracles. More recently, external automated theorem proving (ATP) systems, satisfiability modulo theories (SMT) solvers and other decision procedures have been integrated in a more trustworthy way into ITP systems by internally reconstructing proofs provided by the external tools. This approach is preferable because ITP systems are usually much more simple and transparent than ATP and SMT systems which depend

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 220–240, 2012.
© Springer-Verlag Berlin Heidelberg 2012

on approximations and elaborate heuristics. A prime example for this approach is Isabelle's Sledgehammer tool (cf. [10]), which includes a relevance filter for selecting hypotheses, an interface for passing proof tasks to external tools, and a mechanism for internally reconstructing external proofs.

A sledgehammer style approach seems particularly promising for DTPLs where it could make program development more lightweight and less time consuming. Unfortunately however, ATP integration for DTPLs is not straightforward. First, state-of-the-art ATP technology is designed for classical reasoning whereas DTPLs require constructive logic. Second, the logical kernels of DTPLs tend to be much more complex than those of traditional ITP systems, hence proof reconstruction establishes relatively less trust. Third, proof reconstruction for DTPLs has been highly inefficient in practice due to proof normalisation, whereas in theory it should be linear in the size of input proofs [15].

Due to these issues, ATP integrations for DTPLs certainly deserve to be studied in a pure and simple setting. This essentially amounts to building a simple trustworthy DTPL kernel around an ATP system as its most important proof engine. In this paper we focus in particular on the communication between ATP and ITP and the efficiency of proof reconstruction. For this purpose we implement the extended calculus of constructions with universes as a minimalistic DTPL, called Mella, in Haskell. Details of this implementation can be found in a technical report [2]; since they do not contain any significant research contribution, they are not included in this paper. The complete Mella tool can be obtained online[1]. Our main contributions are as follows:

First, we design and implement a simple proof scripting language for Mella inspired by Isabelle/Isar and Agda. Apart from commands for executing interactive proofs it calls external ATP systems within the Proof General interface [3].

Second, we provide interfaces for executing Mella proofs in the ATP system Waldmeister and for reconstructing Waldmeister proofs within Mella. Proof reconstruction amounts to building a Mella proof term and type checking it; proof normalisation is avoided where possible for efficiency.

Third, we test the performance of the ATP integration on more than 800 proof tasks from the TPTP[2] library [31]. In contrast to previous approaches, proof reconstruction is very effective and does not create a significant overhead to Waldmeister proof search. However, a small number of proof reconstructions currently fail due to dynamic scoping problems.

In many ways, Mella is still a prototype and our approach to proof reconstruction requires refinement. The DTPL implemented has neither recursion nor data types. It is just expressive enough to support proofs in many-sorted first-order constructive logic with equality. But for the main purpose of this paper—exploring effective ATP integrations for DTPLs—this is certainly no limitation.

Two particular features of Mella proof reconstruction are that proof search and proof normalisation are avoided wherever possible. Our micro-step reconstruction is in contrast to Isabelle's current macro-step approach based on the

[1] www.dcs.shef.ac.uk/~alasdair

[2] Thousands of Problems for Theorem Provers. www.cs.miami.edu/~tptp

internally verified ATP system Metis [19], and it seems more robust and efficient. In contrast to Agda or Coq, we only type check the internal proof terms corresponding to external proofs. These proof terms provide proof certificates that could be further normalised if needed. Whenever type checking succeeds, correctness of Mella's type theory guarantees that normalisation is possible.

2 Calculus of Constructions

This section reviews the basics of the calculus of constructions, which is Mella's underlying type theory. We assume familiarity with basic type systems [6,28,26]. The typing rules of this calculus are given in Figure 1; its details are explained in the remainder of this section,

$$\text{T-Axiom}\ \frac{}{\vdash s\ :_{\uparrow} s'}\ s : s' \in \mathcal{A}$$

$$\text{T-Named}\ \frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x}\ :_{\uparrow} T} \qquad \text{T-Unnamed}\ \frac{\Delta\,!\,n \equiv_{\beta} T}{\Delta \vdash n\ :_{\uparrow} T}$$

$$\Gamma\text{-Weakening}\ \frac{\Gamma; \Delta \vdash \mathtt{x}\ :_{\downarrow} T \qquad \Gamma; \Delta \vdash S\ :_{\uparrow} s}{\Gamma, \mathtt{y} : S; \Delta \vdash \mathtt{x}\ :_{\downarrow} T}\ s \in \mathcal{S} \text{ and } \mathtt{y} \text{ is fresh}$$

$$\Delta\text{-Weakening}\ \frac{\Delta \vdash n\ :_{\uparrow} T}{\Delta' +\!\!+ \Delta \vdash n\ :_{\uparrow} T}\ \Delta' \text{ is valid}$$

$$\text{T-Abs}\ \frac{\Gamma; \Delta \vdash S\ :_{\uparrow} s \qquad \Gamma; \Delta, S \vdash t\ :_{\downarrow} T}{\Gamma; \Delta \vdash \lambda t\ :_{\downarrow} \Pi\, S\,.\,T}\ s \in \mathcal{S}$$

$$\text{T-App}\ \frac{\Gamma; \Delta \vdash f\ :_{\uparrow} \Pi\, S\,.\,T \qquad \Gamma; \Delta \vdash x\ :_{\downarrow} S}{\Gamma; \Delta \vdash f\,x\ :_{\uparrow} \downarrow^{1} [0 \mapsto \uparrow^{1} x]T}$$

$$\text{T-Pi}\ \frac{\Gamma; \Delta \vdash S\ :_{\uparrow} s_1 \qquad \Gamma; \Delta, S \vdash T\ :_{\uparrow} s_2}{\Gamma; \Delta \vdash \Pi\, S\,.\,T\ :_{\uparrow} s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\text{T-Inf}\ \frac{\Gamma; \Delta \vdash T\ :_{\uparrow} s \qquad \Gamma; \Delta \vdash t\ :_{\uparrow} T' \qquad T \equiv_{\beta} T'}{\Gamma; \Delta \vdash t\ :_{\downarrow} T}\ s \in \mathcal{S}$$

$$\text{T-Ann}\ \frac{\Gamma; \Delta \vdash T\ :_{\uparrow} s \qquad \Gamma; \Delta \vdash t\ :_{\downarrow} T}{\Gamma; \Delta \vdash t :: T\ :_{\uparrow} T}\ s \in \mathcal{S}$$

Fig. 1. Typing rules for $CC\omega$

The set of terms of the calculus of constructions (CC) is inductively defined by the following grammar.

$$t ::= x \in V \mid \Pi x\!:\!t\,.\,t \mid \lambda x\!:\!t\,.\,t \mid t\; t$$

Here, V is a set of variables. $\Pi x\!:\!t\,.\,t$ is the dependent product type; it essentially amounts to universal quantification. $\lambda x : t\,.\,t$ is lambda abstraction and $t\; t$ is application. In CC, types themselves are terms. They are distinguished, and their mutual dependencies are expressed, by the type inference rules. Terms that are not types are called *non-type terms*, or briefly *terms* if the context allows. A type of a non-type term is called *proper*, whereas types of types are called *sorts*.

Judgements are expressions $\Gamma \vdash t : T$, where Γ is an environment that provides types for variables, t is a term and T a type. They can be proved by the type inference rules.

In CC, every proper type has sort $\star$, while $\star$ is defined to have sort $\Box$. The dependencies between terms and types for CC can be modelled by the set $\{(\star,\star),(\star,\Box),(\Box,\star),(\Box,\Box)\}$. The statement $(\star,\star)$, for instance, says that terms may depend on terms; the statement $(\Box,\star)$ says that terms may depend on types.

To make our calculus rich enough for a DTPL we extend it with universes. The *calculus of constructions with universes*, $CC\omega$, extends CC with an infinite set $\Box_0,\ldots,\Box_n,\ldots$ of sorts [8,23]. A *pure type system* (PTS) is given by a triple $(\mathcal{S},\mathcal{A},\mathcal{R})$, where $\mathcal{S}$ is a set of sorts and $\mathcal{A}$ a set of typing relations $s_1 : s_2$ with $s_1, s_2 \in \mathcal{S}$. The set $\mathcal{R}$ consists of triples (s_1,s_2,s_3), where $s_1,s_2,s_3 \in \mathcal{S}$. This set, in combination with the typing rule T-Pi in Figure 1 controls the dependencies of terms and types. The PTS for $CC\omega$ [8] is given by:

$$\begin{aligned}
\mathcal{S} &= \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\},\\
\mathcal{A} &= \{\star : \Box_0\} \cup \{\Box_i : \Box_{i+1} \mid i \in \mathbb{N}\},\\
\mathcal{R} &= \{\star \rightsquigarrow \star, \star \rightsquigarrow \Box_i, \Box_i \rightsquigarrow \star \mid i \in \mathbb{N}\} \cup \{(\Box_i,\Box_j,\Box_{\max(i,j)}) \mid i \in \mathbb{N}\},
\end{aligned}$$

The notation $s_1 \rightsquigarrow s_2$ is shorthand for (s_1,s_2,s_2). $\star \rightsquigarrow \star$ means that terms can depend on terms, $\star \rightsquigarrow \Box_i$ means that types can depend on terms (dependent types) and $\Box_i \rightsquigarrow \star$ means that terms can depend on types. The set of all triples $(\Box_i,\Box_j,\Box_{\max(i,j)})$ defines how types are allowed to depend on types. If a type $\Box_i$ depends on another type $\Box_j$, it must be at the same level in the hierarchy as the highest type $\Box_{\max(i,j)}$ it depends on.

The syntax of $CC\omega$ terms is defined by the following grammar, which extends and refines that for CC:

$$t ::= \; s \in \mathcal{S} \mid n \in \mathbb{N} \mid \mathtt{x} \in V \mid \lambda t \mid \Pi\, t\,.\,t \mid t\,t \mid t :: t\;.$$

We use de Bruijn indices to represent variables introduced via λ or Π binders, whereas top-level declarations are named [13]. Named variables are elements of V, the set of valid identifiers. Both named and unnamed variables can have free or bound occurrences. For example, in the term $\lambda\, 3$, the index 3 is free because

it is pointing outside the term. A term without free de Bruijn indices is called *locally closed*.

The dependent product type $\Pi\, A\,.\,B$ corresponds to the logical statement $\forall a \in A\,.\; B(a)$. To prove $\forall a \in A\,.\; B(a)$ constructively, one needs to show that for every possible $a \in A$ an inhabitant of B can be constructed. A function of type $\Pi\, A\,.\,B$ is therefore a proof of the statement $\forall a \in A.\ B(a)$. If B does not depend on A, then the dependent product type is $A \to B$. The additional syntax $t :: t$ is *type annotation*. It allows us to explicitly state that a given term has some type.

Because there are two kinds of variables—named and unnamed ones—judgements take the form $\Gamma; \Delta \vdash t : T$, where Γ and Δ are the typing contexts for named and unnamed variables. The syntax for these contexts is

$$\Gamma ::= \ \emptyset \mid \Gamma, \mathrm{x} : T, \qquad \Delta ::= \ \emptyset \mid \Delta, T.$$

Both contexts are lists, but since Γ names must be unique in Γ, it can be treated as a set. We use $\emptyset$ to represent empty contexts. We often omit empty contexts and write $\vdash t : T$ rather than $\emptyset; \emptyset \vdash t : T$. We write $\Gamma, \mathtt{x} : T$ to denote that context Γ is extended with the new binding $\mathtt{x} : T$, whereas for Δ, only a type is supplied. We write $\mathtt{x} : T \in \Gamma$ to assert that T is the type of $\mathtt{x}$ in Γ. We write $[k \mapsto t']t$ for the substitution of term t' for the index k in the term t.

Unlike variables in Γ, those in Δ are nameless and cannot be looked up by name. Instead we define two lookup operators ! and !! with and without index shifting to retrieve the types of variables from Δ.

$$\Delta, T \,!!\, n = \begin{cases} T & \text{if } n = 0, \\ \Delta \,!!\, (n-1) & \text{otherwise}, \end{cases} \qquad\qquad \Delta \,!\, n = \uparrow^{n+1} (\Delta \,!!\, n).$$

$\uparrow_c^d t$ is the d-place shift of a term t above cutoff c [28], where any index under the cutoff is left unshifted. We write $\uparrow^d t$ if the cutoff is zero. An unnamed context is well-formed only if all the terms within it are either proper types or sorts. Unnamed contexts can be concatenated using the $+\!\!+$ operator.

To implement this calculus, a *bidirectional type checker* is used [27,21]. This means that for any term t of type T, one can either infer the type, written $t :_\uparrow T$, or check that the term has the type, written $t :_\downarrow T$. Type inference requires t and returns T, whereas type checking requires both t and T. The two rules T-Inf and T-Ann (the rule for type annotations) provide a conversion between type checking and type inference. Bidirectional type checking ensures that the rules are directly implementable without need for further transformation.

3 An Extended Calculus

Mella requires additional features for equational and incremental interactive reasoning: identity types and metavariables. We call this extension CC_ω^+.

Firstly, we add identity types to $CC\omega$. The identity type $\mathrm{Id}_A(a, b)$ for any type A, where $a, b : A$, denotes that a and b represent identical proofs of proposition A [24]. This captures propositional equality within Mella and supports equational

reasoning. Our identity type corresponds to the implementation of propositional equality as an inductive family in Agda. Several new terms need to be added to the grammar of $CC\omega$:

$$t ::= \lambda x\,.\,t \mid \ \ldots\ \mid \mathrm{refl} \mid \mathrm{Id}_t(t,t) \mid \mathrm{elimJ}\ .$$

The *reflexivity term* refl works exactly like `refl` in Agda [25]. It allows the construction of identity types $\mathrm{Id}_A(a,b)$ where $a \equiv_\beta b$. The typing rules for the reflexivity term Eq-Refl and the identity term Eq-Id are as follows [4]:

$$\text{Eq-Refl}\ \frac{\Gamma;\Delta \vdash A\ :_\uparrow s \qquad \Gamma;\Delta \vdash a,b\ :_\downarrow A \qquad a \equiv_\beta b}{\Gamma;\Delta \vdash \mathrm{refl}\ :_\downarrow \mathrm{Id}_A(a,b)}\ s \in \mathcal{S}$$

$$\text{Eq-Id}\ \frac{\Gamma;\Delta \vdash A\ :_\uparrow s \qquad \Gamma;\Delta \vdash a,b\ :_\downarrow A}{\Gamma;\Delta \vdash \mathrm{Id}_A(a,b)\ :_\uparrow s}\ s \in \mathcal{S}$$

The J rule below eliminates identity types [14], which corresponds to the term elimJ. It can be used in combination with refl to define the standard functions of equational logic in Mella, namely, substitutivity, congruence, transitivity and symmetry. Because displaying the J rule with the locally nameless syntax discussed in Section 2 would render it almost unreadable, we present it in Mella syntax (see Section 5 for details):

```
theorem elimJ : "(A : *) (C : (x y : A) -> Id A x y -> *)
              -> (e : (x : A) -> C x x refl)
              -> (x y : A) (P : Id A x y) -> C x y P".
  "\A C e x y P -> e x".
  qed.
```

Like Isabelle's proof scripting language Isar, Mella uses two levels of syntax. The inner syntax, surrounded by quotation marks, is used for CC_ω^+ terms. The outer syntax is for proof scripting. For user interaction, the locally nameless term representation is extended to a more readable named representation. The inner syntax is essentially a simplification of Agda's syntax for terms without implicit arguments or mixfix operators.

Secondly, metavariables are used in Mella. Just as in Agda, these represent "holes" within terms that can incrementally be filled in—or refined—during proofs. Metavariables require one final language extension:

$$t ::= \lambda x\,.\,t \mid \ \ldots\ \mid ?\ .$$

As an example, consider checking that term $\lambda ?$ has type $\Pi\ \star\ .\ \Pi\, 0\,.\,1$.

$$\frac{\dfrac{}{\Gamma;\Delta \vdash \star\ :_\uparrow \Box_1}\ \text{T-Axiom} \qquad \Gamma;\Delta,\star \vdash\ ?\ :_\downarrow \Pi\, 0\,.\,1}{\Gamma;\Delta \vdash \lambda ?\ :_\downarrow \Pi\ \star\ .\ \Pi\, 0\,.\,1}\ \text{T-Abs}$$

When we try to check that $?\ :_\downarrow \Pi\, 0\,.\,1$, the type checker cannot proceed, so it stores a continuation which allows type checking to resume once a term for the metavariable has been supplied. This forms the basis for interactive theorem proving in Mella.

4 Automated Theorem Proving Technology

Having outlined the type-theoretic foundations of Mella, we now discuss the ATP technology which serves as its proof engine.

ATP systems have been designed and implemented for many decades, but mainly for classical first-order logic with equations. They provide fully automated proof search based on sophisticated term orderings, rewriting techniques and heuristics. They can often prove mathematical statements of moderate difficulty and deal with large hypothesis sets, which makes them ideally suited for discharging "trivial" first-order proof goals in ITP systems. A prime example of an ATP integration is Isabelle's Sledgehammer tool (cf. [10] for an overview), which calls a number of external ATP systems and SMT solvers. A relevance filter selects hypotheses for the proof, and the external proof output is internally reconstructed to increase trustworthiness. Proof reconstruction is based on the Metis tool [19], an Isabelle-verified automated theorem prover, which replays the external proof search with the hypotheses used by the external provers.

An integration of ATP systems into DTPLs is, however, much less straightforward, as discussed in the introduction. We therefore start with the simplest case—pure equational logic—for which classical and constructive reasoning coincide. We integrate the Waldmeister system [18], which is highly effective for this fragment and supports sorts[3].

Waldmeister accepts a set of equations as hypotheses and a single equation as a conclusion. It also requires a term ordering to use rewriting techniques for enhanced proof search. Technically, Waldmeister is based on the unfailing completion procedure [5], a variant of Knuth-Bendix completion [20] that attempts to construct a (ground) canonical term rewrite system from the equational hypotheses. This construction need not be finite, but it is guaranteed that a (rewrite) proof of a valid goal can be found in finite time. Apart from efficient proof search, Waldmeister offers two additional features that benefit an integration into Mella. First, it provides extremely detailed proof output, down to the level of positions and substitutions for rewrites in terms. In contrast to Sledgehammer's macro-step proof reconstruction that replays proof search, we can therefore check individual proof steps efficiently and without search. Second, Waldmeister extracts lemmas from proofs. This memoisation of subproofs further enhances proof reconstruction.

These features can be demonstrated by a simple example from group theory. Let $(G, \circ, ^{-1}, 1)$ be a group with carrier G, multiplication $\circ$, inversion $^{-1}$ and unit 1. It satisfies the axioms of associativity, right identity and right inverse

$$x \circ (y \circ z) = (x \circ y) \circ z, \qquad x \circ 1 = x, \qquad x \circ x^{-1} = 1.$$

Assume that we have implemented groups in Mella and want to prove that every right identity is also a left identity: $x^{-1} \circ x = x \circ x^{-1}$. We then need to pass the axioms and the proof goal to Waldmeister and let it search for a proof. Figure 2 shows the Waldmeister input file that corresponds to this proof task.

[3] We are using the last publicly available version of Waldmeister, released in 1999.

```
NAME group
MODE PROOF
SORTS
  ANY
SIGNATURE
  one: -> ANY
  inv: ANY -> ANY
  op: ANY ANY -> ANY
  a: -> ANY
ORDERING
LPO
  inv > op > one > a
VARIABLES
  x,y,z : ANY
EQUATIONS
  op(x,one) = x
  op(x,inv(x)) = one
  op(op(x,y),z) = op(x,op(y,z))
CONCLUSION
  op(a,inv(a)) = op(inv(a),a)
```

Fig. 2. Waldmeister group input file

The group signature is declared in prefix notation, using sort `ANY`, and functions `op: ANY ANY -> ANY`, `inv:ANY -> ANY` and `one: -> ANY` for multiplication, inverse, and unit. A constant `a` is also introduced to express the conclusion. Waldmeister's term ordering is declared in the `ORDERING` block: a lexicographic path ordering (lpo) is constructed from a precedence on the group signature and the constant `a`. The next block declares three variables `x`, `y` and `z` of type `ANY`. The `EQUATIONS` block lists the group axioms in Waldmeister syntax. Finally, the proof goal is declared in Waldmeister syntax for the constant `a`, since universal goals are Skolemised.

After Waldmeister is called, it returns the proof in Figure 3 within milliseconds. Here, the `--details` flag has been set to obtain precise information for each proof step. In the third step of the proof of `Lemma 1`,

```
f(x1,f(i(x1),i(i(x1)))) = f(x1,e)
```

for instance, the right identity axiom `f(x1,i(x1)) = e` has been used to rewrite from left to right the subterm at position `2` by matching or substituting `i(x1)` for `x1`. This level of detail allows efficient micro-step proof reconstruction; the lemmas generated support proof reconstruction by memoisation. Details of the communication between Mella and Waldmeister, in particular proof reconstruction, are covered in the following section.

```
Lemma 1: op(one,inv(inv(x1))) = x1

    op(one,inv(inv(x1)))
=     by Axiom 2 RL at 1 with {x1 <- x1}
    op(op(x1,inv(x1)),inv(inv(x1)))
=     by Axiom 3 LR at e with {x3 <- inv(inv(x1)), x2 <- inv(x1), x1 <- x1}
    op(x1,op(inv(x1),inv(inv(x1))))
=     by Axiom 2 LR at 2 with {x1 <- inv(x1)}
    op(x1,one)
=     by Axiom 1 LR at e with {x1 <- x1}
    x1

Lemma 2: ...

Lemma 3: ...

Lemma 4: ...

Theorem 1: op(a,inv(a)) = op(inv(a),a)

    op(a,inv(a))
=     by Axiom 2 LR at e with {x1 <- a}
    one
=     by Axiom 2 RL at e with {x1 <- inv(a)}
    op(inv(a),inv(inv(a)))
=     by Lemma 4 LR at 2 with {x1 <- a}
    op(inv(a),a)
```

Fig. 3. Waldmeister group output file

5 ATP Integration

Our general approach to ATP integration is depicted in Figure 4. Mella proof tasks are represented as judgements $\Gamma;\Delta \vdash ? : T$. They encode that from a set of hypotheses given by the contexts Γ and Δ a proof term t—represented by metavariable ?—of type T (the proof goal) is to be inferred. This is achieved by serialising Γ, Δ and T and passing them on to Waldmeister. In our group example, Γ and Δ contain the group axioms, whereas T contains the proof goal. More generally, the contexts can also contain lemmas that have been proved before. If Waldmeister fails to find a proof within a certain time limit, the user is notified. Otherwise, its proof output is translated into a proof term t in Mella, which is then type checked. Since Waldmeister produces intermediate lemmas, as we have seen, an additional context Γ' is added to Γ. Constructing a proof term from a Waldmeister proof and type checking it yields *proof reconstruction*. We now discuss the individual steps in more detail.

Users interact with Mella via the Proof General Emacs interface, which is standard for many ITP systems [3]. User level terms with explicit variables are parsed to an internal Haskell representation using de Bruijn indices corresponding to the type theory discussed in Section 2. A Mella file consists of a list of

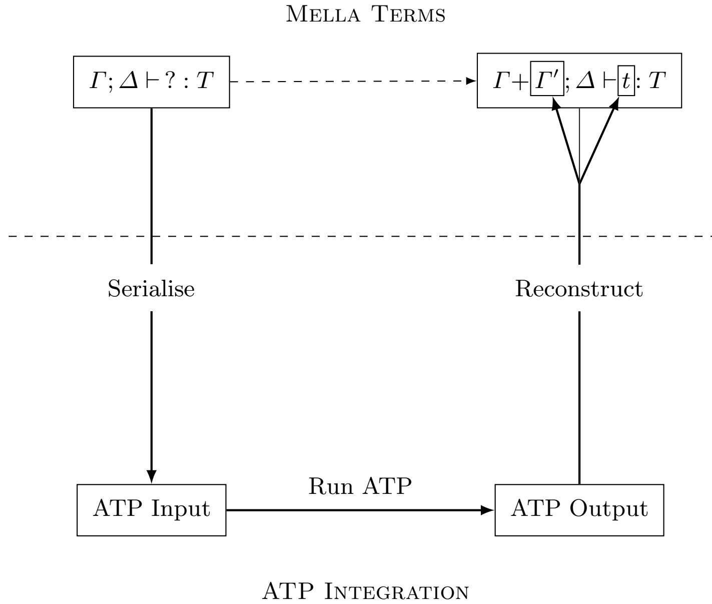


Fig. 4. Overview of Waldmeister Integration

commands delimited by periods, each of which can be processed and undone individually by Proof General. There are about 20 commands available to the user, which can be displayed using the `commands` command. The `help` command provides documentation for every command in the system. The command `fun` introduces a new top-level function or value. The following commands, for instance, introduce an identity function and a constant function in Mella.

```
fun id : "(A : *) -> A -> A"
  "\_ x -> x".

fun const : "(A B : *) -> A -> B -> A"
  "\_ _ x _ -> x".
```

To declare a theorem and start a proof, the `theorem` command is used. It takes the name of the theorem and its type T. To prove the theorem, the user must construct a proof term t such that $t :_\downarrow T$. Proofs are built up incrementally from commands and terms that may themselves contain metavariables.

As an example, assume we want to prove that

$$f(x, g(y, g(x, z))) = x \qquad \text{and} \qquad g(x, f(y, f(x, z))) = x$$

imply

$$f(x, g(y, x)) = x.$$

A “manual” Mella proof without using Waldmeister is as follows:

```
theorem example : "(A : *) (f g : A -> A -> A)
                -> (axiom1 : (x y z : A) -> Id A (f x (g y (g x z))) x)
                -> (axiom2 : (x y z : A) -> Id A (g x (f y (f x z))) x)
                -> (x y : A) -> Id A (f x (g y x)) x".
  intro A f g ax1 ax2 x y.
  = "f x (g y (g x (f x (f x x))))" by "ax2 x x x" at '2,2RL'.
  = "x" by "ax1 x y (f x (f x x))".
  refl.
  qed.

normalize example.
describe example.
```

Mella commands can be terms, which are surrounded by quotation marks, theorem definitions, function definitions, or command expressions. The command `intro` *args* generates a term of the form $\lambda\, args \rightarrow ?$. The command

```
 = "f x (g y (g x (f x (f x x))))" by "ax2 x x x" at '2,2RL'.
```

says that the left-hand term in the proof goal is equal to the term provided by applying the second axiom at position `2,2` to variable `x` from right to left, in a notation similar to Waldmeister. The second proof step is similar. The remaining step is reflexivity of equality. The commands `normalize example` and `describe example` normalise the proof and print out the proof term (which we do not show). We can also use the `agda` command to compile Mella files into Agda files. This is very useful for testing the correctness of our implementation.

Command expressions form a large part of Mella's syntax. Examples are `intro`, `=` and `qed` as displayed above. Commands consist of a command name, followed by zero or more arguments and a list of keywords. Each keyword can again be associated with a list of arguments:

$$\texttt{command}\ arg_1 ... arg_n\ \texttt{:keyword1}\ karg_1 ... karg_n\ \texttt{:keyword2}\ ...$$

Alternatively to the above manual proof we can use Waldmeister to prove our goal. The type is the same as above, so we omit it for brevity.

```
theorem example : "...".
  intro A f g ax1 ax2 x y.
  waldmeister :signature f g x y :axioms ax1 ax2 :kbo :timeout 2.
  qed.
```

The `waldmeister` command is now used to instantiate the metavariable opened by the `intro` command. Waldmeister is given the functions and values it may use in the proof via the `:signature` keyword, which maps to the `SIGNATURE` section of the Waldmeister input file. We give each function or variable in the signature an identifier sn. The number n is the de Bruijn index corresponding to that function in Δ. The axioms to be used when constructing the proof are listed after the `:axioms` keyword, and are used in the `EQUATIONS` section of the Waldmeister

input file. The `:kbo` option tells Waldmeister to use a Knuth-Bendix ordering as the syntactic ordering for terms (based on the precedence given by the order of expressions declared after `:signature`). Finally, the `:timeout` keyword lets one specify the amount of time Waldmeister will be given for proof search.

6 Proof Reconstruction

We now describe proof reconstruction. As already mentioned, Waldmeister splits proofs into lemmas. While this process is primarily intended to increase readability, it also enhances proof reconstruction by memoising subproofs. Were we to reconstruct the Waldmeister proof as a single term, lemmas would be repeated hundreds of times. This would dramatically slow down type checking to the point where it would become infeasible. This is also why we avoid normalisation of the final proof term, as it would reduce the proof to a single term, unmemoising all the lemmas found by Waldmeister and vastly reducing the efficiency of our system. This problem has affected previous integrations of Waldmeister into Agda [15]. Mella is designed in such a way as to avoid such issues, and as such our micro-step proof reconstruction is much more efficient.

The Waldmeister output for the example proof above is shown below. Waldmeister renames axioms in its output; so during reconstruction, they must be matched with the correct terms within Mella. The proof shows that the term `s5(s1,s4(s0,s1))` is equal to `s1`. It consists of two steps. First, Waldmeister applies Axiom 2 from right to left at position `2.2` in `s5(s1,s4(s0,s1))`, which results in the term shown on the next line. Secondly, Waldmeister uses Axiom 1 to reduce the term down to `s1`, proving the goal.

```
 Theorem 1: s5(s1,s4(s0,s1)) = s1

   s5(s1,s4(s0,s1))
=    by Axiom 2 RL at 2.2 with {x3 <- y, x2 <- z, x1 <- s1}
   s5(s1,s4(s0,s4(s1,s5(z,s5(s1,y)))))
=    by Axiom 1 LR at e with {x3 <- s5(z,s5(s1,y)), x2 <- s0, x1 <- s1}
   s1
```

Our proof reconstruction algorithm is relatively simple, and works as follows: We start by parsing the output provided by Waldmeister. Each Waldmeister lemma can be converted to an equation consisting of two terms, and a sequence of rewrite justifications which tell us how one of these terms can be rewritten to the other. A rewrite justification consists of a rule, which can either be an axiom provided to Waldmeister or a previously proven lemma, as well as information telling us where and how the rule is applied. In other words, to prove a goal $x = y$ each step of a proof applies a lemma or axiom to a subterm of x.

As mentioned above, once we have parsed the proof output, we must try to match each axiom in the output with its corresponding term in Mella. This stage is trickier than necessary, as Waldmeister numbers each axiom in a seemingly random order, so we create a list of Mella identifiers corresponding to that order,

which are associated with the Mella axioms in Δ. To do this we must inspect the structure of each axiom in the Waldmeister output, and search for the corresponding Mella term in Δ. This association implicitly maps between the typing context Δ and the Waldmeister proof output.

Lemmas are numbered sequentially by Waldmeister in a similar manner. Each time we reconstruct a lemma we remove it from the context of the goal we are proving and add it to Γ'. We store a list of identifiers referring to these new terms in Γ'. This does not yet provide all the information needed for proof reconstruction, since Waldmeister also explicitly displays information about particular matchings and term positions.

To proceed further with proof reconstruction we must therefore assign types to all the terms within each lemma. Since we encoded the index for each element of the signature in its name, we can easily map it back to the Mella typing context. This is also where the advantages of our bidirectional approach to type checking become clear. Since each Waldmeister expression consisists of a function applied to either variables or other function applications, the types of all terms provided to us by Waldmeister can be inferred by simply invoking the typing rule T-App from Section 2.

Next we can transform the list of typed rewrite justifications into a single Mella term. For each rewrite step, we must use congruence (to select the subterm) and symmetry (to choose the direction). If neither congruence nor symmetry is required for a step, they are omitted from the proof output, as is the case for the second step above. The above Waldmeister proof has two steps, hence we need to use transitivity to join both steps together, resulting in the final reconstructed Mella proof term for our example below. This proof term is somewhat unreadable; it has been indented to make the structure of the proof clearer.

```
trans A (f x (g y x)) (f x (g y (g x (f y (f x y))))) x
  (cong A A x (g x (f y (f x y))) (\rc-cong-var -> f x (g y rc-cong-var))
    (sym A (g x (f y (f x y))) x
      (ax2 x y y)))
  (ax1 x y (f y (f x y)))
```

7 Proof Experiments

We tested the Waldmeister integration on 850 proof goals from the TPTP library [31], among them 115 on Boolean algebras (BOO), 156 on lattices (LAT), 415 on groups (GRP), 106 on relation algebras (REL) and 58 on rings (RNG). The letters in brackets indicate the name given to these problem sets in TPTP. The library contains non-theorems and non-equational theorems that are beyond Waldmeister's scope. In fact, in our experiments, Waldmeister has not been able to find proofs for all goals for principal reasons, but may also have failed to find proofs of equational theorems due to timeout. Here, however, we are only interested in relative success rates for proof reconstruction, that is, the number or percentage of successful Waldmeister proofs that Mella was able to reconstruct,

Table 1. Proof Reconstruction Experiments

	Waldmeister				Proof Reconstruction		
	CPU Time	Timeout	Error	Unprovable	Fail	Success	%
BOO	300	9	51	1	10	44	81.5
	30	41	21	1	10	42	80.8
	10	62	0	1	10	42	80.8
	5	64	0	1	10	40	80
	1	66	0	1	10	38	79.2
LAT	300	91	14	0	11	40	78.4
	30	107	0	0	11	38	77.6
	10	110	0	0	11	35	76.1
	5	110	0	0	11	35	76.1
	1	113	0	0	9	34	79.1
GRP	300	39	4	0	213	159	42.7
	30	53	1	0	202	159	44.0
	10	57	0	0	199	159	44.4
	5	66	0	0	192	157	45
	1	91	0	0	185	139	42.9
REL	300	20	2	0	2	82	97.6
	30	34	0	0	2	70	97.2
	10	58	0	0	2	46	95.8
	5	61	0	0	0	45	100
	1	66	0	0	0	40	100
RNG	300	35	5	0	0	18	100
	30	41	0	0	0	17	100
	10	44	0	0	0	14	100
	5	44	0	0	0	14	100
	1	45	0	0	0	13	100

and in the running times of proof reconstruction relative to proof search. The outcome of these experiments are shown in Table 1.

The first column in the table shows the TPTP problem sets. The first four columns are related to Waldmeister. The first of them shows the Waldmeister CPU time limits for proof search—$1s$, $5s$, $10s$, $30s$ and $300s$. The second one gives the number of proofs searches that exceeded the time limit. The third one gives the number of proofs that aborted, for instance, due to out of memory errors. In the case of Boolean algebras, the fourth column shows that Waldmeister refuted one proof goal. The final three columns contain data on proof reconstruction. The first of them shows the number of proofs for which reconstruction failed; the second one the number of successfully reconstructed proofs. The row-wise sums of these columns give the numbers of successful Waldmeister proofs. The third row gives the percentage of successful proof reconstructions.

First, it turns out that the CPU time limit for Waldmeister has little impact on success rates. The number of successful Waldmeister proofs increases only slightly with proof search time; the success rates for reconstruction remain almost unaffected. This suggests that there is little correlation between proof search time

and the difficulty of reconstructing the resulting proof. Waldmeister could spend a long time traversing a search space only to find a very short and simple proof which is trivial to reconstruct.

Second, success rates are surprisingly different for different problem sets. For groups, proof reconstruction was particularly poor, succeeding only 45% of the time for proofs returned after a 5 second timeout. For rings and relation algebras, reconstruction succeeded almost always, with a 100% reconstruction success rate at 5 seconds. For lattices and Boolean algebras reconstruction was also overall successful; it is 80% for Boolean algebras and 76.1% for lattices (again with a 5 second timeout). Some explanations for this are given below.

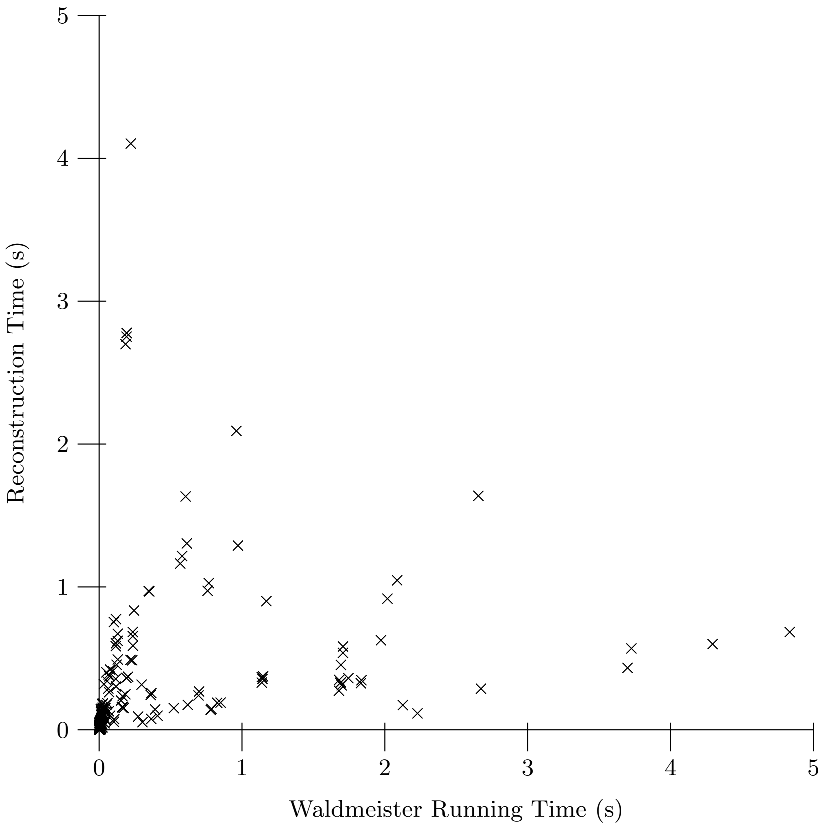


Fig. 5. Waldmeister running times versus proof reconstruction times

Next we have investigated the correlation between proof search and proof reconstruction times. A graph is plotted in Figure 5. Unfortunately, these times were very short for most of our proofs, which makes it very difficult to draw convincing conclusions. For some proofs, proof search took rather long whereas

reconstruction was fast. In other cases, proof search was fast, but the proof could not be reconstructed or type checked efficiently. We have inspected the proof for each goal that took longer than $2s$ to reconstruct. In each of these cases, either proof terms are extremely long, with more than 100 lemmas, or there are extremely large substitutions.

As an example, consider the following line from Figure 3:

```
 =  by Axiom 2 LR at 2 with {x1 <- i(x1)}
```

In the substitution `x1 <- i(x1)`, for instance, the term `i(x1)` can be enormous. In fact, our experiments contain substitutions of terms thousands of characters long, resulting in extremely large and unwieldy lemmas. This underscores the benefit of Waldmeister's lemma generation, which allows us to type check each one individually. As soon as proof terms become large, type checking slows down. These observations confirm what one would expect: proof reconstruction times depend on proof sizes rather than proof search times, whereas proof search time and proof size are often only weakly correlated. Proof length, however, is not a key factor for using ATP systems in DTP program development. Ultimately, our experiments suggest that the Waldmeister integration into Mella is feasible, and proof reconstruction yields little overhead to proof search.

There are several reasons why proof reconstruction may fail. Firstly, Waldmeister sometimes introduces fresh Skolem constants in proofs. These currently cannot be handled by the proof reconstruction code and cause it to fail. More precisely, such constants, which are dynamically generated by Waldmeister, can currently not be associated with an environment during proof reconstruction. Secondly, rules such as the right inverse axiom $x \circ x^{-1} = 1$ for groups, when applied from right to left to a (sub)term 1, can lead Waldmeister to introduce fresh variables in a proof. Mella would then have to introduce this value to the type signature of the lemma and supply it as a parameter. This currently assigns lemmas the wrong types in proofs and causes proof reconstruction to fail. For certain problem sets such as groups, "creative" proof steps of this kind seem particularly frequent, whereas in others (such as Boolean algebras, relation algebras or rings), they are present, but seem less significant.

As an example, consider the proof discussed in Section 5:

```
theorem proof : "(A : *) (f g : A -> A -> A)
             -> (axiom1 : (x y z : A) -> Id A (f x (g y (g x z))) x)
             -> (axiom2 : (x y z : A) -> Id A (g x (f y (f x z))) x)
             -> (x y : A) -> Id A (f x (g y x)) x".
  intro A f g ax1 ax2 x y.
  waldmeister :signature f g x y :axioms ax1 ax2 :kbo :timeout 2.
  qed.
```

Waldmeister uses the following lemma in its proof:

```
  Lemma 1: s10(x1,s14(x2,x1)) = x1

    s10(x1,s14(x2,x1))
```

```
 =    by Axiom 7 RL
    s10(x1,s14(x2,s14(x1,s10(z,s10(x1,y)))))
 =    by Axiom 8 LR
    x1
```

The second line of this proof introduces the new variables y and z. They are not mentioned in that lemma's type, hence the lemma cannot be easily reconstructed. We have implemented heuristics that guess instances of correct type for z and y (in this case $x1$ and $x2$) which are present in the context. In this particular lemma, these heuristics make proof reconstruction succeed. In many other case, we still obtain confusing error messages.

Most of these problems seem to stem from the fact that we reconstruct each lemma individually without taking the rest of the proof into consideration. If we took a more global approach to reconstructing Waldmeister proofs we could use more contextual information to solve such problems without having to resort to heuristics, or in the case of fresh Skolem constants, simply giving up. More precisely, we could preprocess the entire proof before reconstructing it in order to identify all variables and functions occurring in it. This information could then be used to update the proof context in Mella before proof reconstruction. This extended proof context would then prevent the proof reconstruction algorithm from failing. Therefore we believe that the reconstruction failures described above are not insurmountable, and with some improvements to the algorithm described in Section 5 we could reconstruct almost 100% of the proofs returned by Waldmeister, while still reconstructing each lemma individually, which is essential for efficiency reasons.

8 Related Work

The general question of proof automation for ITPs is covered in a wide variety of literature. Barendregt and Barendsen [7] identify three approaches, namely *accepting*, *skeptical*, and *autarkic*. The accepting approach uses ATP systems and SMT solvers as oracles, requiring no proof output. The skeptical approach requires that external tools provide evidence or certificates which allow ITP systems to internally reconstruct external proofs to increase trust. The autarkic approach solely relies on internal implementations of solvers and provers or alternatively by verifying external tools.

The accepting approach has, for many years, been pursued in the PVS ITP system, for instance by integrating the Yices SMT solver [30]. This approach has also been applied to Agda in [12]. However, this approach is often insufficient for constructive logic as proofs have computational content and may require execution, hence proof reconstruction.

The autarkic approach is the ideal, as an internally verified solver is guaranteed to produce correct output. The `omega`, `tauto` and `ring` tactics in Coq, and Isabelle's `blast` and `metis` tactics for instance, are autarkic. Other autarkic approaches allow the user to implement proof automation within the proof language itself using reflection [16]. The disadvantage of this approach however is

clear: there is a need to efficiently re-implement provers in the proof system, rather than using more powerful external provers.

The approach taken in this paper approach is skeptical. We believe this yields an adequate balance between efficiency and trust. Our approach is heavily inspired by Isabelle's Sledgehammer tool, which however is predominantly based on macro-step proof reconstruction. Additionally, ATP integration in Mizar—so far without proof reconstruction—is currently under development [29]. The skeptical approach has also been used in the context of dependent types, in a Waldmeister integration into Agda [15]. The relative inefficiency of this approach due to Agda proof normalisation was another main inspiration for Mella. Work on *proof irrelevance* in the most recent version of Agda, may however lead to a solution to this problem within Agda. More recently, using the skeptical approach, an SMT solver has been integrated into Coq [1].

9 Conclusion and Future Work

We have integrated the equational theorem prover Waldmeister into the prototypical dependently typed programming language Mella which is based on the extended calculus of constructions with universes. In contrast to previous approaches, where theorem provers were added a posteriori to existing ITP systems to complement existing internal tactics and proof strategies, we take the ATP system as a core proof engine for the programming language and build the language around it. As a user front end we have implemented a proof scripting language in the Proof General environment. This provides an interface between Mella and Waldmeister. Since Waldmeister provides highly detailed proof output we can perform micro-step proof reconstruction, translating the proof output into a Mella proof term and type checking that term.

Proof terms in Mella are not normalised. On the one hand, this makes proof reconstruction much more efficient. On the other hand this yields a proof certificate rather than a proper normalised proof. The strong normalisation property of the underlying type system, however, guarantees that all proofs that have been successfully checked can also be normalised. In the case of an equational proof this amounts to a refl term.

In sum, our findings suggest that integrating ATP systems into dependently typed languages can be very beneficial for program development in this setting, and that the approach taken with Mella may serve as a template for future approaches to integrate more expressive ATP systems in more sophisticated languages.

There are various interesting directions for future work.

First, already the minimalist formalism of Mella without recursion or data types requires proofs in full multi-sorted first-order constructive logic with equations. However, state-of-the-art ATP systems are essentially all based on classical first-order logic and often do not support sorts. Our current Waldmeister integration deals only with multi-sorted equational logic, a fragment where classical and constructive reasoning coincide. While using classical ATP systems for more

expressive fragments of first-order logic, such as Harrop formulae, is still possible, specific ATP systems for constructive or intuitionistic logic should be designed for applications in dependently typed programming.

Second, many state-of-the-art ATP systems adhere to a common input standard (TPTP), but many of them do not provide any detailed proof output or use a proprietary format. Detailed proof output is often perceived as detrimental to proof search efficiency. In the context of dependently typed programming, however, its absence is detrimental to proof reconstruction. As Sledgehammer shows, macro-step proof reconstruction, that is, replaying proof search with an internally verified theorem prover, has the disadvantage that many proofs provided by the external ATPs will not be accepted by the ITP system. Our proof experiments show that micro-step reconstruction of individual proofs steps is superior to this approach, but it requires detailed ATP output. Proof standardisation as in the TSTP project [32] is a valuable step in this direction. While sheer proof power was the main emphasis of ATP development in the past, applications in the context of ITP systems require this to be balanced with detailed proof output and support for types.

Third, in its current version, Mella still suffers from the fact that Waldmeister proofs, which introduce new constants or variables, cannot always be reconstructed. We could work around this by reconstructing proofs as they are, with additional constants and variables included, and proving that such reconstructed proofs are equivalent to the desired proofs. The simple heuristics currently used should further be refined to cover more proofs. Alternatively, when heuristics fail, the presence of lemmas in Waldmeister proof outputs allows local manual proof reconstruction. Often, reconstruction failures are caused by a very small number of lemmas. These could be replaced by metavariables so that the proof can be delegated to users. Thus, even when an ATP system cannot completely finish a proof, it might still produce a number of simpler proof goals for the user and at least simplify the global proof goal.

Fourth, Mella needs to be extended with features found in more sophisticated DTPLs and ITP tools. First we could extend CC^+_ω with data types, induction or Σ-types. Alternatively we could extend the proof scripting language by adding more automation, or by providing a more structured method of proof construction, similar to Isar. Some features like induction might only require proof management such as induction tactics, and would not affect the ATP integration, while others, such as the addition of Σ-types seem to require modifications to how ATP systems are integrated.

References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)
2. Armstrong, A., Struth, G., Foster, S.: Dependently typed programming based on automated theorem proving. Technical Report (2011), http://arxiv.org/abs/1112.3833

3. Aspinall, D.: Proof General: A Generic Tool for Proof Development. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 38–42. Springer, Heidelberg (2000)
4. Awodey, S., Warren, M.A.: Homotopy theoretic models of identity types. Math. Proc. Camb. Phil. Soc. 146, 45–55 (2009)
5. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Ait-Kaci, H., Nivat, M. (eds.) Resolution of Equations in Algebraic Structures, pp. 1–30. Academic Press (1989)
6. Barendregt, H.: Introduction to generalized type systems. Journal of functional programming 1(2), 125–154 (1991)
7. Barendregt, H., Barendsen, E.: Autarkic computations in formal proofs. Journal of Automated Reasoning 28(3), 321–336 (2002)
8. Bernardy, J.-P., Jansson, P., Paterson, R.: Parametricity and dependent types. SIGPLAN Not. 45, 345–356 (2010)
9. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer (2004)
10. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCos 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)
11. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
12. Bove, A., Dybjer, P., Sicard-Ramírez, A.: Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 104–118. Springer, Heidelberg (2012)
13. Charguéraud, A.: The locally nameless representation. Journal of Automated Reasoning (2011), doi:10.1007/s10817-011-9225-2
14. Dybjer, P.: Inductive families. Formal Aspects of Computing 6, 440–465 (1994)
15. Foster, S., Struth, G.: Integrating an Automated Theorem Prover into Agda. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 116–130. Springer, Heidelberg (2011)
16. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP 2011, pp. 163–175. ACM (2011)
17. PRL Group. Implementing Mathematics with the Nuprl Proof Development System. Computer Science Department, Cornell University (1995), http://www.cs.cornell.edu/info/projects/nuprl/book/doc.html
18. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister: High performance equational deduction. Journal of Automated Reasoning 18(2), 265–270 (1997)
19. Hurd, J.: System description: The Metis proof tactic. In: Benzmüller, C., Harrison, J., Schürmann, D. (eds.) ESHOL 2005, pp. 103–104 (2005), arXiv.org
20. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
21. Löh, A., McBride, C., Swierstra, W.: A Tutorial Implementation of a Dependently Typed Lambda Calculus. In: Altenkirch, T., Uustalu, T. (eds.) Dependently Typed Programming. Fundamenta Informaticae, vol. 102(2), pp. 177–207. IOS Press (2010)
22. McBride, C.: Epigram: Practical Programming with Dependent Types. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 130–170. Springer, Heidelberg (2005)

23. Miquel, A.: Le calcul des constructions implicite: syntaxe et sémantique. These de doctorat, Université Paris, 7 (2001)
24. Nordstrom, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press, USA (1990)
25. Norell, U.: Dependently Typed Programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
26. Pierce, B.C. (ed.): Advanced topics in types and programming languages. The MIT Press (2005)
27. Pierce, B.C., Turner, D.N.: Local Type Inference. In: Pugh, W. (ed.) TOPLAS 2000, pp. 1–44. ACM (2000)
28. Pierce, B.C.: Types and programming languages. The MIT Press (2002)
29. Rudnicki, P., Urban, J.: Escape to ATP in Mizar. PxTP 2011 (2011)
30. Rushby, J.M.: Tutorial: Automated formal methods with PVS, SAL and Yices. In: Hung, D.V., Pandya, P. (eds.) SEFM 2006, p. 262. IEEE Press (2006)
31. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
32. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: Zhang, W., Sorge, V. (eds.) FroCoS 2004, pp. 201–215. IOS Press (2004)

An Algebraic Calculus of Database Preferences

Bernhard Möller, Patrick Roocks, and Markus Endres

Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany
{moeller,roocks,endres}@informatik.uni-augsburg.de

Abstract. Preference algebra, an extension of the algebra of database relations, is a well-studied field in the area of personalized databases. It allows modelling user wishes by preference terms; they represent strict partial orders telling which database objects the user prefers over other ones. There are a number of constructors that allow combining simple preferences into quite complex, nested ones. A preference term is then used as a database query, and the results are the maximal objects according to the order it denotes. Depending on the size of the database, this can be computationally expensive. For optimisation, preference queries and the corresponding terms are transformed using a number of algebraic laws. So far, the correctness proofs for such laws have been performed by hand and in a point-wise fashion. We enrich the standard theory of relational databases to an algebraic framework that allows completely point-free reasoning about complex preferences. This black-box view is amenable to a treatment in first-order logic and hence to fully automated proofs using off-the-shelf verification tools. We exemplify the use of the calculus with some non-trivial laws, notably concerning so-called preference prefilters which perform a preselection to speed up the computation of the maximal objects proper.

Keywords: relational algebra, preferences, preference algebra, prefilter.

1 Introduction

In many database applications, the queries are based on multiple, and sometimes conflicting, goals. For example, a tourist may be interested in hotels in Nassau (Bahamas) which are *cheap*, have *reasonable ratings* (say, 3-star) and are *close to* the beach. Unfortunately, these goals are conflicting, as the hotels near the beach tend to be more expensive. Thus, there may be no single *optimal* answer: it is unlikely that there exists a single 3-star hotel that is cheapest among all 3-star hotels and closest to the beach. Still, users are looking for *satisfactory* answers. But what does "satisfactory" mean? For the same query, different users, guided by their personal preferences, may find different answers appealing. For example, a person may be willing to pay a little more to be closer to the beach; another may be contented with a cheaper hotel as long as it is convenient to reach the beach from it. Thus, in our example one would like to find a whole set of budget hotels, where those closer to the beach are slightly more expensive.

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 241–262, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Therefore it is important for a database system to present *all interesting* answers that may fulfil a user's need. Still, most current database search engines only deal with *hard constraints*: a tuple belongs to the search result if and only if it fulfils *all* given conditions.

As a remedy, queries with *soft constraints* are investigated, especially the so-called "skyline queries" [BKS01], which combine multiple, equally important, goals. An extension of this leads to the more comprehensive approach of *preference relations* which has been investigated in [Kie02, KH03, KEW11]. A preference allows users to establish a strict partial order that expresses which database objects are better for them than others. Based on this, a query selects according to the "best matches only (BMO)" model [Kie02] those objects that are not dominated by any others in the preference relation. To give the user more flexibility, a large set of predefined operators for constructing preference relations is provided.

Depending on the size of the database, the selection of the best matches according to a complex preference relation can be computationally expensive. To improve the process, preference queries and the corresponding terms are transformed using a number of algebraic laws for heuristically driven optimisation. So far, the correctness proofs for such laws have been performed by hand and in a point-wise fashion.

The contribution of the present paper is to enrich the standard theory of relational databases with an algebraic framework that allows completely point-free reasoning about (complex) preferences and their best matches. This "black-box view" is amenable to a treatment in first-order logic and hence to fully automated proofs using off-the-shelf verification tools. We exemplify the use of the calculus with some non-trivial laws, notably concerning so-called preference prefilters (introduced in [End11]), which perform a preselection to speed up the computation of the best matches proper, in particular, for queries involving expensive join operations. It turns out that the original laws hold under much weaker assumptions; moreover, several new ones are derived.

2 Types and Tuples

In this section we present the formal framework to model database objects as tuples. We introduce typed relations whose types represent attributes, i.e. the columns of a database relation. Conceptually and notationally, we largely base on [Kan90].

2.1 Typed Tuples

Definition 2.1. Let $\mathcal{A}$ be a set of *attribute names*. For $A \in \mathcal{A}$ the set D_A is called the *domain* of A, and $(D_A)_{A \in \mathcal{A}}$ is a family of domains. We define the following notions:

- A *type* T is a subset $T \subseteq \mathcal{A}$.
- An attribute $A \in \mathcal{A}$ is also used for the type $\{A\}$, omitting the set braces.
- A T*-tuple* is a mapping

$$t : T \rightarrow \bigcup_{A \in \mathcal{A}} D_A \;\text{ where }\; \forall\, A \in T : t(A) \in D_A.$$

- For a T-tuple t and a sub-type $T' \subseteq T$ we define the projection $\pi_{T'}(t)$ to T' as the restriction of the mapping t to T': $\pi_{T'}(t) : T' \rightarrow \bigcup_{A \in \mathcal{A}} D_A, A \mapsto t(A)$.
- The domain D_T for a type T is the set of all T-tuples, i.e., $D_T = \prod\limits_{A \in T} D_A$.
- The set $\mathcal{U} =_{df} \bigcup\limits_{T \subseteq \mathcal{A}} D_T$ is called the *universe*.
- For a tuple t, and a set of tuples M we introduce the following abbreviations:

$$t :: T \;\Leftrightarrow_{df}\; t \in D_T, \qquad M :: T \;\Leftrightarrow_{df}\; M \subseteq D_T.$$

Definition 2.2 (Join). The *join* of two types T_1, T_2 is the union of their attributes:

$$T_1 \bowtie T_2 \;=_{df}\; T_1 \cup T_2.$$

For sets of tuples $M_i :: T_i$ $(i = 1, 2)$, the join is defined as the set of all consistent combinations of M_i-tuples:

$$M_1 \bowtie M_2 \;=_{df}\; \{t :: T_1 \bowtie T_2 \mid \pi_{T_i}(t) \in M_i, i = 1, 2\}\;.$$

We illustrate this concept with the following example.

Example 2.3. Assume a database of cars with a unique ID and further attributes for model and horsepower. Hence the attribute names, i.e. types, are ID, model and hp. The tuples are written as explicit mappings. Assume the following sets:

$$M_1 \;=_{df}\; \{\{\text{ID} \mapsto 1,\ \text{model} \mapsto \text{'BMW 7'}\}, \{\text{ID} \mapsto 3,\ \text{model} \mapsto \text{'Mercedes CLS'}\}\},$$
$$M_2 \;=_{df}\; \{\{\text{ID} \mapsto 2,\ \text{hp} \mapsto 230\}, \{\text{ID} \mapsto 3,\ \text{hp} \mapsto 315\}\}.$$

The sets have the types $M_1 :: \text{ID} \bowtie \text{model}$ and $M_2 :: \text{ID} \bowtie \text{hp}$. Now we consider the join $M_1 \bowtie M_2 :: \text{ID} \bowtie \text{model} \bowtie \text{hp}$. We have $(\text{ID} \bowtie \text{model}) \cap (\text{ID} \bowtie \text{hp}) = \text{ID}$. The only tuple $t :: \text{ID} \bowtie \text{model} \bowtie \text{hp}$ which fulfills both $\pi_{T_1}(t) \in M_1$ and $\pi_{T_2}(t) \in M_2$ is the one with $t : \text{ID} \mapsto 3$. Hence the join is given by:

$$M_1 \bowtie M_2 = \{\{\text{ID} \mapsto 3,\ \text{model} \mapsto \text{'Mercedes CLS'},\ \text{hp} \mapsto 315\}\}\,.$$

Corollary 2.4. *The following laws hold:*

1. $\bowtie$ *is associative and commutative and distributes over* $\cup$.
2. $\bowtie$ *preserves the inclusion order, i.e.* $M \bowtie N \subseteq M' \bowtie N$ *for* $M \subseteq M'$.
3. *Assume* $M_i, N_i :: T_i$ $(i = 1, 2)$. *Then the following exchange law holds:*

$$(M_1 \cap N_1) \bowtie (M_2 \cap N_2) = (M_1 \bowtie M_2) \cap (N_1 \bowtie N_2).$$

Proof. (1) and (2) follow directly from definition. Using the definition of the join and the usual intersection of sets we show the exchange law as follows:

$$\begin{aligned} & x \in (M_1 \cap N_1) \bowtie (M_2 \cap N_2) \\ \Leftrightarrow\ & \pi_{T_1}(x) \in (M_1 \cap N_1) \ \wedge\ \pi_{T_2}(x) \in (M_2 \cap N_2) \\ \Leftrightarrow\ & \pi_{T_1}(x) \in M_1 \ \wedge\ \pi_{T_1}(x) \in N_1 \ \wedge\ \pi_{T_2}(x) \in M_2 \ \wedge\ \pi_{T_2}(x) \in N_2 \\ \Leftrightarrow\ & x \in M_1 \bowtie M_2 \ \wedge\ x \in N_1 \bowtie N_2 \\ \Leftrightarrow\ & x \in (M_1 \bowtie M_2) \cap (N_1 \bowtie N_2) \ . \end{aligned}$$

2.2 Typed Relations

Definition 2.5 (Typed homogeneous binary relations). For a type T we define the following abbreviations:

$$(t_1, t_2) :: T^2 \ \Leftrightarrow_{df}\ t_i \in D_T, \qquad R :: T^2 \ \Leftrightarrow_{df}\ R \subseteq D_T \times D_T.$$

We say that the *typed relation* R has type T. There are some special relations: The full relation $\top_T =_{df} D_T \times D_T$, the identity $1_T =_{df} \{(x, x) \mid x \in D_T\}$ and the empty relation $0_T =_{df} \emptyset$.

This concept of typed relations also appears in the relation-based logical, but not primarily algebraic, approach to database notions of [MO04]. We will generalise it in Section 3.2.

Definition 2.6 (Join of relations). Let $R_i :: T_i^2$ $(i = 1, 2)$. Then the *composition* $R_1 \bowtie R_2 :: (T_1 \bowtie T_2)^2$ is defined by

$$t\,(R_1 \bowtie R_2)\,u \ \Leftrightarrow_{df}\ \pi_{T_1}(t)\,R_1\,\pi_{T_1}(u) \ \wedge\ \pi_{T_2}(t)\,R_2\,\pi_{T_2}(u).$$

Corollary 2.7

1. *Assume $M_i, N_i :: T_i$ $(i = 1, 2)$. Then the following exchange law holds:*

$$(M_1 \bowtie M_2) \times (N_1 \bowtie N_2) = (M_1 \times N_1) \bowtie (M_2 \times N_2).$$

2. *For types T_1, T_2 and $X \in \{0, 1, \top\}$ we have $X_{T_1 \bowtie T_2} = X_{T_1} \bowtie X_{T_1}$.*

Proof

1. Straightforward from Definition 2.6.
2. Using part (1), $(D_{T_1} \bowtie D_{T_2}) \times (D_{T_1} \bowtie D_{T_2}) = (D_{T_1} \times D_{T_1}) \bowtie (D_{T_2} \times D_{T_2})$. By definition of the join for types we have that $T_1 \bowtie T_2 = T_1 \cup T_2$. From the definition of the join for sets we infer that $D_{T_1 \bowtie T_2} = D_{T_1} \bowtie D_{T_2}$. This shows the claim for $X = \top$. For $X = 1$ we show the equality component-wise using again the argument $D_{T_1 \bowtie T_2} = D_{T_1} \bowtie D_{T_2}$. For $X = \emptyset$ the claim is obvious.

Corollary 2.8

1. *For $M, N :: T$ we have $M \bowtie N = M \cap N$. In particular, we have $N \bowtie N = N$.*

2. *For* $R_1, R_2 :: T$ *we have* $R_1 \bowtie R_2 = R_1 \cap R_2$.
3. *For* $M_i :: T_i$ $(i = 1, 2)$ *with disjoint* T_i, *i.e., with* $T_1 \cap T_2 = \emptyset$, *the join* $M =_{df} M_1 \bowtie M_2$ *is isomorphic to the cartesian product of* M_1 *and* M_2.

Proof

1. By the definition of join and the typing assumptions we have

$$t \in M \bowtie N \Leftrightarrow t \in M \wedge t \in N \ .$$

2. Similarly we conclude for all $x, y :: T$:

$$x\,(R_1 \bowtie R_2)\,y \Leftrightarrow \pi_T(x)\,R_i\,\pi_T(y)\ (i = 1, 2) \Leftrightarrow x\,R_1\,y \wedge x\,R_2\,y$$

3. For $x \in M$, the two join conditions $\pi_{T_i}(x) \in M_i$ are independent. Hence all elements of M_1 can be joined with all elements of M_2. Thus, by definition,

$$t \in M \Leftrightarrow \pi_{T_1}(t) \in M_1 \wedge \pi_{T_2}(t) \in M_2 \Leftrightarrow (\pi_{T_1}(t), \pi_{T_2}(t)) \in M_1 \times M_2.$$

2.3 Inverse Image and Maximal Elements

Definition 2.9 (Inverse image). For a relation $R :: T^2$ the inverse image of a set $Y :: T$ under R is formally defined as

$$\langle R \rangle\,Y =_{df} \{x :: T \mid \exists y \in Y : x\,R\,y\} \ .$$

The notation stems from the fact that in modal logic the inverse-image operator is a (forward) diamond.

Lemma 2.10. *Assume* $R_i :: T_i^2$ *and* $Y_i :: T_i$ $(i = 1, 2)$ *with disjoint* T_1, T_2. *Then the following exchange law for the join and the inverse image holds:*

$$\langle R_1 \bowtie R_2 \rangle\,(Y_1 \bowtie Y_2) = \langle R_1 \rangle\,Y_1 \bowtie \langle R_2 \rangle\,Y_2 \ .$$

Proof. Using the definition of the inverse image and the composition of relations we infer:

$$\begin{aligned}
& x \in \langle R_1 \bowtie R_2 \rangle\,(Y_1 \bowtie Y_2) \\
\Leftrightarrow\ & \exists y \in (Y_1 \bowtie Y_2) : x\,(R_1 \bowtie R_2)\,y \\
\Leftrightarrow\ & \exists y \in (Y_1 \bowtie Y_2) : \pi_{T_1}(x)\,R_1\,\pi_{T_1}(y) \wedge \pi_{T_2}(x)\,R_2\,\pi_{T_2}(y) \\
\Leftrightarrow\ & \exists y_1 \in Y_1 : \exists y_2 \in Y_2 : \pi_{T_1}(x)\,R_1\,y_1 \wedge \pi_{T_2}(x)\,R_2\,y_2 \\
\Leftrightarrow\ & \pi_{T_1}(x) \in \langle R_1 \rangle\,Y_1 \wedge \pi_{T_2}(x) \in \langle R_2 \rangle\,Y_2 \\
\Leftrightarrow\ & x \in (\langle R_1 \rangle\,Y_1 \bowtie \langle R_2 \rangle\,Y_2) \ .
\end{aligned}$$

Note that splitting y into y_1 and y_2 in the third step is justified by disjointness of the types: because of $T_1 \cap T_2 = \emptyset$ the two join conditions $\pi_{T_i}(y) \in Y_i$ for $i = 1, 2$ are independent of each other, hence the substitution $y_i := \pi_{T_i}(y)$ is allowed.

Assume that R_1, R_2 are strict orders (irreflexive and transitive), which is the case in our application domain of preferences. Then, together with Corollary 2.8.3, this lemma means that, under the stated disjointness assumption, $R_1 \bowtie R_2$ behaves like the *product order* of R_1 and R_2 on the Cartesian product $D_{T_1} \times D_{T_2}$.

The inverse image of a set Y under a relation R, when viewed the other way around, consists of the objects that have an R-successor in Y, i.e., are R-related to some object in Y or, in the preference context, *dominated* by some object in Y. For this reason we can characterise the set of R-maximal objects within a set Y, as follows.

Definition 2.11 (Maximal elements). For a relation $R :: T^2$ and a set $Y :: T$ we define

$$R \rhd Y \ =_{df} \ Y - \langle R \rangle Y \ ,$$

where "−" is set difference.

These are the Y-objects that do not have an R-successor in Y, i.e., are not dominated by any object in Y. The mnemonic behind this notation is that in an order diagram for a preference relation R the maximal objects within Y are the peaks in Y; rotating the diagram clockwise by 90° puts the peaks to the right. Hence $R \rhd Y$ might also be read as "R-peaks in Y".

To develop the central properties of our algebra and the maximality operator it turns out useful to abstract from the concrete setting of binary relations over sets of tuples, which will be done in the next section.

3 An Algebraic Calculus

Since we have shown how to characterise the maximal elements concisely using a diamond operation, it seems advantageous to reuse the known algebraic theory around that. This also allows us to exhibit clearly which assumptions are really necessary; it turns out that most of the development is completely independent of the properties of irreflexivity and transitivity that were originally assumed for preference relations in [Kie02], and in fact also independent of the use of relations at all.

3.1 Semirings

Definition 3.1. An *idempotent semiring* consists of a set S of elements together with binary operations $+$ of *choice* and $\cdot$ of *composition*. Both are required to be associative, choice also to be commutative and idempotent. Moreover, composition has to distribute over choice in both arguments. Finally, there have to be units 0 for choice and 1 for composition.

Binary homogeneous relations over a set form an idempotent semiring with choice $\cup$ and composition ";", which have $\emptyset$ and the identity relation as their respective units.

Definition 3.2. Every idempotent semiring induces a *subsumption order* by $x \leq y \Leftrightarrow x + y = y$. A *test* is an element $x \leq 1$ that has a complement $\neg x$ relative to 1, i.e., which satisfies

$$x + \neg x = 1 \ , \qquad x \cdot \neg x = 0 \ .$$

It is well known (e.g. [MB85]) that the complement is unique when it exists and that the set of all tests forms a Boolean algebra with $+$ as join and $\cdot$ as meet. Tests are used to represent subsets or assertions in an algebraic way. In the semiring of binary relations over a set M the tests are subidentities, i.e., subsets of the identity relation, of the form $I_N =_{df} \{(x,x) \,|\, x \in N\}$ for some subset $N \subseteq M$ and hence in one-to-one correspondence with the subsets of M. Because of that we will, by a slight abuse of language, say that x *lies in* I_N when $(x,x) \in I_N$.

We will use small letters $a, b, c, ...$ at the beginning of the alphabet to denote arbitrary semiring elements and $p, q, ...$ to denote tests.

Based on complementation, the difference of two tests p, q can be defined as $p - q =_{df} p \cdot \neg q$. It satisfies, among other laws,

$$(p{+}q){-}r = (p{-}r){+}(q{-}r) \ , \quad (p{-}q){-}r = p{-}(q{+}r) \ , \quad p{-}(q{+}r) = (p{-}q){\cdot}(p{-}r) \ .$$

For the interaction between the complement and the subsumption ordering we can use the *shunting rule*

$$p \cdot q \leq r \Leftrightarrow p \leq \neg q + r \ .$$

A special case of applying this rule twice with $p = 1$ is the *contraposition* rule

$$q \leq r \Leftrightarrow \neg r \leq \neg q \ .$$

Tests can be used to express domain or range restrictions. For instance, when a is a relation and p, q are tests, $p \cdot a$ and $a \cdot q$ are the subrelations of a all of whose initial points lie in p and end points in q, respectively. Hence, all initial points of a lie in p if and only if $a \leq p \cdot a$.

With these properties we can give an algebraic characterisation of the test $\langle a \rangle \, q$ that represents the inverse image under a of the set represented by q or, equivalently, the set of initial points of $a \cdot q$.

Definition 3.3. Following [DMS06], the *(forward) diamond* is axiomatised by the universal property

$$\langle a \rangle \, q \leq p \Leftrightarrow a \cdot q \leq p \cdot a \cdot q \Leftrightarrow a \cdot q \leq p \cdot a \ .$$

Following the terminology of [DMS06], it would be more accurately termed a *pre-diamond*, since we do not require the axiom $\langle a \cdot b \rangle \, q = \langle a \rangle \, \langle b \rangle \, q$, which is not needed for our application. In the relational setting of [BW93], test and diamond are called monotoype and monotype factor, respectively.

The diamond enjoys the following useful algebraic properties:

$$\langle a \rangle \, 0 \ = \ 0 \ , \quad \langle a + b \rangle \, p \ = \ \langle a \rangle \, p + \langle b \rangle \, p \ , \quad \langle a \rangle \, (p + q) \ = \ \langle a \rangle \, p + \langle a \rangle \, q \ .$$

The latter two imply that diamond is isotone (i.e., monotonically increasing) in both arguments:

$$a \leq b \Rightarrow \langle a \rangle\, p \leq \langle b \rangle\, p \;, \quad p \leq q \Rightarrow \langle a \rangle\, p \leq \langle a \rangle\, q \;.$$

A special role is played by the test

$$\ulcorner a =_{df} \langle a \rangle\, 1 \;.$$

It represents the set of all objects that have an a-successor at all and therefore is called the *domain* of a. From the isotony of diamond we conclude, for test p,

$$\langle a \rangle\, p \leq \ulcorner a \;.$$

3.2 Representing Types

There are a number of ways to represent types algebraically, among then heterogeneous relation algebras [SHW97], relational allegories [BD97] or typed Kleene algebra [Koz98]. All these involve some amount of machinery and notation, which we want to avoid here.

More simply, we now interpret the largest test 1 as representing the universe $\mathcal{U}$ and use other tests to stand for subsets of it, e.g., for the domains associated with types. With every type $T \subseteq \mathcal{A}$, we associate a test 1_T representing its domain D_T. An assertion $p :: T$ means that p is a test, representing a set of tuples, with $p \leq 1_T$. Arbitrary semiring elements $a, b, c, ...$ will stand for preference relations. A type assertion $a :: T^2$ is short for $a \leq 1_T \cdot a \cdot 1_T$. By $1_T \leq 1$ this can be strengthened to an equality. Hence, since tests are idempotent under composition, $a :: T^2$ implies $1_T \cdot a = a = a \cdot 1_T$.

This latter property entails that the diamond respects types, i.e., for $a :: T^2$ and $q :: T$ we calculate

$$\langle a \rangle\, q :: T \Leftrightarrow \langle a \rangle\, q \leq 1_T \Leftrightarrow a \cdot q \leq 1_T \cdot a \cdot q \Leftrightarrow a \cdot q \leq a \cdot q \Leftrightarrow \mathsf{TRUE} \;.$$

To express that x is either an element which represents a relation *or* a test, we introduce the following notation:

$$x :: T^{(2)} \Leftrightarrow x :: T \vee x :: T^2 \;.$$

We will also need the infimum for elements $a_1, a_2 :: T^2$, which is axiomatised as follows:

$$\forall x :: T^2 : \quad x \leq a_1 \sqcap a_2 \Leftrightarrow_{df} x \leq a_1 \;\wedge\; x \leq a_2 \;.$$

In the semiring of binary relations this coincides with the intersection of two relations. For tests p, q we have, in every semiring, $p \sqcap q = p \cdot q$.

Finally, we assume for every type T a greatest element $\top_T$ in $\{x \,|\, x :: T^{(2)}\}$, i.e. we have $\forall\, x :: T^{(2)} : x \leq \top_T$.

3.3 Join Algebras

We now deal with the central notion of join. For this, we assume the typing mechanism of the previous section.

Definition 3.4 (Join algebra). A *join algebra* is an idempotent semiring with an additional binary operator $\bowtie$ satisfying the following requirements.

1. Join is associative, commutative and idempotent and distributes over choice $+$ in both arguments. Hence $\bowtie$ is isotone in both arguments.
2. If $a_i :: T_i^{(2)}$ $(i = 1, 2)$ then $a_1 \bowtie a_2 :: (T_1 \bowtie T_2)^{(2)}$.
3. For types T_i $(i = 1, 2)$ we have

$$1_{T_1 \bowtie T_2} = 1_{T_1} \bowtie 1_{T_2} \quad \text{and} \quad \top_{T_1 \bowtie T_2} = \top_{T_1} \bowtie \top_{T_2} \ .$$

4. Join and composition satisfy, for $a_i, b_i :: T_i^{(2)}$ $(i = 1, 2)$ with disjoint T_i, the exchange law

$$(a_1 \bowtie a_2) \cdot (b_1 \bowtie b_2) = (a_1 \cdot b_1) \bowtie (a_2 \cdot b_2).$$

5. The diamond operator respects joins of elements with disjoint types: for $a :: T_1^2, p :: T_1$ and $b :: T_2^2, q :: T_2$ with $T_1 \cap T_2 = \emptyset$ we have the exchange law

$$\langle a \bowtie b \rangle \, (p \bowtie q) \;=\; \langle a \rangle \, p \bowtie \langle b \rangle \, q \ .$$

Our typed relations from Section 2.2 form a join algebra.

3.4 Representation of Preferences

Preferences introduced in [Kie02] are strict partial orders, i.e. a special kind of binary homogeneous relations. These relations are defined on domains of types, and the objects compared are "database tuples" contained in a "database relation", i.e., a set of tuples.

To avoid confusion between the two uses of the word "relation" we call tuples *database elements* here and the database relation the *basic set* of objects. This means that we consider a "static" snapshot of the database at the time of the respective preference-based query and assume that no data is deleted or inserted into the database while the query being evaluated.

Abstractly, preferences can now be modelled as typed elements $a :: T^2$ for some type T. If one wants to express transitivity or irreflexivity of a, this can be done by requiring $a \cdot a \leq a$ or $a \sqcap 1_T = 0$, respectively. However, as we will see, for the most part these assumptions are inessential for the laws we will derive.

4 Maximal Element Algebra

Now we are ready for the algebraic treatment of our central notion.

4.1 Basic Definitions and Results

Definition 4.1. The *best* or *maximal* objects w.r.t. element $a :: T^2$ and test $p :: T$ are represented by the test

$$a \rhd p =_{df} p - \langle a \rangle\, p\ .$$

In particular, the test $a \rhd 1_T$ represents the a-best objects overall.

This definition is also given, in different notation, in [DMS06]. An analogous formulation, however, with tests encoded as vectors, i.e., right-universal relations, can be found in [SS93].

To give a first impression of the algebra at work, we show a number of useful basic properties of the $\rhd$ operator. Proofs of the following two lemmas can be found in Appendix B.1 and B.2.

Lemma 4.2. *Assume $a, b :: T^2$, $p :: T$. Then the following holds:*

1. $a \rhd 1_T = \neg\ulcorner a$.
2. $\ulcorner b \leq \ulcorner a \Leftrightarrow a \rhd 1_T \leq b \rhd 1_T$.
3. $a \rhd p \leq p$.
4. $a \rhd 1_T \leq p \Leftrightarrow a \rhd 1_T \leq a \rhd p$.
5. $a \rhd 1_T \leq a \rhd (a \rhd 1_T)$.
6. $a \rhd (a \rhd p) = a \rhd p$.
7. $(a + b) \rhd p = (a \rhd p) \cdot (b \rhd p)$.
8. $b \leq a \Rightarrow a \rhd p \leq b \rhd p$.
9. $1_T \leq a \Rightarrow a \rhd p = 0_T$.

Lemma 4.3. *Let $p, q :: T$ be a disjoint decomposition of 1_T, i.e. $p + q = 1_T$, $p \cdot q = 0_T$. Then we have $\neg p = q$.*

4.2 Basic Applications

Now we want to demonstrate how the maximality operator $\rhd$ works.

Example 4.4. Let $a :: T^2$ be a preference relation and suppose $p_1, p_2 :: T$ are tests that form a disjoint decomposition of 1_T. Assume that all elements in p_2 are better than all elements in p_1, i.e.,

$$\langle a \rangle\, p_2 = p_1, \quad \langle a \rangle\, p_1 = 0_T.$$

We show that p_2 represents the maximal elements, i.e. $p_2 = a \rhd 1_T$:

$$\begin{array}{ll} & a \rhd 1_T \\ = & \quad \{\!\![\ \text{definition}\ ]\!\!\} \\ & \neg \langle a \rangle\, 1_T \\ = & \quad \{\!\![\ p_1 + p_2 = 1_T\ ]\!\!\} \\ & \neg(\langle a \rangle\, (p_1 + p_2)) \end{array}$$

$$
\begin{aligned}
= &\quad \{\!\![\text{ distributivity of diamond }]\!\!\} \\
&\neg(\langle a \rangle\, p_1 + \langle a \rangle\, p_2) \\
= &\quad \{\!\![\text{ assumptions of } a \text{ }]\!\!\} \\
&\neg p_1 \\
= &\quad \{\!\![\text{ Lemma 4.3 }]\!\!\} \\
&p_2
\end{aligned}
$$

By this tiny example one can see how the maximality operator works in general, because one can always decompose 1_T into tests representing the non-maximal (p_1) and the maximal (p_2) elements, where p_1 and p_2 are disjoint.

4.3 Prefilters

In practical applications, e.g., in databases, the tests, in particular the test 1 representing all objects in the database, can be quite large. Hence it may be very expensive to compute $a \triangleright 1$ for a given a. However, it can be less expensive to compute $b \triangleright 1$ for another element b; ideally, that set is much smaller and the a-best objects overall coincide with the a-best objects within $b \triangleright 1$. This motivates the following definition.

Definition 4.5. Assume $a, b :: T^2$. We call b a *prefilter* for a, written as $b \,\mathsf{pref}\, a$, if and only if

$$a \triangleright 1_T = a \triangleright (b \triangleright 1_T) \ .$$

Note that no connection between a and b is assumed. By Lemma 4.2.6 we have $a \,\mathsf{pref}\, a$ for all a. A concrete example of a prefilter will be given in Section 5.1.

We can give another, calculationally useful, characterisation of prefilters. The proof of the following theorem can be found in appendix B.3.

Theorem 4.6. *$b \,\mathsf{pref}\, a \Leftrightarrow \ulcorner b \le \ulcorner a \wedge \ulcorner a \le \ulcorner b + \langle a \rangle \neg \ulcorner b$.*

So far, we have not required any special properties of the elements a that represent, e.g., preference relations. Instead of transitivity or irreflexivity we need an assumption that such elements admit "enough" maximal objects. This is expressed by requiring every non-maximal object to be dominated by some maximal one. In a setting with finitely many objects, such as a database, and a preference relation on them this property is always satisfied and hence is no undue restriction for our purposes. We forego a discussion of this assumption for infinite sets of objects, since there it is related to fundamental issues such as Zorn's Lemma and Hausdorff's maximality principle, hence to the axiom of choice.

Definition 4.7. We call an element $a :: T^2$ *normal* if it satisfies $\neg(a \triangleright 1_T) \le \langle a \rangle\, (a \triangleright 1_T)$.

This is a compact algebraic formulation of the above domination requirement. By Lemma 4.2.1 it is equivalent to $\ulcorner a \le \langle a \rangle \neg \ulcorner a$.

First we show that any relation on a subset of the domain of a normal relation provides a prefilter.

Theorem 4.8. *Assume $a, b :: T^2$.*

1. *Let a be normal. Then $\ulcorner b \leq \ulcorner a \Rightarrow b \,\mathsf{pref}\, a$.*
2. *Let $a + b$ be normal. Then $a \,\mathsf{pref}\, (a + b)$.*

Proof

1. The assumption about b is the first conjunct of the right hand side in Theorem 4.6. For the second conjunct we calculate

$$
\begin{array}{ll}
& \mathsf{TRUE} \\
\Leftrightarrow & \quad \{\!\![\ a \text{ normal }]\!\!\} \\
& \ulcorner a \leq \langle a \rangle \neg \ulcorner a \\
\Rightarrow & \quad \{\!\![\ \ulcorner b \leq \ulcorner a, \text{ contraposition and isotony of diamond }]\!\!\} \\
& \ulcorner a \leq \langle a \rangle \neg \ulcorner b \\
\Rightarrow & \quad \{\!\![\ x \leq x + y \text{ and transitivity of } \leq\ ]\!\!\} \\
& \ulcorner a \leq \ulcorner b + \langle a \rangle \neg \ulcorner b\ .
\end{array}
$$

2. Since $a \leq a + b$, isotony of diamond and hence of domain imply $\ulcorner a \leq \ulcorner(a + b)$ and the claim follows from Part 1.

Next we show that under certain conditions prefilters can be nested.

Theorem 4.9. *Assume $a, b, c :: T^2$, where $b \rhd 1_T \leq c \rhd 1_T$ and $b \,\mathsf{pref}\, a$ with normal a. Then also $c \,\mathsf{pref}\, a$.*

Proof. First, by Theorem 4.6 we have $\ulcorner b \leq \ulcorner a \wedge \ulcorner a \leq \ulcorner b + \langle a \rangle \neg \ulcorner b$. Second, by Lemma 4.2.1 and contraposition the assumption $b \rhd 1_T \leq c \rhd 1_T$ is equivalent to $\ulcorner c \leq \ulcorner b$. Hence by transitivity of $\leq$ we infer $\ulcorner c \leq \ulcorner a$. Now normality of a and Theorem 4.8.1 show the claim.

5 Complex Preferences

We have seen how some laws of single preference relations can be proved in point-free style in our algebra.

Now we want to *compose* preferences into *complex preferences*. To this end we will introduce some special operators. The standard semiring operations like multiplication, addition and meet also lead to some kind of complex preferences, but they are rarely used in the typical application domain of preference algebra [Kie02, KEW11]. Instead the so-called *Prioritisation* and *Pareto composition* are the most important constructors for complex preferences.

5.1 Complex Preferences as Typed Relations

To motivate our algebraic treatment we first repeat the definitions of these preference combinators in the concrete setting of typed relations [Kie02].

For basic sets M, N and preference relations $R \subseteq M^2, S \subseteq N^2$ the prioritisation $R \,\&\, S$ is defined as:

$$(x_1, x_2)\,(R \,\&\, S)\,(y_1, y_2) \;\Leftrightarrow_{df}\; x_1\,R\,y_1 \;\vee\; (x_1 = y_1 \;\wedge\; x_2\,S\,y_2)$$

where $x_i \in M, y_i \in N$. The Pareto preference is defined as:

$$\begin{aligned}(x_1, x_2)\,(R \otimes S)\,(y_1, y_2) \;\Leftrightarrow_{df}\;\; & x_1\,R\,y_1 \wedge (x_2\,S\,y_2 \vee x_2 = y_2) \;\vee \\ & x_2\,S\,y_2 \wedge (x_1\,R\,y_1 \vee x_1 = y_1)\end{aligned}$$

In order theory the prioritisation is well-known as *lexicographical order*.

We now want to get rid of the point-wise notation in favour of operators on relations. The technique is mostly standard; we exemplify it for the prioritisation. We calculate, assuming first $M :: A, N :: B$ with distinct attribute names A, B,

$$\begin{aligned}& (x_1, x_2)\,(R \,\&\, S)\,(y_1, y_2) \\ \Leftrightarrow \quad & \{\![\text{ definition }]\!\} \\ & x_1\,R\,y_1 \;\vee\; (x_1 = y_1 \;\wedge\; x_2\,S\,y_2) \\ \Leftrightarrow \quad & \{\![\text{ logic }]\!\} \\ & (x_1\,R\,y_1 \;\wedge\; \mathsf{true}) \;\vee\; (x_1 = y_1 \;\wedge\; x_2\,S\,y_2) \\ \Leftrightarrow \quad & \{\![\text{ definitions of } \top_B \text{ and } 1_A]\!\} \\ & (x_1\,R\,y_1 \;\wedge\; x_2 \top_B y_2) \;\vee\; (x_1\,1_A\,y_1 \;\wedge\; x_2\,S\,y_2) \\ \Leftrightarrow \quad & \{\![\text{ definition of cartesian product of relations }]\!\} \\ & (x_1, x_2)\,(R \times \top_B)\,(y_1, y_2) \;\vee\; (x_1, x_2)\,(1_A \times S)\,(y_1, y_2) \\ \Leftrightarrow \quad & \{\![\text{ definition of relational union }]\!\} \\ & (x_1, x_2)\,((R \times \top_B) \cup (1_A \times S))\,(y_1, y_2)\;.\end{aligned}$$

A similar calculation can be done for the Pareto composition. Now we can write the point-free equations

$$\begin{aligned} R \,\&\, S \;&=\; (R \times \top_B) \cup (1_A \times S)\;, \\ R \otimes S \;&=\; (R \times (S \cup 1_B)) \cup ((R \cup 1_A) \times S)\;. \end{aligned}$$

This is close to an abstract algebraic formulation. However, since we want to cover also the case of non-disjoint, overlapping tuples, we will replace the Cartesian product $\times$ by the join $\bowtie$. From now on a preference x has type T_x, i.e. $a :: T_a^2, b :: T_b^2, \ldots.$

Definition 5.1. For the sake of readability we define for $x :: T^2$:

$$0_x =_{df} 0_T, \quad 1_x =_{df} 1_T, \quad \top_x =_{df} \top_T$$

Definition 5.2 (Prioritisation/Pareto composition of preferences). Assume a join algebra. For $a :: T_a^2, b :: T_b^2$ the *Prioritisation* $a \,\&\, b :: T_a \bowtie T_b$ is defined by

$$a \,\&\, b =_{df} a \bowtie \top_b + 1_a \bowtie b.$$

The *Pareto compositions* $a \mathrel{\lessdot\!\!\otimes} b, a \mathrel{\otimes\!\!\gtrdot} b, a \otimes b :: T_a \bowtie T_b$ are defined by

$$\begin{aligned} a \mathrel{\lessdot\!\!\otimes} b &=_{df} a \bowtie (b + 1_b), \\ a \mathrel{\otimes\!\!\gtrdot} b &=_{df} (a + 1_a) \bowtie b, \\ a \otimes b &=_{df} a \mathrel{\lessdot\!\!\otimes} b + a \mathrel{\otimes\!\!\gtrdot} b. \end{aligned}$$

$a \mathrel{\lessdot\!\!\otimes} b$ and $a \mathrel{\otimes\!\!\gtrdot} b$ are called *left* and *right Semi-Pareto compositions*, while $a \otimes b$ is the standard *Pareto composition*.

Remark 5.3. Under certain circumstances the term $\langle \top_T \rangle\, q$ occurring, for example, in $\langle a \,\&\, b \rangle\, (p \bowtie q)$ can be simplified. Call an idempotent semiring *weakly Tarskian* if for all types T and tests $q :: T$ we have

$$\langle \top_T \rangle\, q = \begin{cases} 1_T & \text{if } q \neq 0_T, \\ 0_T & \text{if } q = 0_T. \end{cases}$$

For instance, the semiring of binary relations is weakly Tarskian. This implies that in a term like $\langle a \bowtie \top_b \rangle\, (q_1 \bowtie q_2)$ with $a :: T_a^2$ the test q_2 is irrelevant as long as $q_2 \neq 0_b$. This is exactly what we want, because $q_1 \bowtie 0_b$ $(= 0_{a \bowtie b})$ is a zero element and must not have successors in any relation.

A semiring with $\top$ is called *Tarskian* when $a \neq 0 \Rightarrow \top \cdot a \cdot \top = \top$. This property was first stated for the semiring of binary relations (see, e.g., [SS93]). By the standard theory of diamond and domain [DMS06], a Tarskian semiring is also weakly Tarskian, but generally not vice versa.

In our hotel example from the introduction, the user would typically express her preference as the Pareto composition of price and distance to the beach.

The definition of the Pareto compositions immediately yields an important optimisation tool.

Corollary 5.4. *The preferences $a \mathrel{\lessdot\!\!\otimes} b$ and $a \mathrel{\otimes\!\!\gtrdot} b$ are prefilters for $a \otimes b$. Likewise, $a \bowtie \top_B$ is a prefilter for $a \,\&\, b$.*

Proof. By definition, $a \mathrel{\lessdot\!\!\otimes} b, a \mathrel{\otimes\!\!\gtrdot} b \leq a \otimes b$ and $a \bowtie \top_B \leq a \,\&\, b$; hence Theorem 4.8.1 applies.

Hence, in our hotel example from the introduction, we may prefilter by price or by distance to the beach to speed up the overall filtering. Further applications of this principle are discussed in detail in [End11].

5.2 Maximality for Complex Preferences

We first state the behaviour of the maximality operator for joins of preference elements.

Lemma 5.5. *For $a :: T_a^2, p :: T_a$ and $b :: T_b^2, q :: T_b$ with $T_a \cap T_b = \emptyset$ we have*

$$(a \bowtie b) \rhd (p \bowtie q) \;=\; (a \rhd p) \bowtie q + p \bowtie (b \rhd q)\ .$$

Proof. We observe that, under the disjointness assumption, by Corollary 2.8.3 and a standard law for Cartesian products, for $r :: T_a, s :: T_b$, we have

$$(p \bowtie q) - (r \bowtie s) \;=\; (p - r) \bowtie q \;+\; p \bowtie (q - s)\ .$$

Hence, by the definitions and Lemma 2.10,

$$\begin{array}{rl} & (a \bowtie b) \rhd (p \bowtie q) \\ = & (p \bowtie q) - \langle a \bowtie b)\rangle\, (p \bowtie q) \\ = & (p \bowtie q) - (\langle a\rangle\, p \bowtie \langle b\rangle\, q) \\ = & (p - \langle a\rangle\, p) \bowtie q + p \bowtie (q - \langle b\rangle\, q) \\ = & (a \rhd p) \bowtie q + p \bowtie (b \rhd q)\ . \end{array}$$

Since both prioritisation and Pareto composition are defined as sums of joins, we can now use this together with Lemma 4.2.7, 4.2.1 and the exchange axiom of Definition 3.4.4 to calculate their maximal elements.

Lemma 5.6. *For $a :: T_a^2, p :: T_a$ and $b :: T_b^2, q :: T_b$ with $T_a \cap T_b = \emptyset$ we have*

$$\begin{array}{rcl} (a \lessdot\!\otimes b) \rhd (p \bowtie q) & = & (a \rhd p) \bowtie q\ , \\ (a \otimes\!\gtrdot b) \rhd (p \bowtie q) & = & p \bowtie (b \rhd q)\ , \\ (a \otimes b) \rhd (p \bowtie q) & = & (a \rhd p) \bowtie (b \rhd q)\ , \\ (a \,\&\, b) \rhd (p \bowtie q) & = & (a \rhd p) \bowtie (b \rhd q)\ . \end{array}$$

The proofs are straightforward and hence omitted.

Remark 5.7. It follows directly from the above lemma that

$$(a \,\&\, b) \rhd (p \bowtie q) \;=\; (b \,\&\, a) \rhd (q \bowtie p) \;=\; (a \otimes b) \rhd (p \bowtie q)\,,$$

i.e. Pareto composition and Prioritisation are identical on tests of the form $p \bowtie q$.

Note that this does not hold for general tests. Consider, for instance, the basic set $\{0,1\}^2$ and its subset $N =_{df} \{(0,1),(1,0)\}$, both represented by tests. Assume a preference order R_i in the i-th component which fulfills $0\, R_i\, 1$, for $i = 1,2$. Then $(R_1 \,\&\, R_2) \rhd N = \{(1,0)\}$, whereas $(R_1 \otimes R_2) \rhd N = N$. This does not contradict our above result, since N cannot be represented in the form $L \times M$ with $L, M \subseteq \{0,1\}$.

5.3 Equivalence of Preference Terms

Corollary 5.8. *Let $a :: T_a^2$ and $b, b' :: T_b^2$. Then we have:*

$$a \,\&\, (b + b') = a \,\&\, b + a \,\&\, b'.$$

Proof. Follows from definition of & and distributivity of $\bowtie$ over $+$.

Corollary 5.9. *For $a :: T_a^2$ we have $a \lessdot\!\otimes a \;=\; a \otimes\!\gtrdot a \;=\; a \otimes a \;=\; a$.*

Proof. $a \lessdot\!\otimes a \;=_{df}\; (a + 1_a) \bowtie a \;=\; (a + 1_a) \sqcap a \;=\; a$. For Right Semi-Pareto and Pareto an analogous argument shows the claim.

Theorem 5.10. *For $a :: T_a$ we have that $(a\,\&)$ distributes over $\mathbin{<\!\!\!\otimes}, \mathbin{\otimes\!\!\!>}$ and $\otimes$.*

Proof. Let $b :: T_b^2, c :: T_c^2$. We use the auxiliary equation (see Appendix B.4 for a proof)

$$a \,\&\, b + 1_{a \bowtie b} = a \,\&\, (b + 1_B). \tag{1}$$

Now we calculate:

$$\begin{aligned}
& (a \,\&\, b) \mathbin{\otimes\!\!\!>} (a \,\&\, c) \\
= \quad & \{\!\![\text{ definition of } \mathbin{\otimes\!\!\!>}]\!\!\} \\
& (a \,\&\, b + 1_{a \bowtie b}) \bowtie (a \,\&\, c) \\
= \quad & \{\!\![\text{ equation (1) }]\!\!\} \\
& (a \,\&\, (b + 1_b)) \bowtie (a \,\&\, c) \\
= \quad & \{\!\![\text{ definition of } \&]\!\!\} \\
& (a \bowtie \top_b + 1_a \bowtie (b + 1_b)) \bowtie (a \bowtie \top_c + 1_a \bowtie c) \\
= \quad & \{\!\![\text{ distributivity of } \bowtie]\!\!\} \\
& a \bowtie \top_b \bowtie a \bowtie \top_c \qquad + a \bowtie \top_b \bowtie 1_a \bowtie c + \\
& 1_a \bowtie (b + 1_b) \bowtie a \bowtie \top_c + 1_a \bowtie (b + 1_b) \bowtie 1_a \bowtie c \\
= \quad & \{\!\![\, a \bowtie a = a \text{ and } a \bowtie 1_a = a \sqcap 1_a, \text{ compare Corollary 2.8.2 }]\!\!\} \\
& a \bowtie \top_b \bowtie \top_c \qquad\qquad + (a \sqcap 1_a) \bowtie \top_b \bowtie c + \\
& (a \sqcap 1_a) \bowtie (b + 1_b) \bowtie \top_c + 1_a \bowtie (b + 1_b) \bowtie c \\
= \quad & \{\!\![\, a \sqcap 1_a \leq a,\, c \leq \top_c, \text{ subsumption order }]\!\!\} \\
& a \bowtie \top_b \bowtie \top_c + 1_a \bowtie (b + 1_b) \bowtie c \\
= \quad & \{\!\![\, \top_{b \bowtie c} = \top_b \bowtie \top_c, \text{ definition of } \&]\!\!\} \\
& a \,\&\, ((b + 1_b) \bowtie c) \\
= \quad & \{\!\![\text{ definition } \mathbin{\otimes\!\!\!>}]\!\!\} \\
& a \,\&\, (b \mathbin{\otimes\!\!\!>} c)
\end{aligned}$$

A symmetric argument holds for $\mathbin{<\!\!\!\otimes}$, so that $(a\,\&)$ distributes over $\mathbin{<\!\!\!\otimes}$ and $\mathbin{\otimes\!\!\!>}$. Using this we infer the distributivity over $\otimes$, see Appendix B.4 for details.

The proof of this theorem shows that the framework of typed relations is rich enough to prove non-trivial preference term equivalences.

We have proved this theorem using Prover9. The input for the auxiliary equation (1) can be found in Appendix A and the input for the entire theorem is given in [MR12].

Such equivalences are useful for an optimized evaluation of preferences, because the evaluation of an equivalent term may be faster.

6 Conclusion and Outlook

The present work intends to advance the state of the art in formalising preference algebra. Besides the point-wise "semi-formal" proofs by hand that had been

used originally we wanted to use automatic theorem provers like PROVER9 to get the theorems of preference algebra machine-checked. But we realized that there was no straightforward way to put theorems like the prefilter properties or the distributive law for Prioritisation/Pareto into a theorem prover. Especially for the latter problem, the main reason is that originally the equivalence of preference terms was defined, e.g. in [Kie02], in a very implicit manner: two preference terms are equivalent if and only if the corresponding relations are identical on the basic set. This definition is not very useful if one tries to find (automatically) general equivalence proofs.

The presented concept of a typed join algebra makes it possible to define such equivalences explicitly: two preference terms are identical, if and only if their algebraic representations are equal in the algebra.

Other theorems for which proofs are necessary do not just involve preference terms, but properties of the maximality operator and prefilters. With the inverse image we have employed a well-known algebraic concept to define the maximality operator in quantifier-free form. This reformulation led us to point-free proofs.

The relevance of this topic stems from the demand for optimizing the evaluation of preference queries (e.g. [KH03, HK05]. The paper [REM+12] presents a practical application of preference algebra, where complex preference terms and huge data sets occur, and therefore optimisation methods are of essential interest.

Our algebra is rich enough to cover the concept of preferences and their complex compositions as well as the application of the maximality operator to them. Simultaneously, the algebra is simple enough to be encoded in theorem provers like PROVER9.

With this we have produced a framework which hopefully will be the first step for a comprehensive algebraic description of preference algebra. Our work also covers some aspects of databases in general and thus contributes to the formal description of database-related problems. A project in which our calculus is applied systematically at a larger scale, using machine assistance, is under way.

Acknowledgements. We are grateful to Jeremy Gibbons (MPC Co-Chair) and the anonymous referees for valuable comments.

References

[BW93] Backhouse, R., van der Woude, J.: Demonic Operators and Monotype Factors. Mathematical Structures in Computer Science 3, 417–433 (1993)

[BD97] Bird, R., de Moor, O.: Algebra of programming. Prentice Hall (1997)

[BKS01] Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: Data Engineering (ICDE 2001), pp. 421–430 (2001)

[DMS06] Desharnais, J., Möller, B., Struth, G.: Kleene Algebra With Domain. ACM Transactions on Computational Logic 7, 798–833 (2006)

[End11] Endres, M.: Semi-Skylines and Skyline Snippets - Theory and Applications. Fakultät für Angewandte Informatik, Universität Augsburg, Dissertation. Books on Demand GmbH, Norderstedt (2011) ISBN: 978-3-8423-5246-9

[Kan90] Kanellakis, P.: Elements of Relational Database Theory. In: Handbook of Theoretical Computer Science. Formal Models and Semantics (B), vol. B, pp. 1073–1156 (1990)

[KEW11] Kießling, W., Endres, M., Wenzel, F.: The Preference SQL System – An Overview. IEEE Data Eng. Bull. 34(2), 11–18 (2011)

[KH03] Kießling, W., Hafenrichter, B.: Algebraic Optimization of Relational Preference Queries, Technical Report 2003-1, University of Augsburg (2003)

[HK05] Hafenrichter, B., Kießling, W.: Optimization of Relational Preference Queries. In: Williams, H., Dobbie, G. (eds.) Proc. Sixteenth Australasian Database Conference, ADC 2005, Database Technologies 2005, Newcastle, Australia, January 31-February 3. CRPIT, vol. 39, pp. 175–184. Australian Computer Society (2005)

[Kie02] Kießling, W.: Foundations of Preferences in Database Systems. In: Very Large Databases (VLDB 2002), pp. 311–322 (2002)

[Koz98] Kozen, D.: Typed Kleene algebra. Technical Report TR98-1669, Computer Science Department, Cornell University (March 1998)

[MO04] MacCaull, W., Orłowska, E.: A Calculus of Typed Relations. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 191–201. Springer, Heidelberg (2004)

[MB85] Manes, E., Benson, D.: The Inverse Semigroup of a Sum-Ordered Semiring. Semigroup Forum 31, 129–152 (1985)

[MR12] Möller, B., Roocks, P.: Proof of the Distributive Law for Prioritisation and Pareto Composition, http://www.informatik.uni-augsburg.de/lehrstuehle/dbis/pmi/staff/roocks/publications/distributivity_proof.pdf

[REM+12] Roocks, P., Endres, M., Mandl, S., Kießling, W.: Composition and Efficient Evaluation of Context-Aware Preference Queries. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part II. LNCS, vol. 7239, pp. 81–95. Springer, Heidelberg (2012)

[SS93] Schmidt, G., Ströhlein, T.: Relations and Graphs: Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science (1993)

[SHW97] Schmidt, G., Hattensperger, C., Winter, M.: Heterogeneous relation algebra. In: Brink, C., Kahl, W., Schmidt, G. (eds.) Relational Methods in Computer Science. Advances in Computer Science, pp. 39–53. Springer, Vienna (1997)

A Sample Prover Input

For $a :: T_a^2$ and $b :: T_b^2$ we show the auxiliary equation (1) from Theorem 5.10:

$$a \mathbin{\&} b + 1_{a \bowtie b} = a \mathbin{\&} (b + 1_b).$$

We use the following operators:

Prover-Input	mathematically
`a typed T_a`	$a :: T_a^2$
`a join b`	$a \bowtie b$
`T_a tjoin T_b`	$T_a \bowtie T_b$
`a prior b`	$a \,\&\, b$
`a + b`	$a + b$

The assumptions are given as follows:

```
% all elements are typed
exists T (x typed T).

% addition is associative, commutative and idempotent
(x + y) + z = x + (y + z).
x + y = y + x.
x + x = x.

% addition preserves type
x typed z & y typed z -> (x+y) typed z.

% subsumption order
x <= y <-> y = x + y.

% top is greatest element
x typed z -> x <= top(z).

% typing of top
top(z) typed z.
top(z1 tjoin z2) = top(z1) join top(z2).

% typing of one
one(z) typed z.
one(z1 tjoin z2) = one(z1) join one(z2).

% abbreviated typing
x typed z -> top(x) = top(z).
x typed z -> one(x) = one(z).

% distributivity of the join over addition
x join (y1 + y2) = x join y1 + x join y2.

% typing of join
x typed z1 & y typed z2 -> (x join y) typed (z1 tjoin z2).

% prioritisation (without resulting type)
x prior y = x join top(y) + one(x) join y.
```

Finally our goal is:

```
% auxiliary equation for distributive law
u prior v + one(u join v) = u prior (v + one(v)).
```

The entire input for the proof of theorem 5.10 can be found in [MR12].

B Proofs

In the proofs of section 4 we omit the type-index of 1 and assume type-compatibility.

B.1 Proof of Lemma 4.2

1. Immediate from the definitions and $1 - q = \neg q$.
2. Immediate from Part 1 and shunting.
3. $$\begin{array}{ll} & a \rhd p \\ = & \{\!\!\{ \text{ definitions of } \rhd \text{ and } - \}\!\!\} \\ & p \cdot \neg \langle a \rangle\, p \\ \leq & \{\!\!\{ \text{ property of intersection } \}\!\!\} \\ & p \,. \end{array}$$
4. $$\begin{array}{ll} & a \rhd 1 \leq a \rhd p \\ \Leftrightarrow & \{\!\!\{ \text{ definition of } \rhd \text{ and Part 1 } \}\!\!\} \\ & \neg \ulcorner a \leq p - \langle a \rangle\, p \\ \Leftrightarrow & \{\!\!\{ \text{ definition of } - \text{ and universal property of intersection } \}\!\!\} \\ & \neg \ulcorner a \leq p \quad \wedge \quad \neg \ulcorner a \leq \neg \langle a \rangle\, p \\ \Leftrightarrow & \{\!\!\{ \text{ shunting in second conjunct } \}\!\!\} \\ & \neg \ulcorner a \leq p \quad \wedge \quad \langle a \rangle\, p \leq \ulcorner a \\ \Leftrightarrow & \{\!\!\{ \text{ second conjunct true by (3.1) } \}\!\!\} \\ & \neg \ulcorner a \leq p \\ \Leftrightarrow & \{\!\!\{ \text{ definition of } \rhd \}\!\!\} \\ & a \rhd 1 \leq p \,. \end{array}$$
5. Immediate from the previous property by setting $p = a \rhd 1$.
6. $$\begin{array}{ll} & a \rhd (a \rhd p) \\ = & \{\!\!\{ \text{ definition of } \rhd \}\!\!\} \\ & (p - \langle a \rangle\, p) - \langle a \rangle\, (p - \langle a \rangle) \\ = & \{\!\!\{ \text{ property of difference } \}\!\!\} \\ & p - (\langle a \rangle\, p + \langle a \rangle\, (p - \langle a \rangle)) \\ = & \{\!\!\{ \text{ distributivity of } \langle\rangle \}\!\!\} \\ & p - \langle a \rangle\, (p + (p - \langle a \rangle)) \\ = & \{\!\!\{ \text{ since } p - \langle a \rangle \leq p \}\!\!\} \\ & p - \langle a \rangle\, p \\ = & \{\!\!\{ \text{ definition of } \rhd \}\!\!\} \\ & a \rhd p \,. \end{array}$$
7. $$\begin{array}{ll} & (a + b) \rhd p \\ = & \{\!\!\{ \text{ definition of } \rhd \}\!\!\} \\ & p - \langle a + b \rangle\, p \\ = & \{\!\!\{ \text{ distributivity of } \langle\rangle \}\!\!\} \\ & p - (\langle a \rangle\, p + \langle b \rangle\, p) \\ = & \{\!\!\{ \text{ property of difference } \}\!\!\} \\ & (p - \langle a \rangle\, p) \cdot (p - \langle b \rangle\, p) \end{array}$$

$$\begin{array}{ll} = & \{\!\![\text{ definition of } \triangleright \ ]\!\!\} \\ & (a \triangleright p) \cdot (b \triangleright p) \ . \end{array}$$

8. Assume $b \leq a$, i.e., $b + a = a$.

$$\begin{array}{ll} & a \triangleright p \\ = & \{\!\![\text{ assumption }]\!\!\} \\ & b + a \triangleright p \\ = & \{\!\![\text{ previous property }]\!\!\} \\ & (b \triangleright p) \cdot (a \triangleright p) \\ \leq & \{\!\![\text{ property of intersection }]\!\!\} \\ & b \triangleright p \ . \end{array}$$

9. By isotony of the diamond we have $p = \langle 1 \rangle\, p \leq \langle a \rangle\, p$ and hence $a \triangleright p = p - \langle a \rangle\, p = 0$.

B.2 Proof of Lemma 4.3

$$q = (p + \neg p) \cdot q = \overbrace{p \cdot q}^{=0} + \neg p \cdot q \leq \neg p$$
$$\neg p = \underbrace{(p + q)}_{=1} \cdot \neg p = p \cdot \neg p + q \cdot \neg p \leq q$$

By antisymmetry we have $\neg p = q$.

B.3 Proof of Theorem 4.6

We split the left-hand side of the claim equivalently into

$$b \,\mathsf{pref}\, a \;\Leftrightarrow\; a \triangleright 1 \leq a \triangleright (b \triangleright 1) \;\wedge\; a \triangleright (b \triangleright 1) \leq a \triangleright 1 \ .$$

By Parts 4 and 2 of Lemma 4.2 the first conjunct is equivalent to $\ulcorner b \leq \ulcorner a$. For the second conjunct we calculate

$$\begin{array}{ll} & a \triangleright (b \triangleright 1) \leq a \triangleright 1 \\ \Leftrightarrow & \{\!\![\text{ definition of } \triangleright \text{ and Lemma 4.2.1 }]\!\!\} \\ & \neg \ulcorner b - \langle a \rangle\, \neg \ulcorner b \leq \neg \ulcorner a \\ \Leftrightarrow & \{\!\![\text{ contraposition and De Morgan }]\!\!\} \\ & \ulcorner a \leq \ulcorner b + \langle a \rangle\, \neg \ulcorner b \ . \end{array}$$

B.4 Proof of Theorem 5.10

Auxiliary equation (1):

$$
\begin{aligned}
& a \mathbin{\&} b + 1_{a \bowtie b} \\
= & \quad \{\!\![\text{ definition of } \& \,]\!\!\} \\
& a \bowtie \top_b + 1_a \bowtie b + 1_a \bowtie 1_b \\
= & \quad \{\!\![\text{ distributivity of } \bowtie \,]\!\!\} \\
& a \bowtie \top_b + 1_a \bowtie (b + 1_b) \\
= & \quad \{\!\![\text{ definition of } \& \,]\!\!\} \\
& a \mathbin{\&} (b + 1_b)
\end{aligned}
$$

Distributivity of $(a\,\&)$ over $\otimes$:

$$
\begin{aligned}
& a \mathbin{\&} (b \otimes c) \\
= & \quad \{\!\![\text{ definition of } \otimes \,]\!\!\} \\
& a \mathbin{\&} (b \mathbin{\lessdot\!\!\otimes} c + b \mathbin{\otimes\!\!\gtrdot} c) \\
= & \quad \{\!\![\text{ distributivity of } \& \text{ over } +\text{, cor. 5.8 }]\!\!\} \\
& a \mathbin{\&} (b \mathbin{\lessdot\!\!\otimes} c) + a \mathbin{\&} (b \mathbin{\otimes\!\!\gtrdot} c) \\
= & \quad \{\!\![\text{ distributivity of } (a\,\&) \text{ over } \lessdot\!\!\otimes \text{ and } \otimes\!\!\gtrdot \,]\!\!\} \\
& (a \mathbin{\&} b) \mathbin{\lessdot\!\!\otimes} (a \mathbin{\&} c) + (a \mathbin{\&} b) \mathbin{\otimes\!\!\gtrdot} (a \mathbin{\&} c) \\
= & \quad \{\!\![\text{ definition of } \otimes \,]\!\!\} \\
& (a \mathbin{\&} b) \otimes (a \mathbin{\&} c)
\end{aligned}
$$

Modular Tree Automata

Patrick Bahr

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract. Tree automata are traditionally used to study properties of tree languages and tree transformations. In this paper, we consider tree automata as the basis for modular and extensible recursion schemes. We show, using well-known techniques, how to derive from standard tree automata highly modular recursion schemes. Functions that are defined in terms of these recursion schemes can be combined, reused and transformed in many ways. This flexibility facilitates the specification of complex transformations in a concise manner, which is illustrated with a number of examples.

1 Introduction

Functional programming languages are an excellent tool for specifying abstract syntax trees (ASTs) and defining syntax-directed transformations on them: algebraic data types provide a compact notation for both defining types of ASTs as well as constructing and manipulating ASTs. As a complement to that, recursively defined functions on algebraic data types allow us to traverse ASTs defined by algebraic data types.

For example, writing an evaluation function for a small expression language is easily achieved in Haskell [19] as follows:

$$\mathbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp}$$

$$\begin{aligned} &\mathit{eval} :: \mathit{Exp} \to \mathit{Int}\\ &\mathit{eval}\ (\mathit{Val}\ i) \quad\ \ = i\\ &\mathit{eval}\ (\mathit{Plus}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y \end{aligned}$$

Unfortunately, this simple approach does not scale very well. As soon as we have to implement more complex transformations that work on more than just a few types of ASTs, simple recursive function definitions become too inflexible and complicated.

Specifying and implementing such transformations is an everyday issue for compiler construction and thus has prompted a lot of research in this area. One notable approach to address both sides is the use of attribute grammars [15, 22]. These systems facilitate compact specification and efficient implementation of syntax-directed transformations.

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 263–299, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper, we take a different but not unrelated approach. We still want to implement the transformations in a functional language. But instead of writing transformation functions as general recursive functions as the one above, our goal is to devise *recursion schemes*, which can then be used to define the desired transformations. The use of these recursion schemes will allow us reuse, combine and reshape the syntax-directed transformations that we write. In addition, the embedding into a functional language will give us a lot of flexibility and expressive power such as a powerful type system and generic programming techniques.

As a starting point for our recursion schemes we consider various kinds of tree automata [3]. For each such kind we show how to implement them in Haskell. From the resulting recursion schemes we then derive more sophisticated and highly modular recursion schemes. In particular, our contributions are the following:

- We implement bottom-up tree acceptors (Section 2), bottom-up tree transducers (Section 4) and top-down tree transducers (Section 5) as recursion schemes in Haskell. While the implementation of the first two is well-known, the implementation of the last one is new but entirely straightforward.
- From the thus obtained recursion schemes, we derive more modular variants (Section 3) using a variation of the well-know product automaton construction (Section 3.1) and Swierstra's *data types à la carte* [23] (Section 3.2).
- We decompose the recursion schemes derived from bottom-up and from top-down tree transducer into a homomorphism part and a state transition part (Section 4.5 and Section 5.3). This makes it possible to specify these two parts independently and to modify and combine them in a flexible manner.
- We derive a recursion scheme that combines both bottom-up and top-down state propagation (Section 6).
- We illustrate the merit of our recursion schemes by a running example in which we develop a simple compiler for a simple expression language. Utilising the modularity of our approach, we extend the expression language throughout the paper in order to show how the more advanced recursion schemes help us in devising an increasingly more complex compiler. In addition to that, the high degree of modularity of our approach not only simplifies the construction of the compiler but also allows us to reuse earlier iterations of the compiler.

Apart from the abovementioned running example, we also include a number of independent examples illustrating the mechanics of the presented tree automata.

The remainder of this paper is structured as follows: we start in Section 2 with bottom-up tree acceptors and their implementation in Haskell. In Section 3, we introduce two dimensions of modularity that can be exploited in the recursion scheme obtained from bottom-up tree acceptors. In Section 4, we will turn to bottom-up tree transducers, which, based on a state that is propagated upwards, perform a transformation of an input term to an output term. In Section 4.5 we will then introduce yet another dimension of modularity by separating the state propagation in tree transducers from the tree transformation. This will also allow

us to adopt the modularity techniques from Section 3. In Section 5, we will do the same thing again, however, for top-down tree transducers in which the state is propagated top-down rather than bottom-up. Finally, in Section 6, we will combine both bottom-up and top-down state transitions.

The library of recursion schemes that we develop in this paper is available as part of the `compdata` package [2]. Additionally, this paper is written as a literate Haskell file[1], which can be directly loaded into the GHCi Haskell interpreter.

2 Bottom-Up Tree Acceptors

The tree automata that we consider in this paper operate on terms over some signature $\mathcal{F}$. In the setting of tree automata, a signature $\mathcal{F}$ is simply a set of function symbols with a fixed arity and we write $f/n \in \mathcal{F}$ to indicate that f is a function symbol in $\mathcal{F}$ of arity n. Given a signature $\mathcal{F}$ and some set $\mathcal{X}$, the set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set T such that $\mathcal{X} \subseteq T$ and if $f/n \in \mathcal{F}$ and $t_1, \dots, t_n \in T$ then $f(t_1, \dots, t_n) \in T$. Instead of $\mathcal{T}(\mathcal{F}, \emptyset)$ we also write $\mathcal{T}(\mathcal{F})$ and call elements of $\mathcal{T}(\mathcal{F})$ terms over $\mathcal{F}$. Tree automata run on terms in $\mathcal{T}(\mathcal{F})$.

Each of the tree automata that we describe in this paper consists at least of a finite set Q of states and a set of rules according to which an input term is transformed into an output term. While performing such a transformation, these automata maintain state information, which is stored in the intermediate results of the transformation. To this end each state $q \in Q$ is considered as a unary function symbol and a subterm t is annotated with state q by writing $q(t)$. For example, $f(q_0(a), q_1(b))$ represents the term $f(a, b)$, where the two subterms a and b are annotated with states q_0 and q_1, respectively.

The rules of the tree automata in this paper will all be of the form $l \rightarrow r$ with $l, r \in \mathcal{T}(\mathcal{F}', \mathcal{X})$, where $\mathcal{F}' = \mathcal{F} \uplus \{q/1 \mid q \in Q\}$. The rules can be read as term rewrite rules, i.e. the variables in l and t are placeholders that are instantiated with terms when the rule is applied. Running an automaton is then simply a matter of applying these term rewrite rules to a term. The different kinds of tree automata only differ in the set of rules they allow.

2.1 Deterministic Bottom-Up Tree Acceptors

A *deterministic bottom-up tree acceptor* (DUTA) over a signature $\mathcal{F}$ consists of a (finite) set of states Q, a set of accepting states $Q_a \subseteq Q$, and a set of transition rules of the form

$$f(q_1(x_1), \dots, q_n(x_n)) \rightarrow q(f(x_1, \dots, x_n)), \quad \text{with } f/n \in \mathcal{F} \text{ and } q, q_1, \dots, q_n \in Q$$

The variable symbols $x_1, \dots, x_n$ serve as placeholders in these rules and states in Q are considered as function symbols of arity 1. The set of transition rules must be deterministic – i.e. there are no two different rules with the same left-hand

[1] Available from the author's web site.

side – and complete – i.e. for each $f/n \in \mathcal{F}$ and $q_1, \ldots, q_n \in Q$, there is a rule with the left-hand side $f(q_1(x_1), \ldots, q_n(x_n))$. The state q on the right-hand side of the transition rule is also called the *successor state* of the transition.

By repeatedly applying the transition rules to a term t over $\mathcal{F}$, initial states are created at the leaves which then get propagated upwards through function symbols. Eventually, we obtain a final state q_f at the root of the term. That is, an input term t is transformed into $q_f(t)$. The term t is *accepted* by the DUTA iff $q_f \in Q_a$. In this way, a DUTA defines a term language.

Example 1. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0\}$ and the DUTA over $\mathcal{F}$ with $Q = \{q_0, q_1\}$, $Q_a = \{q_1\}$ and the following transition rules:

$$\begin{array}{lll} \mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{not}(x)) & \mathsf{and}(q_0(x), q_1(y)) \to q_0(\mathsf{and}(x,y)) \\ \mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{not}(x)) & \mathsf{and}(q_1(x), q_0(y)) \to q_0(\mathsf{and}(x,y)) \\ \multicolumn{2}{c}{\mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{and}(x,y))} & \mathsf{and}(q_0(x), q_0(y)) \to q_0(\mathsf{and}(x,y)) \end{array}$$

Terms over signature $\mathcal{F}$ are Boolean expressions and the automaton accepts such an expression iff it evaluates to true.

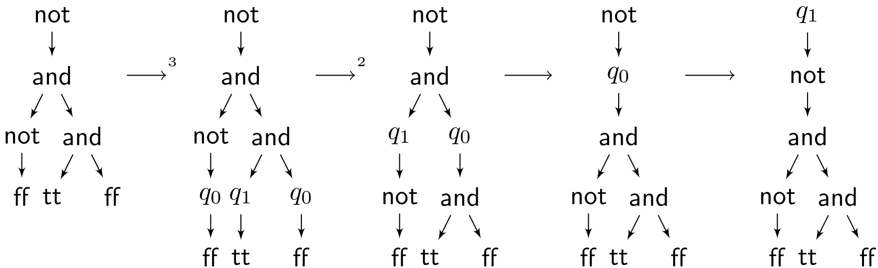
Note that the rules are complete – for each function symbol, every combination of input states occurs in the left-hand side of some rule – and deterministic – there are no two rules with the same left-hand side.

The transition rules are applied by interpreting them as rules in a term rewriting system, where variables are placeholders for terms. For the term $\mathsf{and}(\mathsf{tt}, \mathsf{ff})$, we get the following derivation:

$$\mathsf{and}(\mathsf{tt}, \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), \mathsf{ff}) \to \mathsf{and}(q_1(\mathsf{tt}), q_0(\mathsf{ff})) \to q_0(\mathsf{and}(\mathsf{tt}, \mathsf{ff}))$$

The result of this derivation is the final state q_0; the term is rejected.

The following picture illustrates a run of the automaton on the bigger term $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{ff}), \mathsf{and}(\mathsf{tt}, \mathsf{ff})))$:



For the sake of conciseness, we applied rules in parallel where possible. At first we apply the rules to the leaves of the term, performing three rewrite steps in parallel. This effectively produces the initial states of the run. Subsequent rule applications propagate the states according to the rules until we obtain the final state at the root of the term.

Note that in both runs, apart from the final state at the root, the result term is the same as the one we started with. This is expected. The only significant output of a DUTA run is the final state.

The rules of a DUTA contain some syntactic overhead as they explicitly copy the function symbol from the left-hand side to the right-hand side. This formulation serves two purposes: first, it makes it possible to describe the run of a DUTA as a term reduction as in the above example. Secondly, we will see that the more sophisticated automata that we will consider later are simply generalisations of the rules of a DUTA, which for example do not require copying the function symbol but allow arbitrary transformations.

2.2 Algebras and Catamorphisms

For the representation of recursion schemes in Haskell, we consider data types as fixed points of polynomial functors:

$$\textbf{data}\ \mathit{Term}\ f = \mathit{In}\ (f\ (\mathit{Term}\ f))$$

Given a functor f that represents some signature, *Term* f constructs its fixed point, which represents the terms over f. For example, the data type *Exp* from the introduction may be instead defined as *Term Sig* with[2]

$$\textbf{data}\ \mathit{Sig}\ e = \mathit{Val}\ \mathit{Int} \mid \mathit{Plus}\ e\ e$$

The functoriality of *Sig* is given by an instance of the type class *Functor*:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{Sig}\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{Val}\ i) \quad\ \ = \mathit{Val}\ i\\
&\quad \mathit{fmap}\ f\ (\mathit{Plus}\ x\ y) = \mathit{Plus}\ (f\ x)\ (f\ y)
\end{aligned}$$

The function *eval* from the introduction is defined by a simple recursion scheme: its recursive definition closely follows the recursive definition of the data type *Exp*. This recursion scheme is known as *catamorphism* (or also *fold*). Given an *algebra*, i.e. a functor f and type a together with a function of type $f\ a \to a$, its catamorphism is a function of type $\mathit{Term}\ f \to a$ constructed as follows:

$$\begin{aligned}
&\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to (\mathit{Term}\ f \to a)\\
&\mathit{cata}\ \phi\ (\mathit{In}\ t) = \phi\ (\mathit{fmap}\ (\mathit{cata}\ \phi)\ t)
\end{aligned}$$

In the definition of the algebra for the evaluation function, we make use of the fact that the arguments of the *Plus* constructor are already the results of evaluating the corresponding subexpressions:

$$\begin{aligned}
&\mathit{evalAlg} :: \mathit{Sig}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{evalAlg}\ (\mathit{Val}\ i) \quad\ \ = i\\
&\mathit{evalAlg}\ (\mathit{Plus}\ x\ y) = x + y
\end{aligned}
\qquad
\begin{aligned}
&\mathit{eval} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Int}\\
&\mathit{eval} = \mathit{cata}\ \mathit{evalAlg}
\end{aligned}$$

Programming in algebras and catamorphisms or other algebraic or coalgebraic recursion schemes is a well-known technique in functional programming [20]. We shall use this representation in order to implement the recursion schemes that we derive from the tree automata.

[2] *Term Sig* is "almost" isomorphic to *Exp*. The only difference stems from the fact that the constructor *In* is non-strict.

2.3 Bottom-Up State Transition Functions

If we omit the syntactic overhead of the state transition rules of DUTAs, we see that DUTAs are algebras – in fact, they were originally defined as such [5]. For instance, the algebra of the automaton in Example 1 is an algebra that evaluates Boolean expressions. Speaking in Haskell terms, a DUTA over a signature functor F is given by a type of states Q, a state transition function in the form of an F-algebra $trans :: F\ Q \to Q$, and a predicate $acc :: Q \to Bool$. A term over F is an element of type $Term\ F$. When running a DUTA on a term t of type $Term\ F$, we obtain the final state $cata\ trans\ t$ of the run. Afterwards, the predicate acc checks whether the final state is accepting:

```
runDUTA :: Functor f => (f q -> q) -> (q -> Bool) -> Term f -> Bool
runDUTA trans acc = acc . cata trans
```

Example 2. We implement the DUTA from Example 1 in Haskell as follows:

```
data F a = And a a                trans :: F Q -> Q
         | Not a                  trans FF             = Q0
         | TT | FF                trans TT             = Q1
data Q = Q0 | Q1                  trans (Not Q0)       = Q1
acc :: Q -> Bool                  trans (Not Q1)       = Q0
acc Q1 = True                     trans (And Q1 Q1)    = Q1
acc Q0 = False                    trans (And _  _ )    = Q0
```

The automaton is run on a term of type $Term\ F$ as follows:

```
evalBool :: Term F -> Bool
evalBool = runDUTA trans acc
```

The restriction to a finite state space is not crucial for our purposes as we are not interested in deciding properties of automata. Instead, we want to use automata as powerful recursion schemes that allow for modular definitions of functions on terms. Since we are only interested in the traversal of the term that an automaton provides, we also drop the predicate and consider the final state as the output of a run of the automaton. We, therefore, consider only the transition function of a DUTA:

```
type UpState f q = f q -> q
runUpState :: Functor f => UpState f q -> Term f -> q
runUpState = cata
```

With the functions *evalAlg* from Section 2.2 and *trans* from Example 2, we have already seen two simple examples of bottom-up state transition functions. In practice, only few state transitions of interest are that simple, of course.

In the following, we want to write a simple compiler for our expression language that generates code for a simple virtual machine with a single accumulator register and a random access memory indexed by non-negative integers. At first, we devise the instructions of the virtual machine:

```
type Addr = Int
data Instr = Acc Int | Load Addr | Store Addr | Add Addr
type Code = [Instr]
```

For simplicity, we use integers to represent addresses for the random access memory. The four instructions listed above write an integer constant to the accumulator, load the contents of a memory cell into the accumulator, store the contents of the accumulator into a memory cell, and add the contents of a memory cell to the contents of the accumulator, respectively.

The code that we want to produce for an expression e of type *Term Sig* should evaluate e, i.e. after executing the code, the virtual machine's accumulator is supposed to contain the integer value *eval* e:

```
codeSt :: UpState Sig Code
codeSt (Val i)      = [Acc i]
codeSt (Plus x y) = x ++ [Store a] ++ y ++ [Add a]
  where a = ...
```

In order to perform addition, the result of the computation for the first summand has to be stored into a temporary memory cell at some address a. However, we also have to make sure that this memory cell is not overwritten by the computation for the second summand. To this end, we maintain a counter that tells us which address is safe to use:

```
codeAddrSt :: UpState Sig (Code, Addr)
codeAddrSt (Val i)                    = ([Acc i], 0)
codeAddrSt (Plus (x, a') (y, a)) = (x ++ [Store a] ++ y ++ [Add a],
                                        1 + max a a')

code :: Term Sig → Code
code = fst . runUpState codeAddrSt
```

While this definition yields the desired code generator, it is not very elegant as it mixes the desired output state – the code – with an auxiliary state – the fresh address. This flaw can be mitigated by using a state monad to carry around the auxiliary state. In this way we can still benefit from computing both states side by side has, which means that the input term is only traversed once.

This however still leaves the specification of two computations uncomfortably entangled, which is not only more prone to errors but also inhibits reuse and flexibility: the second component of the state, which we use as a fresh address, is in fact the height of the expression and might be useful for other computations:

```
heightSt :: UpState Sig Int
heightSt (Val _)      = 0
heightSt (Plus x y) = 1 + max x y
```

Moreover, as we extend the expression language with new language features, we might have to change the way we allocate memory locations for intermediate

results. Thus, separating the two components of the computation is highly desirable since it would then allow us to replace the *heightSt* component with a different one while reusing the rest of the code generator.

The next section addresses this concern.

3 Making Tree Automata Modular

Our goal is to devise modular recursion schemes. In this section, we show how to leverage two dimensions of modularity inherent in tree automata, viz. the state space and the signature. For each dimension, we present a well-know technique to make use of the modularity in the specification of automata. In particular, we shall demonstrate these techniques on bottom-up state transitions. However, due to their generality, both techniques are applicable also to the more advanced tree automata that we consider in later sections.

3.1 Product Automata

A common construction in automata theory combines two automata by simply forming the cartesian product of their state spaces and defining the state transition componentwise according to the state transitions of the original automata. The resulting automaton runs the original automata in parallel. We shall follow the same idea to construct the state transition *codeAddrSt* from Section 2.3 by combining the state transition *heightSt* with a state transition that computes the machine code using the state maintained by *heightSt*.

However, in contrast to the standard product automaton construction, the two computations in our example are not independent from each other – the code generator depends on the height in order to allocate memory addresses. Therefore, we need a means of communication between the constituent automata.

In order to allow access to components of a compound state space, we define a binary type class $\in$ that tells us if a type is a component of a product type and provides a projection for that component:

$$\textbf{class}\ a \in b\ \textbf{where}\\ \quad pr :: b \to a$$

Using *overlapping instance declarations*, we define the relation $a \in b$ as follows:

$$\begin{array}{lll} \textbf{instance} & a \in a & \textbf{where}\ pr = id \\ \textbf{instance} & a \in (a, b) & \textbf{where}\ pr = fst \\ \textbf{instance}\ (c \in b) \Rightarrow & c \in (a, b) & \textbf{where}\ pr = pr\ .\ snd \end{array}$$

That is, we have $a \in b$ if b is of the form $(b_1, (b_2, ...))$ and $a = b_i$ for some i.

We generalise bottom-up state transitions by allowing the successor state of a transition to be dependent on a potentially larger state space:

$$\textbf{type}\ DUpState\ f\ p\ q = (q \in p) \Rightarrow f\ p \to q$$

The result state of type q for the state transition of the above type may depend on the states that are propagated from below. However, in contrast to ordinary bottom-up state transitions, these states – of type p – may contain more components in addition to the component of type q.

Every ordinary bottom-up state transition such as *heightSt* can be readily converted into such a *dependent bottom-up state transition function* by precomposing the projection *pr*:

$$\begin{aligned}&\mathit{dUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{UpState}\ f\ q \rightarrow \mathit{DUpState}\ f\ p\ q\\&\mathit{dUpState}\ \mathit{st} = \mathit{st}\ .\ \mathit{fmap}\ \mathit{pr}\end{aligned}$$

A dependent state transition function is the same as an ordinary state transition function if the state spaces p and q coincide. Hence, we can run such a dependent state transition function in the same way:

$$\begin{aligned}&\mathit{runDUpState} :: \mathit{Functor}\ f \Rightarrow \mathit{DUpState}\ f\ q\ q \rightarrow \mathit{Term}\ f \rightarrow q\\&\mathit{runDUpState}\ f = \mathit{runUpState}\ f\end{aligned}$$

When defining a dependent state transition function, we can make use of the fact that the state propagated from below may contain additional components. For the definition of the state transition function generating the code, we declare that we expect an additional state component of type *Int*.

$$\begin{aligned}&\mathit{codeSt} :: (\mathit{Int} \in q) \Rightarrow \mathit{DUpState}\ \mathit{Sig}\ q\ \mathit{Code}\\&\mathit{codeSt}\ (\mathit{Val}\ i) \quad\ \ = [\mathit{Acc}\ i]\\&\mathit{codeSt}\ (\mathit{Plus}\ x\ y) = \mathit{pr}\ x +\!\!+ [\mathit{Store}\ a] +\!\!+ \mathit{pr}\ y +\!\!+ [\mathit{Add}\ a]\\&\quad \textbf{where}\ a = \mathit{pr}\ y\end{aligned}$$

Using the method *pr* of the type class $\in$, we project to the desired components of the state: $pr\ x$ and the first occurrence of $pr\ y$ are of type *Code* whereas the second occurrence of $pr\ y$ is of type *Int*.

The product construction that combines two dependent state transition functions is simple: it takes two state transition functions depending on the same (compound) state space and combines them by forming the product of their respective outcomes:

$$\begin{aligned}&(\otimes) :: (p \in c, q \in c) \Rightarrow \mathit{DUpState}\ f\ c\ p \rightarrow \mathit{DUpState}\ f\ c\ q\\&\qquad\qquad\qquad\qquad\quad \rightarrow \mathit{DUpState}\ f\ c\ (p, q)\\&(\mathit{sp} \otimes \mathit{sq})\ t = (\mathit{sp}\ t, \mathit{sq}\ t)\end{aligned}$$

We obtain the desired code generator from Section 2.3 by combining our two (dependent) state transition functions and running the resulting state transition function:

$$\begin{aligned}&\mathit{code} :: \mathit{Term}\ \mathit{Sig} \rightarrow \mathit{Code}\\&\mathit{code} = \mathit{fst}\ .\ \mathit{runDUpState}\ (\mathit{codeSt} \otimes \mathit{dUpState}\ \mathit{heightSt})\end{aligned}$$

Note that combining state transition functions in this way is not restricted to such simple dependencies. State transition functions may depend on each other.

The construction that we have seen in this section makes it possible to decompose state spaces into isolated modules with a typed interface to access them. This practice of decomposing state spaces is not different from the abstraction and reuse that we perform when writing mutual recursive functions. Functions which can be defined in this way are also known as *mutumorphisms* [6].

There are still two minor shortcomings, which we shall address when we consider other types of automata below. First, the extraction of components from compound states is purely based on the type information, which can easily result in confusion of distinct state components that happen to have the same type. This can be seen in the instance declarations for the type class $\in$, which are overlapping and will simply select the left-most occurrence of a type. Secondly, we only allow access to the state of the children of the current node. In principle, this restriction is no problem as we can use the states of the children nodes to compute the state of the current node. For example, if, in the code generation, we needed the height of the current expression instead of the height of the right summand, we could have computed it from the height of both summands. However, this means that code as well as the corresponding computations are duplicated since the state of the current node is already computed by the corresponding state transition.

3.2 Compositional Data Types

We also want to leverage the modularity that stems from the data types on which we want to define functions. This modularity is based on the ability to combine functors by forming coproducts:

$$
\begin{aligned}
&\mathbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ \ e) \mid \mathit{Inr}\ (g\ \ e)\\
&\mathbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (f \oplus g)\ \mathbf{where}\\
&\quad \mathit{fmap}\ f\ \ (\mathit{Inl}\ \ e) = \mathit{Inl}\ (\mathit{fmap}\ f\ \ e)\\
&\quad \mathit{fmap}\ f\ \ (\mathit{Inr}\ \ e) = \mathit{Inr}\ (\mathit{fmap}\ f\ \ e)
\end{aligned}
$$

Using the $\oplus$ operator, we can extend the signature functor *Sig* with an increment operation, for example:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Inc}\ e = \mathit{Inc}\ e\\
&\mathbf{type}\ \mathit{Sig}'\ \ = \mathit{Inc} \oplus \mathit{Sig}
\end{aligned}
$$

In order to make use of this composition of functors for defining automata on functors in a modular fashion, we will follow Swierstra's *data types à la carte* [23] approach, which we will summarise briefly below.

The use of coproducts entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the correct tags automatically, injections are derived using a type class:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{sub} \preceq \mathit{sup}\ \mathbf{where}\\
&\quad \mathit{inj} :: \mathit{sub}\ a \to \mathit{sup}\ a
\end{aligned}
$$

Similarly to the type class $\in$, we define the subsignature relation $\preceq$ as follows:

$$
\begin{array}{lll}
\textbf{instance} & f \preceq f & \textbf{where } inj = id \\
\textbf{instance} & f \preceq (f \oplus g) & \textbf{where } inj = Inl \\
\textbf{instance } (f \preceq g) \Rightarrow & f \preceq (h \oplus g) & \textbf{where } inj = Inr \mathbin{.} inj
\end{array}
$$

That is, we have $f \preceq g$ if g is of the form $g_1 \oplus (g_2 \oplus ...)$ and $f = g_i$ for some i. From the injection function *inj*, we derive an injection function for terms:

$$
\begin{array}{l}
inject :: (g \preceq f) \Rightarrow g\ (Term\ f) \rightarrow Term\ f \\
inject = In \mathbin{.} inj
\end{array}
$$

Additionally, in order to reduce syntactic overhead, we assume, for each signature functor such as *Sig* or *Inc*, smart constructors that comprise the injection, e.g.:

$$
\begin{array}{l}
plus :: (Sig \preceq f) \Rightarrow Term\ f \rightarrow Term\ f \rightarrow Term\ f \\
plus\ x\ y = inject\ (Plus\ x\ y) \\
inc :: (Inc \preceq f) \Rightarrow Term\ f \rightarrow Term\ f \\
inc\ x = inject\ (Inc\ x)
\end{array}
$$

Using these smart constructors, we can write, for example, $inc\ (val\ 3\,`plus`\,val\ 4)$ to denote the expression $inc(3 + 4)$.

For writing modular functions on compositional data types, we use type classes. For example, for recasting the definition of the *heightSt* state transition function, we introduce a new type class and make it propagate over coproducts:

$$
\begin{array}{l}
\textbf{class } HeightSt\ f\ \textbf{where} \\
\quad heightSt :: UpState\ f\ Int \\
\textbf{instance } (HeightSt\ f, HeightSt\ g) \Rightarrow HeightSt\ (f \oplus g)\ \textbf{where} \\
\quad heightSt\ (Inl\ x) = heightSt\ x \\
\quad heightSt\ (Inr\ x) = heightSt\ x
\end{array}
$$

The above instance declaration lifts instances of *HeightSt* over coproducts in a straightforward manner. Subsequently, we will omit these instance declarations as they always follow the same pattern and thus can be generated automatically like instances declarations for *Functor*.

We then instantiate this class for each (atomic) signature functor separately:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{instance } HeightSt\ Sig\ \textbf{where}} \\
\quad heightSt\ (Val\ _) & = 0 \\
\quad heightSt\ (Plus\ x\ y) & = 1 + max\ x\ y \\
\multicolumn{2}{l}{\textbf{instance } HeightSt\ Inc\ \textbf{where}} \\
\quad heightSt\ (Inc\ x) & = 1 + x
\end{array}
$$

Due to the propagation of instances over coproducts, we obtain an instance of *HeightSt* for *Sig′* for free.

With the help of the type class *HeightSt*, we eventually obtain an extensible definition of the height function.

$$height :: (Functor\ f, HeightSt\ f) \Rightarrow Term\ f \rightarrow Int$$
$$height = runUpState\ heightSt$$

Since we have instantiated *HeightSt* for the signature Sig' and all its subsignatures, the function *height* may be given any argument of type $Term\ f$, where f is the Sig' or any of its subsignatures. Moreover, by simply providing further instance declarations for *HeightSt*, we can extend the domain of *height* to further signatures.

4 Bottom-Up Tree Transducers

A compiler usually consists of several stages that perform diverse kinds of transformations on the abstract syntax tree, e.g. renaming variables or removing syntactic sugar. Representing syntax trees as terms, i.e. values of type $Term\ f$, such transformations are functions of type $Term\ f \rightarrow Term\ g$ that map terms over some signature to terms over a potentially different signature. Tree transducers are a well-established technique for specifying such transformations [3, 7]. Moreover, there are a number of composition theorems that permit the composition of certain tree transducers such that the transformation function denoted by the composition is equal to the composition of the transformation functions denoted by the original tree transducers [7]. These composition theorems permit us to perform deforestation [26], i.e. eliminating intermediate results by fusing several stages of a compiler to a single tree transducer [16, 25], thus making tree transducers an attractive recursion scheme.

4.1 Deterministic Bottom-Up Tree Transducers

A *deterministic bottom-up tree transducer* (*DUTT*) defines – like a DUTA – for each function symbol a successor state. But, additionally, it also defines an expression that should replace the original function symbol. More formally, a DUTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q and a set of transduction rules of the form

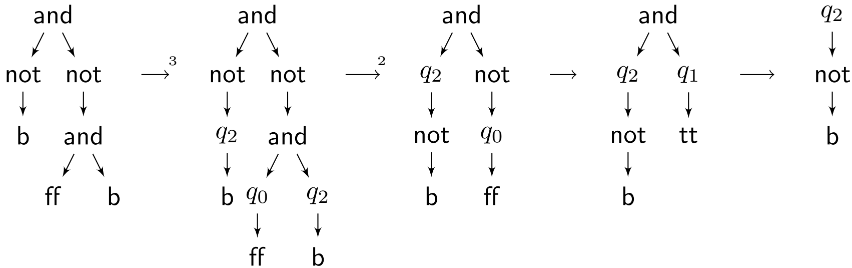
$$f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(u), \quad \text{with } f \in \mathcal{F} \text{ and } q, q_1, \ldots, q_n \in Q$$

where $u \in \mathcal{T}(\mathcal{G}, \mathcal{X})$ is a term over signature $\mathcal{G}$ and the set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$. Compare this to the state transition rules of DUTAs, which are simply a restriction of the transduction rules above with $u = f(x_1, \ldots, x_n)$, thus only allowing the identity transformation. By repeatedly applying its transduction rules in a bottom-up fashion, a run of a DUTT transforms an input term over $\mathcal{F}$ into an output term over $\mathcal{G}$ plus – similarly to DUTAs – a final state at the root.

Example 3. Consider the signature $\mathcal{F} = \{\mathsf{and}/2, \mathsf{not}/1, \mathsf{ff}/0, \mathsf{tt}/0, \mathsf{b}/0\}$ and the DUTT from $\mathcal{F}$ to $\mathcal{F}$ with $Q = \{q_0, q_1, q_2\}$ and the following transduction rules:

$$\begin{array}{lll}
\mathsf{tt} \to q_1(\mathsf{tt}) & \mathsf{not}(q_0(x)) \to q_1(\mathsf{tt}) & \mathsf{and}(q_1(x), q_1(y)) \to q_1(\mathsf{tt}) \\
\mathsf{ff} \to q_0(\mathsf{ff}) & \mathsf{not}(q_1(x)) \to q_0(\mathsf{ff}) & \mathsf{and}(q_1(x), q_2(y)) \to q_2(y) \\
\mathsf{b} \to q_2(\mathsf{b}) & \mathsf{not}(q_2(x)) \to q_2(\mathsf{not}(x)) & \mathsf{and}(q_2(x), q_1(y)) \to q_2(x) \\
\multicolumn{2}{l}{\mathsf{and}(q(x), p(y)) \to q_0(\mathsf{ff}) \quad \text{if } q_0 \in \{p, q\}} & \mathsf{and}(q_2(x), q_2(y)) \to q_2(\mathsf{and}(x, y))
\end{array}$$

The signature $\mathcal{F}$ allows us to express Boolean expression containing a single Boolean variable b. When applied to such an expression, the automaton performs constant folding, i.e. it evaluates subexpression if possible. With the states q_0 and q_1 it signals that a subexpression is false respectively true; q_2 indicates uncertainty. For example, applying the automaton to the expression $\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{not}(\mathsf{and}(\mathsf{ff}, \mathsf{b})))$ yields the following derivation:



The rules for the constant symbols do not perform any transformation in this example and simply provide initial states. Then the first real transformation is performed, which collapses the subterm rooted in and to $q_0(\mathsf{ff})$. The run of the automaton is completed as soon as a state appears at the root, the final state of the run.

4.2 Contexts in Haskell

In order to, represent transduction rules in Haskell, we need a representation of the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of terms over signature $\mathcal{F}$ and variables $\mathcal{X}$. We call such extended terms *contexts*. These contexts appear on the right-hand side of transduction rules of DUTTs. We obtain a representation of contexts by simply extending the definition of the data type *Term* by an additional constructor:

```
data Context f a = In (f (Context f a)) | Hole a
```

We call this additional constructor *Hole* as we will use it also for things other than variables. For example, the holes in a context may be filled by other contexts over the same signature. The following function substitutes the contexts in the holes into the surrounding context.

```
appCxt :: Functor f ⇒ Context f (Context f a) → Context f a
appCxt (Hole x) = x
appCxt (In t)   = In (fmap appCxt t)
```

Context f is in fact the *free monad* of the functor f with *Hole* and *appCxt* as unit and multiplication operation, respectively. The functoriality of *Context f* is given as follows:

```
instance Functor f ⇒ Functor (Context f) where
  fmap f (Hole v) = Hole (f v)
  fmap f (In t)   = In (fmap (fmap f) t)
```

Recall that the set of terms $\mathcal{T}(\mathcal{F})$ is defined as the set $\mathcal{T}(\mathcal{F}, \emptyset)$ of terms without variables. We can do the same in the Haskell representation and replace our definition of the type *Term* with the following:

```
data Empty
type Term f = Context f Empty
```

Here, *Empty* is simply an empty type.[3] This definition of *Term* allows us to use terms and context in a uniform manner. For example, the function *appCxt* defined above can also be given the type *Context f* (*Term f*) → *Term f*. Moreover, this encoding allows us to give a more general type for the injection function:

```
inject :: (g ≼ f) ⇒ g (Context f a) → Context f a
```

The definition of *inject* remains the same. The same also applies to smart constructors; for example, the smart constructor *plus* has now the more general type

```
plus :: (Sig ≼ f) ⇒ Context f a → Context f a → Context f a
```

Most of the time we are using very simple contexts that only consist of a single functor application as constructed by the following function:

```
simpCxt :: Functor f ⇒ f a → Context f a
simpCxt t = In (fmap Hole t)
```

4.3 Bottom-Up Transduction Functions

The transduction rules of a DUTT use placeholder variables x_1, x_2, etc. in order to refer to arguments of function symbols. These placeholder variables can then be used on the right-hand side of a transduction rule. This mechanism makes it possible to rearrange, remove and duplicate the terms that are matched against these placeholder variables. On the other hand, it is not possible to inspect them. For instance, in Example 3, $\mathsf{not}(q_0(\mathsf{ff})) \rightarrow q_1(\mathsf{tt})$ would not be a valid transduction rule as we are not allowed to pattern match on the arguments of not. We can only observe the state.

[3] Note that in Haskell, every data type – including *Empty* – is inhabited by ⊥. Thus the definition of *Term* is not entirely accurate. However, for the sake of simplicity, we prefer this definition over a more precise one such as in [1].

When representing transduction rules as Haskell functions, we have to be careful in order to maintain this restriction on DUTTs. In their categorical representation, Hasuo et al. [11] recognised that the restriction due to placeholder variables in the transduction rules can be enforced by a *naturality* condition. Naturality, in turn, can be represented in Haskell's type system as *parametric polymorphism*. Following this approach, we represent DUTTs from signature functor f to signature functor g with state space q by the following type:

$$\textbf{type}\ \mathit{UpTrans}\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \to (q, \mathit{Context}\ g\ a)$$

In the definition of tree automata, states are used syntactically as a unary function symbol – an argument with state q is written as $q(x)$ in the left-hand side. In the Haskell representation, we use pairs and simply write (q, x).

In the type *UpTrans*, the type variable a represents the type of the placeholder variables. The universal quantification over a makes sure that placeholders can only be used if they appear on the left-hand side and that they cannot be inspected.

Example 4. We implement the DUTT from Example 3 in Haskell. At first we define the signature and the state space.

$$\begin{aligned}&\textbf{data}\ F\ a = \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B\\ &\textbf{data}\ Q\quad = \mathit{Q0} \mid \mathit{Q1} \mid \mathit{Q2}\end{aligned}$$

For the definition of the transduction function, we use the smart constructors *and*, *not*, *tt*, *ff* and b for the constructors of the signature F. These smart constructors are defined as before, e.g.

$$\begin{aligned}&\mathit{and} :: (F \preceq f) \Rightarrow \mathit{Context}\ f\ a \to \mathit{Context}\ f\ a \to \mathit{Context}\ f\ a\\ &\mathit{and}\ x\ y = \mathit{inject}\ (\mathit{And}\ x\ y)\end{aligned}$$

The definition of the transduction function is a one-to-one translation of the transduction rules of the DUTT from Example 3.

$$\begin{aligned}&\mathit{trans} :: \mathit{UpTrans}\ F\ Q\ F\\ &\mathit{trans}\ \mathit{TT} = (\mathit{Q1}, \mathit{tt}); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q0}, x)) = (\mathit{Q1}, \mathit{tt})\\ &\mathit{trans}\ \mathit{FF} = (\mathit{Q0}, \mathit{ff}); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q1}, x)) = (\mathit{Q0}, \mathit{ff})\\ &\mathit{trans}\ B\ \ = (\mathit{Q2}, b); \qquad \mathit{trans}\ (\mathit{Not}\ (\mathit{Q2}, x)) = (\mathit{Q2}, \mathit{not}\ (\mathit{Hole}\ x))\\ &\mathit{trans}\ (\mathit{And}\ (q, x)\ (p, y))\\ &\quad \mid q \equiv \mathit{Q0} \lor p \equiv \mathit{Q0} \qquad = (\mathit{Q0}, \mathit{ff})\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q1}, x)\ (\mathit{Q1}, y)) = (\mathit{Q1}, \mathit{tt})\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q1}, x)\ (\mathit{Q2}, y)) = (\mathit{Q2}, \mathit{Hole}\ y)\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q2}, x)\ (\mathit{Q1}, y)) = (\mathit{Q2}, \mathit{Hole}\ x)\\ &\mathit{trans}\ (\mathit{And}\ (\mathit{Q2}, x)\ (\mathit{Q2}, y)) = (\mathit{Q2}, \mathit{and}\ (\mathit{Hole}\ x)\ (\mathit{Hole}\ y))\end{aligned}$$

Since we do not constrain ourselves to finite state spaces, DUTTs do not add any expressive power to the state transition functions of DUTAs. Each DUTT

can be transformed into an algebra whose catamorphism is the transformation denoted by the DUTT:

$$
\begin{aligned}
&\mathit{runUpTrans} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{UpTrans}\ f\ q\ g \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Term}\ f \to (q, \mathit{Term}\ g) \\
&\mathit{runUpTrans}\ \mathit{trans} = \mathit{cata}\ (\mathit{appCxt}' \mathbin{.} \mathit{trans}) \\
&\quad \textbf{where}\ \mathit{appCxt}'\ (x, y) = (x, \mathit{appCxt}\ y)
\end{aligned}
$$

For instance, we run the DUTT from Example 4 as follows:

$$
\begin{aligned}
&\mathit{foldBool} :: \mathit{Term}\ F \to (Q, \mathit{Term}\ F) \\
&\mathit{foldBool} = \mathit{runUpTrans}\ \mathit{trans}
\end{aligned}
$$

As we have seen in Section 3.1, a tree acceptor with a compound state space comprises several computations which may be disentangled in order to increase modularity. A tree transducer intrinsically combines two computations: the state transition and the actual transformation of the term. We will see in Section 4.5 how to disentangle these two components. Before that, we shall look at a special case of DUTTs.

4.4 Tree Homomorphisms

To simplify matters, Bahr and Hvitved [1] focused on tree transducers with a singleton state space, also known as *tree homomorphisms* [3]:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Hom}\ f\ g = \forall\ a \mathbin{.} f\ a \to \mathit{Context}\ g\ a \\
&\mathit{runHom} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Hom}\ f\ g \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{runHom}\ \mathit{hom} = \mathit{cata}\ (\mathit{appCxt} \mathbin{.} \mathit{hom})
\end{aligned}
$$

Tree homomorphisms can only transform the tree structure uniformly without the ability to maintain a state. Nonetheless, tree homomorphisms provide a useful recursion scheme. For example, desugaring, i.e. transforming syntactic sugar of a language to the language's core operations, can in many cases be implemented as a tree homomorphism. Reconsider the signature $\mathit{Sig}' = \mathit{Inc} \oplus \mathit{Sig}$ that extends *Sig* with an increment operator. The increment operator is only syntactic sugar for adding the value 1. The corresponding desugaring transformation can be implemented as a tree homomorphism:

$$
\begin{aligned}
&\textbf{class}\ \mathit{DesugHom}\ f\ g\ \textbf{where} \\
&\quad \mathit{desugHom} :: \mathit{Hom}\ f\ g \\
&\quad \text{-- instance declaration lifting DesugHom to coproducts omitted} \\
&\mathit{desugar} :: (\mathit{Functor}\ f, \mathit{Functor}\ g, \mathit{DesugHom}\ f\ g) \Rightarrow \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{desugar} = \mathit{runHom}\ \mathit{desugHom} \\
&\textbf{instance}\ (\mathit{Sig} \preceq g) \Rightarrow \mathit{DesugHom}\ \mathit{Inc}\ g\ \textbf{where} \\
&\quad \mathit{desugHom}\ (\mathit{Inc}\ x) = \mathit{Hole}\ x\ \text{`}\mathit{plus}\text{`}\ \mathit{val}\ 1 \\
&\textbf{instance}\ (\mathit{Functor}\ g, f \preceq g) \Rightarrow \mathit{DesugHom}\ f\ g\ \textbf{where} \\
&\quad \mathit{desugHom} = \mathit{simpCxt} \mathbin{.} \mathit{inj}
\end{aligned}
$$

The first instance declaration states that as long as the target signature g contains *Sig*, we can desugar the signature *Inc* to g by mapping $inc(x)$ to $x + 1$. Using overlapping instances, the second instance declaration then defines the desugaring for all other signatures f – provided f is contained in the target signature – by leaving the input untouched.

The above instance declarations make it now possible to use the *desugar* function with type $Term\ Sig' \to Term\ Sig$. That is, *desugar* transforms a term over signature Sig' to a term over signature *Sig*.

As an ordinary recursive Haskell function we would implement desugaring as follows:

```
data Exp  = Val  Int | Plus  Exp  Exp
data Exp' = Val' Int | Plus' Exp' Exp' | Inc' Exp'

desugExp :: Exp' → Exp
desugExp (Val' i)     = Val i
desugExp (Plus' e f) = desugExp e `Plus` desugExp f
desugExp (Inc' e)     = desugExp e `Plus` Val 1
```

Note that we have to provide two separate data types for the input and output types of the function instead of using the compositionality of signatures. Moreover, the function *desugar* is applicable more broadly. It can be used as a function of type $Term\ (f \oplus Inc) \to Term\ f$ for any signature f that contains *Sig*, i.e. for which we have $Sig \preceq f$. Apart from these advantages in modularity and extensibility we also obtain all the advantages of using a transducer, which we shall discuss in more detail in Section 7.

4.5 Combining Tree Homomorphisms with State Transitions

We aim to combine the simplicity of tree homomorphisms and the expressivity of bottom-up tree transducers. To this end, we shall devise a method to combine a tree homomorphism and a state transition function to form a DUTT. This construction will be complete in the sense that any DUTT can be constructed in this way.

At first, compare the types of automata that we have considered so far:

```
type Hom      f   g = ∀ a . f      a  →      Context g a
type UpState  f q   =        f   q     →  q
type UpTrans f q g = ∀ a . f (q, a) → (q, Context g a)
```

We can observe from this – admittedly suggestive – comparison that a bottom-up tree transducer is roughly a combination of a tree homomorphism and a state transition function. Our aim is to make use of this observation by decomposing the specification of a bottom-up tree transducer into a tree homomorphism and a bottom-up state transition function. Like for the product construction of state transition functions from Section 3.1, we have to provide a mechanism to deal with dependencies between the two components. Since the state transition is

independent from the tree transformation, we only need to allow the tree homomorphism to access the state information that is produced by the bottom-up state transition.

A *stateful tree homomorphism* can thus be (tentatively) defined as follows:

$$\textbf{type}\ QHom\ f\ q\ g = \forall\ a\ .\ f\ (q, a) \rightarrow Context\ g\ a$$

Since q appears to the left of the function arrow but not to the right, functions of the above type have access to the states of the arguments, but do not transform the state themselves. However, we want to make it easy to ignore the state if it is not needed as the state is often only needed for a small number of cases. This goal can be achieved by replacing the pairing with the state space q by an additional argument of type $a \rightarrow q$.

$$\textbf{type}\ QHom\ f\ q\ g = \forall\ a\ .\ (a \rightarrow q) \rightarrow f\ a \rightarrow Context\ g\ a$$

We can still push this interface even more to the original tree homomorphism type *Hom* by turning the function argument into an implicit parameter [18]:

$$\textbf{type}\ QHom\ f\ q\ g = \forall\ a\ .\ (?state :: a \rightarrow q) \Rightarrow f\ a \rightarrow Context\ g\ a$$

In a last refinement step, we add an implicit parameter that provides access to the state of the current node as well:

$$\textbf{type}\ QHom\ f\ q\ g = \forall\ a\ .\ (?above :: q, ?below :: a \rightarrow q) \Rightarrow f\ a \rightarrow Context\ g\ a$$

Functions with implicit parameters have to be invoked in the scope of appropriate bindings. For functions of the above type this means that $?below$ has to be bound to a function of type $a \rightarrow q$ and $?above$ to a value of type q. We shall use the following function to make implicit parameters explicit:

$$explicit :: ((?above :: q, ?below :: a \rightarrow q) \Rightarrow b) \rightarrow q \rightarrow (a \rightarrow q) \rightarrow b$$
$$explicit\ x\ ab\ be = x\ \textbf{where}\ ?\ above = ab; ?below = be$$

In particular, given a stateful tree homomorphism h of type $QHom\ f\ q\ g$, we thus obtain a function $explicit\ h$ of type $q \rightarrow (a \rightarrow q) \rightarrow f\ a \rightarrow Context\ g\ a$.

The use of implicit parameters is solely for reasons of syntactic appearance and convenience. One can think of implicit parameters as reader monads without the syntactic overhead of monads. If, in the definition of a stateful tree homomorphism, the state is not needed, it can be easily ignored. Hence, tree homomorphisms are, in fact, also syntactic special cases of stateful tree homomorphisms.

The following construction combines a stateful tree homomorphism of type $QHom\ f\ q\ g$ and a state transition function of type $UpState\ f\ q$ into a tree transducer of type $UpTrans\ f\ q\ g$, which can then be used to perform the desired transformation:

$$upTrans :: (Functor\ f, Functor\ g) \Rightarrow$$
$$UpState\ f\ q \rightarrow QHom\ f\ q\ g \rightarrow UpTrans\ f\ q\ g$$

```
upTrans st hom t = (q, c) where
  q = st (fmap fst t)
  c = fmap snd (explicit hom q fst t)
runUpHom :: (Functor f, Functor g) ⇒
              UpState f q → QHom f q g → Term f → (q, Term g)
runUpHom st hom = runUpTrans (upTrans st hom)
```

Often the state space accessed by a stateful tree homomorphism is compound. Therefore, it is convenient to have the projection function *pr* built into the interface to the state space:

```
above :: (?above :: q, p ∈ q) ⇒ p
above = pr ?above
below :: (?below :: a → q, p ∈ q) ⇒ a → p
below = pr . ?below
```

In order to illustrate how stateful tree homomorphisms are programmed, we extend the signature *Sig* with variables and let bindings:

```
type Name = String
data Let e  = LetIn Name e e | Var Name
type LetSig = Let ⊕ Sig
```

We shall implement a simple optimisation that removes let bindings whenever the variable that is bound is not used in the scope of the let binding. To this end, we define a state transition that computes the set of free variables:

```
type Vars = Set Name
class FreeVarsSt f where
  freeVarsSt :: UpState f Vars
instance FreeVarsSt Sig where
  freeVarsSt (Plus x y) = x `union` y
  freeVarsSt (Val _)    = empty
instance FreeVarsSt Let where
  freeVarsSt (Var v)       = singleton v
  freeVarsSt (LetIn v e s) = if v `member` s then delete v (e `union` s)
                                             else s
```

Note that the free variables occurring in the right-hand side of a binding are only included if the bound variable occurs in the scope of the let binding. The transformation itself is simple:

```
class RemLetHom f q g where
  remLetHom :: QHom f q g
instance (Vars ∈ q, Let ⪯ g, Functor g) ⇒ RemLetHom Let q g where
  remLetHom (LetIn v _ s) | ¬ (v `member` below s) = Hole s
```

```
    remLetHom t                                    = simpCxt (inj t)
  instance (Functor f, Functor g, f ⪯ g) ⇒ RemLetHom f q g where
    remLetHom = simpCxt . inj
```

The homomorphism removes a let binding whenever the bound variable is not found in the set of free variables. Otherwise, no transformation is performed. Notice that the type specifies that the transformation depends on a state space that at least contains a set of variables. In addition, we make use of overlapping instances to define the transformation for all signatures different from *Let*. We then obtain the desired transformation function by combining the stateful tree homomorphism with the state transition computing the free variables:

```
  remLet :: (Functor f, FreeVarsSt f, RemLetHom f Vars f)
         ⇒ Term f → Term f
  remLet = snd . runUpHom freeVarsSt remLetHom
```

In particular, we can give *remLet* the type *Term LetSig* → *Term LetSig* but also *Term* (*Inc* ⊕ *LetSig*) → *Term* (*Inc* ⊕ *LetSig*).

4.6 Refining Dependent Bottom-Up State Transition Functions

The implicit parameters ?*below* and ?*above* of stateful tree homomorphisms provide an interface to the states of the children of the current node as well as the state of the current node itself. The same interface can be given to dependent bottom-up state transition functions as well. We therefore redefine the type of these state transitions from Section 3.1 as follows:

```
  type DUpState f p q = ∀ a . (?below :: a → p, ?above :: p, q ∈ p) ⇒ f a → q
```

While the definition of the product operator ⊗ remains the same, we have to change the other functions slightly to accommodate this change:

```
  dUpState :: Functor f ⇒ UpState f q → DUpState f p q
  dUpState st = st . fmap below
  upState :: DUpState f q q → UpState f q
  upState st s = res where
    res = explicit st res id s
  runDUpState :: Functor f ⇒ DUpState f q q → Term f → q
  runDUpState = runUpState . upState
```

Note that definition of *res* in *upState* is cyclic and thus crucially depends on Haskell's non-strict semantics. This also means that dependent state transition functions do not necessarily yield a terminating run since one can create a cyclic dependency by defining a state transition that depends on its own result such as the following:

```
  loopSt :: DUpState f p q
  loopSt _ = above
```

The definition of the code generator from Section 3.1 is easily adjusted to the slightly altered interface of dependent state transitions. Since we intend to extend the code generator in Section 6, we also turn it into a type class:

```
class CodeSt f q where
  codeSt :: DUpState f q Code

code :: (Functor f, CodeSt f (Code, Int), HeightSt f)
     ⇒ Term f → (Code, Addr)
code = runDUpState (codeSt ⊗ dUpState heightSt)

instance (Int ∈ q) ⇒ CodeSt Sig q where
  codeSt (Val i)      = [Acc i]
  codeSt (Plus x y) = below x ++ [Store a] ++ below y ++ [Add a]
    where a = below y
```

Note that the access to the state of the current node – via *above* – solves one of the minor issues we have identified at the end of Section 3.1. In order to obtain the state of the current node, we do not have to duplicate the corresponding state transition anymore. Moreover, we can use the same interface when we move to top-down state transitions in the next section.

5 Top-Down Automata

Operations on abstract syntax trees are often dependent on a state that is propagated top-down rather than bottom-up, e.g. typing environments and variable bindings. For such operations, recursion schemes derived from bottom-up automata are not sufficient. Hence, we shall consider top-down automata as a complementary paradigm to overcome this restriction.

Unlike the bottom-up case, we will not start with acceptors but with transducers. Our interest for bottom-up acceptors was based on the fact that such automata produce an output state. For top-down acceptors this application vanishes since such automata rather consume an input state than produce an output state. We will however come back to top-down state transition in order to make the state transition of top-down transducer modular – using the same stateful tree homomorphisms that we introduced in Section 4.5.

5.1 Deterministic Top-Down Tree Transducers

Deterministic top-down tree transducers (DDTTs) are able to produce transformations that depend on a top-down flow of information. They work in a fashion similar to bottom-up tree transducers but propagate their state downwards rather than upwards. More formally, a DDTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a set of states Q, an initial state $q_0 \in Q$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n)) \to u \quad \text{with } f \in \mathcal{F} \text{ and } q \in Q$$

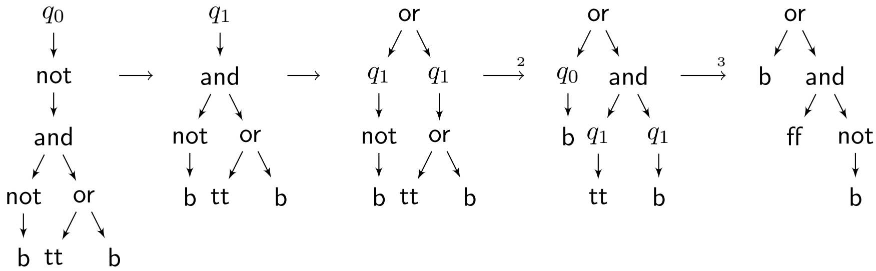
where $u \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ is a term over $\mathcal{G}$ and $Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$. That is, the right-hand side is a term that may have subterms of the form $p(x_i)$ with x_i a variable from the left-hand side and p a state in Q. In other words, each *occurrence* of a variable on the right-hand side is given a successor state.

In order to run a DDTT on a term $t \in \mathcal{T}(\mathcal{F})$, we have to provide an initial state q_0 and then apply the transduction rules to $q_0(t)$ in a top-down fashion. Eventually, this yields a result term $t' \in \mathcal{T}(\mathcal{G})$.

Example 5. Consider the signature $\mathcal{F} = \{\mathsf{or}/2, \mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0, \mathsf{b}/0\}$ and the DDTT from $\mathcal{F}$ to $\mathcal{F}$ with the set of states $Q = \{q_0, q_1\}$, initial state q_0 and the following transduction rules:

$$q_0(\mathsf{b}) \to \mathsf{b} \qquad q_0(\mathsf{tt}) \to \mathsf{tt} \quad q_0(\mathsf{ff}) \to \mathsf{ff} \qquad q_0(\mathsf{not}(x)) \to q_1(x)$$
$$q_1(\mathsf{b}) \to \mathsf{not}(\mathsf{b}) \quad q_1(\mathsf{tt}) \to \mathsf{ff} \quad q_1(\mathsf{ff}) \to \mathsf{tt} \qquad q_1(\mathsf{not}(x)) \to q_0(x)$$
$$q_0(\mathsf{and}(x, y)) \to \mathsf{and}(q_0(x), q_0(y)) \quad q_0(\mathsf{or}(x, y)) \to \mathsf{or}(q_0(x), q_0(y))$$
$$q_1(\mathsf{and}(x, y)) \to \mathsf{or}(q_1(x), q_1(y)) \qquad q_1(\mathsf{or}(x, y)) \to \mathsf{and}(q_1(x), q_1(y))$$

Terms over $\mathcal{F}$ are Boolean expressions with a single Boolean variable b. The above DDTT transforms such an expression into negation normal form by moving the operator not inwards. For instance, applied to the Boolean expression $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{or}(\mathsf{tt}, \mathsf{b})))$, the automaton yields the following derivation:



In order to start the run of a DDTT, the initial state q_0 has to be explicitly inserted at the root of the input term. The run of the automaton is completed as soon as all states in the term have vanished; there is no final state.

5.2 Top-Down Transduction Functions

Similar to bottom-up tree transducers, we follow the *placeholders-via-naturality* principle of Hasuo et al. [11] in order to represent top-down transduction functions:

$$\textbf{type}\ \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q, f\ \ a) \to \mathit{Context}\ g\ (q, a)$$

Now the state comes from above and is propagated downwards to the holes of the context, which defines the actual transformation that the transducer performs.

Running a top-down tree transducer on a term is a straightforward affair:

```
runDownTrans :: (Functor f, Functor g) ⇒ DownTrans f q g → q
                                        → Term f → Term g
runDownTrans tr q t = run (q, t) where
  run (q, In t) = appCxt (fmap run (tr (q, t)))
```

A top-down transducer is run by applying its transduction function – *tr* (q, t) – then recursively running the transformation in the holes of the produced context – *fmap run* – and finally joining the context with the thus produced embedded terms – *appCxt*.

Example 6. We implement the DDTT from Example 5 in Haskell as follows:

```
data F a = Or a a | And a a | Not a | TT | FF | B
data Q = Q0 | Q1

trans :: DownTrans F Q F
trans (Q0, TT)  = tt;          trans (Q0, B) = b
trans (Q1, TT)  = ff;          trans (Q1, B) = not b
trans (Q0, FF)  = ff;          trans (Q0, Not x) = Hole (Q1, x)
trans (Q1, FF)  = tt;          trans (Q1, Not x) = Hole (Q0, x)
trans (Q0, And x y) = Hole (Q0, x) `and` Hole (Q0, y)
trans (Q1, And x y) = Hole (Q1, x) `or`  Hole (Q1, y)
trans (Q0, Or  x y) = Hole (Q0, x) `or`  Hole (Q0, y)
trans (Q1, Or  x y) = Hole (Q1, x) `and` Hole (Q1, y)
```

The definition of the transduction function *trans* is a one-to-one translation of the transduction rules of the DDTT from Example 5. Note, that we use the smart constructors *or*, *and*, *not*, *tt*, *ff* and *b* on the right-hand side of the definitions. We apply the thus defined DDTT to a term of type *Term F* as follows:

```
negNorm :: Term F → Term F
negNorm = runDownTrans trans Q0
```

5.3 Top-Down State Transition Functions

Unfortunately, we cannot provide a full decomposition of DDTTs into a state transition and a homomorphism part in the way we did for DUTTs in Section 4.5. Unlike in DUTTs, the state transition in a DDTT is inherently dependent on the transformation: since a placeholder variable may be copied on the right-hand side, each copy may be given a different successor state! For example, a DDTT may have a transduction rule

$$q_0(f(x)) \to g(q_1(x), q_2(x))$$

which transforms a function symbol f into g and copies the argument of f. However, the two copies are given different successor states, viz. q_1 and q_2.

In order to avoid this dependency of state transitions on the transformation, we restrict ourselves to DDTTs in which successor states are given to placeholder variables and not their occurrences. That is, for each two occurrences of subterms $q_1(x)$ and $q_2(x)$ on the right-hand side of a transduction rule, we require that $q_1 = q_2$. The DDTT given in Example 5 is, in fact, of this form.

The top-down state transitions we are aiming for are dual to bottom-up state transitions. The run of a bottom-up state transition function assigns a state to each node by an upwards state propagation, performing the same computation as an upwards accumulation [8]. The run of a top-down state transition function, on the other hand, should do the same by a downwards state propagation and thus perform the same computation as a downwards accumulation [9, 10].

However, representing top-down state transitions is known to be challenging [8, 9, 10]. A first attempt yields the type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$. This type, however, allows apart from the state transition also a transformation. The result is not required to have the same shape as the input. For example, the following equation (partially) defines a function *bad* of type $\forall\ a\ .\ (Q, \mathit{Sig}\ \ a) \to \mathit{Sig}\ \ Q$:

$$\mathit{bad}\ (q, \mathit{Plus}\ \ x\ \ y) = \mathit{Val}\ 1$$

In order to assign a successor state to each child of the input node without permitting changes to its structure, we use explicit placeholders to which we can assign the successor states:

$$\textbf{type}\ \mathit{DownState}\ f\ \ q = \forall\ i\ .\ \mathit{Ord}\ \ i \Rightarrow (q, f\ \ i) \to \mathit{Map}\ \ i\ \ q$$

The type $\mathit{Map}\ \ i\ \ q$ represents finite mappings from type i to type q. Since such finite mappings are implemented by search trees, we require that the domain type i is of class *Ord*, which provides a total ordering.

The idea is to produce, from a state transition function of the above type, a function of type $\forall\ a\,.\,(q, f\ \ a) \to f\ \ q$ that does preserve the structure of the input and only produces the successor states. This is achieved by injecting unique placeholders of type i into a value of type $f\ \ a$ – one for each child node. We can then produce the desired value of type $f\ \ q$ from the mapping of type $\mathit{Map}\ \ i\ \ q$ given by the state transition function. A placeholder that is not mapped to a state explicitly is assumed to keep the state of the current node by default.

To work with finite mappings, we assume an interface with $\emptyset$ denoting the empty mapping, $x \mapsto y$ the singleton mapping that maps x to y, $m \cup n$ the left-biased union of two mappings m and n, and a lookup function $\mathit{lookup} :: \mathit{Ord}\ \ i \Rightarrow i \to \mathit{Map}\ \ i\ \ q \to \mathit{Maybe}\ \ q$. Moreover, we define the lookup with default as follows:

$$\begin{aligned}
&\mathit{findWithDefault} :: \mathit{Ord}\ \ i \Rightarrow q \to i \to \mathit{Map}\ \ i\ \ q \to q\\
&\mathit{findWithDefault}\ \ \mathit{def}\ \ i\ \ m = \textbf{case}\ \mathit{lookup}\ \ i\ \ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{def}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ q \quad\ \to q
\end{aligned}$$

At first, we need a mechanism to introduce unique placeholders into the structure of a functorial value. To this end, we will use the standard Haskell type class *Traversable* that provides the method

$$mapM :: (Traversable\ f, Monad\ m) \Rightarrow (a \to m\ b) \to f\ a \to m\ (f\ b)$$

which allows us to apply a monadic function to the components of a functorial value and then sequence the resulting monadic effects. Every polynomial functor can be made an instance of *Traversable*. Declarations to that effect can be derived automatically.

Ultimately, we want to number the elements in a functorial value to make them unique placeholders. To this end, we introduce a type of numbered values.

```
newtype Numbered a = Numbered (Int, a)
unNumbered :: Numbered a → a
unNumbered (Numbered (_, x)) = x
instance Eq (Numbered a) where
  Numbered (i, _) ≡ Numbered (j, _) = i ≡ j
instance Ord (Numbered a) where
  compare (Numbered (i, _)) (Numbered (j, _)) = compare i j
```

The instance declarations allow us to use elements of the type *Numbered a* as placeholders.

With the help of the *mapM* combinator, we define a function that numbers the components in a functorial value by counting up using a state monad:

```
number :: Traversable f ⇒ f a → f (Numbered a)
number x = fst (runState (mapM run x) 0) where
  run b = do n ← get
             put (n + 1)
             return (Numbered (n, b))
```

where $runState :: State\ s\ a \to s \to (a, s)$ runs a state monad with state type s, $put :: s \to State\ s\ m\ ()$ sets the state and $get :: State\ s\ s$ queries the state inside a state monad.

Using the above numbering combinator to create unique placeholders, we construct the explicit top-down propagation of states from a mapping of placeholders to successor states. Since the mapping of placeholders to successor states is partial, we also have to give a default state:

```
appMap :: Traversable f ⇒ (∀ i . Ord i ⇒ f i → Map i q)
                              → q → f a → f (q, a)
appMap qmap q s = fmap qfun s′ where
  s′      = number s
  qfun k = (findWithDefault q k (qmap s′), unNumbered k)
```

Finally, we can combine a top-down state transition function with a stateful tree homomorphism by propagating the successor states using the *appMap* combinator. As the default state, we take the state of the current node, i.e. by default the state remains unchanged.

$$
\begin{aligned}
&\mathit{downTrans} :: \mathit{Traversable}\ f \Rightarrow \mathit{DownState}\ f\ q \to \mathit{QHom}\ f\ q\ g \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{DownTrans}\ f\ q\ g \\
&\mathit{downTrans}\ \mathit{st}\ f\ (q, s) = \mathit{explicit}\ f\ q\ \mathit{fst}\ (\mathit{appMap}\ (\mathit{curry}\ \mathit{st}\ q)\ q\ s) \\
&\mathit{runDownHom} :: (\mathit{Traversable}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{DownState}\ f\ q \\
&\qquad\qquad\qquad \to \mathit{QHom}\ f\ q\ g \to q \to \mathit{Term}\ f \to \mathit{Term}\ g \\
&\mathit{runDownHom}\ \mathit{st}\ h = \mathit{runDownTrans}\ (\mathit{downTrans}\ \mathit{st}\ h)
\end{aligned}
$$

Note that we use the same type of stateful tree homomorphisms that we introduced for bottom-up state transitions. The roles of ?*above* and ?*below* are simply swapped: ?*above* refers to the state propagated from above whereas ?*below* provides the successor states of the current subterm. Stateful tree homomorphisms are ignorant of the direction in which the state is propagated.

Example 7. We reconstruct the DDTT from Example 6 by defining the state transition and the transformation separately:

$$
\begin{aligned}
&\mathit{state} :: \mathit{DownState}\ F\ Q \\
&\mathit{state}\ (\mathit{Q0}, \mathit{Not}\ x) = x \mapsto \mathit{Q1} \\
&\mathit{state}\ (\mathit{Q1}, \mathit{Not}\ x) = x \mapsto \mathit{Q0} \\
&\mathit{state}\ _ \qquad\qquad\quad = \emptyset \\
&\mathit{hom} :: \mathit{QHom}\ F\ Q\ F \\
&\mathit{hom}\ \mathit{TT} \qquad\quad = \mathbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \mathbf{then}\ \mathit{tt}\ \mathbf{else}\ \mathit{ff} \\
&\mathit{hom}\ \mathit{FF} \qquad\quad = \mathbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \mathbf{then}\ \mathit{ff}\ \mathbf{else}\ \mathit{tt} \\
&\mathit{hom}\ B \qquad\quad\ = \mathbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \mathbf{then}\ b\ \ \mathbf{else}\ \mathit{not}\ b \\
&\mathit{hom}\ (\mathit{Not}\ x) \quad = \mathit{Hole}\ x \\
&\mathit{hom}\ (\mathit{And}\ x\ y) = \mathbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \mathbf{then}\ \mathit{Hole}\ x\ \text{`}\mathit{and}\text{`}\ \mathit{Hole}\ y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathit{Hole}\ x\ \text{`}\mathit{or}\text{`}\ \ \mathit{Hole}\ y \\
&\mathit{hom}\ (\mathit{Or}\ x\ y) \ \ = \mathbf{if}\ \mathit{above} \equiv \mathit{Q0}\ \mathbf{then}\ \mathit{Hole}\ x\ \text{`}\mathit{or}\text{`}\ \ \mathit{Hole}\ y \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathit{Hole}\ x\ \text{`}\mathit{and}\text{`}\ \mathit{Hole}\ y
\end{aligned}
$$

Note that in the definition of the state transition function, we return the empty mapping for all constructors different from *Not*. Consequently, the input state for these constructors is propagated unchanged by default.

By combining the state transition function and the stateful homomorphism, we obtain the same transformation function as in Example 6.

$$
\begin{aligned}
&\mathit{negNorm}' :: \mathit{Term}\ F \to \mathit{Term}\ F \\
&\mathit{negNorm}' = \mathit{runDownHom}\ \mathit{state}\ \mathit{hom}\ \mathit{Q0}
\end{aligned}
$$

Instead of introducing explicit placeholders in order to distribute the successor state, we could have also simply taken the encoding we first suggested, i.e. via a function ρ of type $\forall\ a\ .\ (q, f\ \ a) \to f\ \ q$, and required as (an unchecked) side condition that ρ must preserve the shape of the input. This approach was taken in Gibbons' generic downwards accumulations [10] in which he requires the accumulation operation to be shape preserving.

Alternatively, we could have also adopted Gibbons' earlier approach to downwards accumulations [9], which instead represents the downward flow of information as a fold over a separately constructed data type called *path*. This path data type is constructed as the fixed point of a functor that is constructed from the signature functor. Unfortunately, this functor is quite intricate and not easy to program with in practice. Apart from that, it would be difficult to construct this path functor for each signature functor in Haskell.

In the end, our approach yields a straightforward representation of downward state transitions that is easy to work with in practise. Moreover, the ability to have a default behaviour for unspecified transitions makes for compact specifications as we have seen in Example 7. However, this default behaviour may also lead to errors more easily due to forgotten transitions.

5.4 Making Top-Down State Transition Functions Modular

Analogously to bottom-up state transition functions, we also define a variant of top-down state transition functions that has access to a bigger state space whose components are defined separately.

$$\begin{aligned}\mathbf{type}\ \mathit{DDownState}\ f\ p\ q = \forall\ i\ .\ (\mathit{Ord}\ i, ?\mathit{below} :: i \to p, ?\mathit{above} :: p, q \in p)\\ \Rightarrow f\ i \to \mathit{Map}\ i\ q\end{aligned}$$

Translations between ordinary top-down state transitions and their generalised variants are produced as follows:

$$\begin{aligned}&\mathit{dDownState} :: \mathit{DownState}\ f\ q \to \mathit{DDownState}\ f\ p\ q\\ &\mathit{dDownState}\ f\ t = f\ (\mathit{above}, t)\\ &\mathit{downState} :: \mathit{DDownState}\ f\ q\ q \to \mathit{DownState}\ f\ q\\ &\mathit{downState}\ f\ (q, s) = \mathit{res}\ \mathbf{where}\\ &\quad \mathit{res}\ \ \ = \mathit{explicit}\ f\ q\ \mathit{bel}\ s\\ &\quad \mathit{bel}\ k = \mathit{findWithDefault}\ q\ k\ \mathit{res}\end{aligned}$$

Similarly to their bottom-up counterparts, dependent top-down state transition functions that depend on the same state space can be combined to form a product state transition:

$$\begin{aligned}&(\circledast) :: (p \in c, q \in c) \Rightarrow \mathit{DDownState}\ f\ c\ p \to \mathit{DDownState}\ f\ c\ q\\ &\qquad\qquad\qquad\qquad \to \mathit{DDownState}\ f\ c\ (p, q)\\ &(\mathit{sp} \circledast \mathit{sq})\ t = \mathit{prodMap}\ \mathit{above}\ \mathit{above}\ (\mathit{sp}\ t)\ (\mathit{sq}\ t)\\ &\mathit{prodMap} :: \mathit{Ord}\ i \Rightarrow p \to q \to \mathit{Map}\ i\ p \to \mathit{Map}\ i\ q \to \mathit{Map}\ i\ (p, q)\end{aligned}$$

This construction is based on the pointwise product of mappings defined by *prodMap*, which we do not give in detail here. Since the mappings are partial, we have to provide a default state that is used in case only one of the mappings has a value for a given index. In accordance with the default behaviour of top-down state transition functions, this default state is the state from above.

As an example, we will define a transformation that replaces variables bound by let expressions with de Bruijn indices. For the sake of demonstration, we will implement this transformation using two states: the scope level, i.e. the number of let-bindings that are in scope, and a mapping from bound variables to the scope level of their respective binding site.

The scope level state simply counts the nesting of let bindings:

$$
\begin{array}{l}
\textbf{class}\ \mathit{ScopeLvlSt}\ f\ \textbf{where} \\
\quad \mathit{scopeLvlSt} :: \mathit{DownState}\ f\ \mathit{Int} \\
\textbf{instance}\ \mathit{ScopeLvlSt}\ \mathit{Let}\ \textbf{where} \\
\quad \mathit{scopeLvlSt}\ (d, \mathit{LetIn}\ _\ _\ b) = b \mapsto (d+1) \\
\quad \mathit{scopeLvlSt}\ _ \qquad\qquad\qquad = \emptyset \\
\textbf{instance}\ \mathit{ScopeLvlSt}\ f\ \textbf{where} \\
\quad \mathit{scopeLvlSt}\ _ \qquad\qquad\qquad = \emptyset
\end{array}
$$

Here we use the fact that if a successor state is not defined for a subexpression, then the current state is propagated by default.

The state that maintains a mapping from variables to the scope level of their respective binding site is dependent on the scope level state:

$$
\begin{array}{l}
\textbf{type}\ \mathit{VarLvl} = \mathit{Map}\ \mathit{Name}\ \mathit{Int} \\
\textbf{class}\ \mathit{VarLvlSt}\ f\ q\ \textbf{where} \\
\quad \mathit{varLvlSt} :: \mathit{DDownState}\ f\ q\ \mathit{VarLvl} \\
\textbf{instance}\ (\mathit{Int} \in q) \Rightarrow \mathit{VarLvlSt}\ \mathit{Let}\ q\ \textbf{where} \\
\quad \mathit{varLvlSt}\ (\mathit{LetIn}\ v\ _\ b) = b \mapsto ((v \mapsto \mathit{above}) \cup \mathit{above}) \\
\quad \mathit{varLvlSt}\ _ \qquad\qquad\quad = \emptyset \\
\textbf{instance}\ \mathit{VarLvlSt}\ f\ q\ \textbf{where} \\
\quad \mathit{varLvlSt}\ _ = \emptyset
\end{array}
$$

Note that the first occurrence of *above* is of type *Int* – derived from the type constraint $\mathit{Int} \in q$ – whereas the second occurrence is of type *VarLvl* – derived from the type constraint $\mathit{VarLvl} \in q$ in the type $\mathit{DDownState}\ f\ q\ \mathit{VarLvl}$.

Since we want to replace explicit variables with de Bruijn indices, we have to replace the signature *Let* with the following signature in the output term:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Let}'\ e\ \ = \mathit{LetIn}'\ e\ e \mid \mathit{Var}'\ \mathit{Int} \\
\textbf{type}\ \mathit{LetSig}' = \mathit{Let}' \oplus \mathit{Sig}
\end{array}
$$

The actual transformation is defined as a stateful tree homomorphism:

$$
\begin{array}{l}
\textbf{class}\ \mathit{DeBruijnHom}\ f\ q\ g\ \textbf{where} \\
\quad \mathit{deBruijnHom} :: \mathit{QHom}\ f\ q\ g \\
\textbf{instance}\ (\mathit{VarLvl} \in q, \mathit{Int} \in q, \mathit{Let}' \preceq g) \Rightarrow \mathit{DeBruijnHom}\ \mathit{Let}\ q\ g\ \textbf{where} \\
\quad \mathit{deBruijnHom}\ (\mathit{LetIn}\ _\ a\ b) = \mathit{letIn}'\ (\mathit{Hole}\ a)\ (\mathit{Hole}\ b) \\
\quad \mathit{deBruijnHom}\ (\mathit{Var}\ v) \qquad\ \ = \textbf{case}\ \mathit{lookup}\ v\ \mathit{above}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{error}\ \texttt{"free variable"}
\end{array}
$$

```
                                        Just i → var′ (above − i)
instance (Functor f, Functor g, f ⪯ g) ⇒ DeBruijnHom f q g where
  deBruijnHom = simpCxt . inj
```

Note that we issue an error if we encounter a variable that is not bound by a let expression. Otherwise, we create the de Bruijn index by subtracting the variable's scope level from the current scope level.

Finally, we have to tie the components together by forming the product state transition and providing an initial state:

```
deBruijn :: Term LetSig → Term LetSig′
deBruijn = runDownHom stateTrans deBruijnHom init
  where init = (∅, 0) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

Due to its open definition, we can give the function *deBruijn* also the type *Term* (*Inc* ⊕ *LetSig*) → *Term* (*Inc* ⊕ *LetSig*′), for example.

6 Bidirectional State Transitions

We have seen recursion schemes that use an upwards flow of information as well as recursion schemes that use a downwards flow of information. Some computations, however, require the combination of both. For example, if we want to extend the code generator from Section 4.6 to also work on let bindings, we need to propagate the generated code *upwards* but the symbol table for bound variables *downwards*.

In this section, we show two ways of achieving this combination.

6.1 Avoiding the Problem

The issue of combining two directions of information flow is usually circumvented by splitting up the computation in several runs instead. For the code generator, for instance, we can introduce a preprocessing step that translates let bindings into explicit assignments to memory addresses and variables into corresponding references to memory addresses.

This preprocessing step is easily implemented by modifying the stateful tree homomorphism from Section 5.4 that transforms variables into de Bruijn indices. Instead of de Bruijn indices we generate memory addresses.

At first, we define the signature that contains explicit addresses for bound variables:

```
data LetAddr e = LetAddr Addr e e | VarAddr Addr
type AddrSig   = LetAddr ⊕ Sig
```

The following stateful homomorphism then transforms a term over a signature containing *Let* into a signature containing *LetAddr* instead. The homomorphism depends on the same state as the de Bruijn homomorphism from Section 5.4:

```
class AddrHom f q g where
  addrHom :: QHom f q g

instance (VarLvl ∈ q, Int ∈ q, LetAddr ⪯ g) ⇒ AddrHom Let q g where
  addrHom (LetIn _ x y) = letAddr above (Hole x) (Hole y)
  addrHom (Var v)       = case lookup v above of
                            Nothing → error "free variable"
                            Just a → varAddr a

instance (Functor f, Functor g, f ⪯ g) ⇒ AddrHom f q g where
  addrHom = simpCxt . inj
```

By combining all components of the computation including the state transition functions *varLvlSt* and *scopeLvlSt* from Section 5.4, we obtain the desired transformation:

```
toAddr :: Addr → Term LetSig → Term AddrSig
toAddr startAddr = runDownHom stateTrans addrHom init
  where init = (∅, startAddr) :: (VarLvl, Int)
        stateTrans :: DownState LetSig (VarLvl, Int)
        stateTrans = downState (varLvlSt ⊛ dDownState scopeLvlSt)
```

The additional argument of type *Addr* allows us to control from which address we should start when assigning addresses to variables.

The actual code generation can then proceed on the signature *LetAddr* instead of *Let*:

```
instance CodeSt LetAddr q where
  codeSt (LetAddr a s e) = below s ++ [Store a] ++ below e
  codeSt (VarAddr a)     = [Load a]
```

To this end, we must also extend the *HeightSt* type class, which is used by the code generator:

```
instance HeightSt LetAddr where
  heightSt (LetAddr _ x y) = 1 + max x y
  heightSt (VarAddr _)     = 0
```

Now, we can use the function *code* from Section 4.6 with the type

```
code :: Term AddrSig → (Code, Addr)
```

Combining this function with the above defined transformation *toAddr*, yields the desired code generator:

```
codeLet :: Term LetSig → Code
codeLet t = c
  where t'        = toAddr (addr + 1) t
        (c, addr) = code t'
```

When combining the two functions *toAddr* and *code*, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. To this end, we use the result *addr* of the code generator function *code*, which is the highest address used for intermediate results, to initialise the address counter for the transformation *toAddr*. This makes sure that we use different addresses for intermediate results and bound variables.

6.2 A Direct Implementation

An alternative approach performs the bottom-up and the top-down computations side-by-side, taking advantage of the non-strict semantics of Haskell. This approach avoids the construction of an intermediate syntax tree that contains the required information.

For implementing a suitable recursion scheme, we make use of the fact that both bottom-up as well as top-down state transition functions in their dependent form share the same interface to access other components of the state space via the implicit parameters *?above* and *?below*.

The following combinator runs a bottom-up and a top-down state transition function that both depend on the product of the state spaces they define:

$$
\begin{aligned}
&\mathit{runDState} :: \mathit{Traversable}\ f \Rightarrow \mathit{DUpState}\ f\ (u,d)\ u\\
&\qquad\qquad\qquad\qquad\qquad \to \mathit{DDownState}\ f\ (u,d)\ d \to d \to \mathit{Term}\ f \to u\\
&\mathit{runDState}\ \mathit{up}\ \mathit{down}\ d\ (\mathit{In}\ t) = u\ \mathbf{where}\\
&\quad \mathit{bel}\ (\mathit{Numbered}\ (i,s)) =\\
&\qquad \mathbf{let}\ d' = \mathit{findWithDefault}\ d\ (\mathit{Numbered}\ (i,\bot))\ \mathit{qmap}\\
&\qquad \mathbf{in}\ \mathit{Numbered}\ (i, (\mathit{runDState}\ \mathit{up}\ \mathit{down}\ d'\ s, d'))\\
&\quad t' \quad\ \ = \mathit{fmap}\ \mathit{bel}\ (\mathit{number}\ t)\\
&\quad \mathit{qmap} = \mathit{explicit}\ \mathit{down}\ (u,d)\ \mathit{unNumbered}\ t'\\
&\quad u \quad\ \ = \mathit{explicit}\ \mathit{up}\ (u,d)\ \mathit{unNumbered}\ t'
\end{aligned}
$$

The definition of *runDState* looks convoluted but follows a simple structure: the two lines at the bottom apply both state transition functions at the current node. To this end, the state from above and the state from below is given as (u, d) and *unNumbered*, respectively. The latter works as t' is computed by first numbering the child nodes and then using the numbering to lookup the successor states from *qmap* as well as recursively applying *runDState* at the child nodes.

Note that the definition of *runDState* is cyclic in several different ways and thus essentially depends on Haskell's non-strict semantics: the result u of the bottom-up state transition function is used also as input for the bottom-up state transition function. Likewise the result *qmap* of the top-down state transition function is fed into the construction of t', which is given as argument to the top-down state transition function. Moreover, the definition of both u and *qmap* depend on each other.

The above combinator allows us to write a code generator for the signature *LetSig* without resorting to an intermediate syntax tree. However, we have to be

careful as this requires combining state transition functions with the same state space type: both *heightSt* and *scopeLvlSt* use the type *Int*.

However, the ambiguity can be easily resolved by "tagging" the types using *newtype* type synonyms. For the *scopeLvlSt* state transition, we define such a type like this:

```
newtype ScopeLvl = ScopeLvl {scopeLvl :: Int}
```

The tagging itself is a straightforward construction given the isomorphism between the type and its synonym in the form of a forward and a backward function:

```
tagDownState :: (q → p) → (p → q) → DownState f q → DownState f p
tagDownState i o t (q, s) = fmap i (t (o q, s))
```

We thus obtain a tagged variant of *scopeLvlSt*:

```
scopeLvlSt′ :: ScopeLvlSt f ⇒ DownState f ScopeLvl
scopeLvlSt′ = tagDownState ScopeLvl scopeLvl scopeLvlSt
```

The state maintained by *scopeLvlSt′* can now be accessed via the function *scopeLvl* in any compound state space containing *ScopeLvl*. A similar combinator can be defined for bottom-up state transitions.

Using the above state, we define a state transition function that assigns a memory address to each bound variable.

```
type VarAddr = Map Name Addr
class VarAddrSt f q where
  varAddrSt :: DDownState f q VarAddr
instance (ScopeLvl ∈ q) ⇒ VarAddrSt Let q where
  varAddrSt (LetIn v _ e) = e ↦ ((v ↦ scopeLvl above) ∪ above)
  varAddrSt _             = ∅
instance VarAddrSt f q where
  varAddrSt _ = ∅
```

Here, we use again overlapping instance declarations to give a uniform instance of *VarAddrSt* for all signatures different from *Let*.

We can now extend the type class *CodeSt* for the signature *Let*:

```
instance HeightSt Let where
  heightSt (LetIn _ x y) = 1 + max x y
  heightSt (Var _)       = 0
instance (ScopeLvl ∈ q, VarAddr ∈ q) ⇒ CodeSt Let q where
  codeSt (LetIn _ b e) = below b ++ [Store a] ++ below e
    where a = scopeLvl above
  codeSt (Var v)       = case lookup v above of
                           Nothing → error "unbound variable"
                           Just i  → [Load i]
```

Again, we have to be careful to avoid clashes in the use of addresses for storing intermediate results on the one hand and for storing results of let bindings on the other hand. Similar to our implementation in Section 6.1, we use the output of the bottom-up state transition to obtain the maximum address used for storing intermediate results.

Thus, we tie the different components of the computation together as follows:

$$
\begin{aligned}
&\mathit{codeLet}' :: \mathit{Term\ LetSig} \rightarrow \mathit{Code}\\
&\mathit{codeLet}'\ t = c\\
&\quad \mathbf{where}\ (c, \mathit{addr}) = \mathit{runDState}\ (\mathit{codeSt} \otimes \mathit{dUpState\ heightSt})\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{varAddrSt} \circledast \mathit{dDownState\ scopeLvlSt}')\\
&\qquad\qquad\qquad\qquad\qquad\quad (\emptyset :: \mathit{VarLvl}, \mathit{ScopeLvl}\ (\mathit{addr} + 1))\ t
\end{aligned}
$$

Note that in both implementations, we could have avoided the use of the result of the state transition function *heightSt* to initialise the address counter for bound variables. The modularity of our recursion schemes makes it possible to replace the *heightSt* state transition function with a different one. In this way, we could avoid clashes by using even address numbers for intermediate results and odd address numbers for variables.

We already observed that stateful tree homomorphisms cannot discern the direction in which the state is propagated. Thus we can supply them with a state using either bottom-up or top-down state transitions. In fact, following the bidirectional state transitions we considered above, we can provide a stateful tree homomorphism with a combined state given by both a bottom-up and a top-down state transition function. Such a transformation can for example be used to rename apart all bound variables or inline simple let bindings.

7 Discussion

We have seen that with some adjustments tree automata can be turned into highly modular recursion schemes. These recursion schemes allow us to take advantage of two orthogonal dimensions of modularity: modularity in the state that is propagated and – courtesy of Swierstra's [23] *data types à la carte* – modularity in the structure of terms. In addition to that, we also showed how to decompose transducers into a homomorphism and into a state transition part. This high level of modularity makes our automata-based recursion schemes especially valuable for constructing modular compilers as we have illustrated in our running example. However, we should point out that there are many more aspects to consider when constructing compilers in a modular fashion [4].

The dependent forms of bottom-up and top-down state transitions that we have developed in this paper are nothing else than the *synthesised* and *inherited attributes* known from *attribute grammars* [22]. In fact, the combinator *runDState* that runs both a bottom-up and a top-down state transition can be seen as a run of an attribute grammar with corresponding synthesised and inherited attributes. Viera et al. [24] have developed a Haskell library that allows to specify such attribute grammars in Haskell in a very concise way.

We also obtain an added value by using a powerful functional language for the embedding of our recursion schemes. One immediate benefit that we obtain is the use of further generic programming techniques. For example, the *heightSt* state transition function could have been defined entirely generically, without having to extend the definition for every new signature.

Why Tree Transducers?. In principle, tree transducers offer no increase in expressiveness over (dependent) bottom-up state transition functions since we allow for infinite state spaces anyway. However, due to their additional structure they provide at least two advantages.

First of all, tree transducers are very flexible in the way they can be manipulated in order to form new transformations. For example, we can extend a given signature functor f with annotations of some type a by using the construction

$$\textbf{data}\ (f :\&: a)\ e = f\ e :\&: a$$

A term over the signature (f :&: a) is similar to a term over f but it additionally contains annotations of type a at every subterm. We can provide a combinator that modifies a tree transducer from F to G into one from F :&: A to G :&: A that propagates the annotations from the input term to the output term [1].

Secondly, tree transducers can be composed. That is, given two bottom-up (respectively top-down) tree transducers – one from F to G, the other one from G to H, we can generically construct a bottom-up (respectively top-down) transducer from F to H whose transformation is equal to the composition of the transformations denoted by the original transducers [7]. The resulting transducer then only has to traverse the input term once and avoids the construction of the intermediate term [26]. Note that tree homomorphisms can be considered both a special case of bottom-up and of top-down tree transducers and can thus be composed with either kind.

The two abovementioned features also set tree transducers apart from other generic programming approaches such as *Scrap your Boilerplate* [13, 12, 17] or *Uniplate* [21]. We do not give the full technical details of the two features here but the implementation can be found in the `compdata` package [2].

Extensions & Future Work. While we only considered single recursive data types, this restriction is not essential: following the construction of Yakushev et al. [27] and Bahr and Hvitved [1], our recursion schemes can be readily extended to work on mutually recursive data types as well.

Note that the *runDState* combinator of Section 6.2 constructs the product of the two state spaces u and d. Consequently, if u is a compound state space, we obtain a product type that is not a right-associative nesting of pairs which we require for the type class $\in$ to work properly. However, this can be remedied by a more clever encoding of compound state spaces as *heterogeneous lists* [14] or generating instance declarations for products of a limited number of components via *Template Haskell*.

The transducers that we have considered here have one severe limitation. This limitation can be seen when looking at the implementation of these transducers in Haskell: the parametric polymorphism of the type for placeholder variables prevents us from using these placeholder variables in the state transition. This would allow us to store and retrieve subterms that the placeholder variables are instantiated with. The ability to do that is necessary in order to perform "non-local" transformations such as inlining of arbitrary let bindings or applying substitutions. However, we can remedy this issue by making the state a functor. The type of bottom-up respectively top-down transducers would then look as follows:

$$\begin{aligned}&\textbf{type } \mathit{UpTrans} \quad f\ q\ g = \forall\ a\ .\ f\ (q\ a, a) \rightarrow (q\ (\mathit{Context}\ g\ a), \mathit{Context}\ g\ a)\\ &\textbf{type } \mathit{DownTrans}\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \rightarrow \mathit{Context}\ g\ (q\ (\mathit{Context}\ g\ a), a)\end{aligned}$$

We can then, for example, instantiate q with *Map Var* such that the state is a substitution, i.e. a mapping from variables to terms (respectively term placeholders).

The above types represent a limited form of macro tree transducers [7]. Wile the decomposition of such an extended bottom-up transducer into a homomorphism and a state transition function is again straightforward, the decomposition of an extended top-down transducer is trickier: at least the representation with explicit placeholders that we used for dependent top-down state transition functions does not straightforwardly generalise to polymorphic states.

Note that the abovementioned limitation only affects transducers, not state transition functions. We can, of course, implement inlining and substitution as a bidirectional state transition. However, if we want to make use of the nice properties of transducers, we have to move to the extended tree transducers illustrated above.

Acknowledgements. The author would like to thank Tom Hvitved, Jeremy Gibbons, Wouter Swierstra, Doaitse Swierstra and the anonymous referees for valuable comments, corrections, suggestions for improvements and pointers to the literature.

References

[1] Bahr, P., Hvitved, T.: Compositional Data Types. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming, WGP 2011, pp. 83–94. ACM, New York (2011)

[2] Bahr, P., Hvitved, T.: Compdata (2012), `http://hackage.haskell.org/package/compdata`, module Data.Comp.Automata

[3] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://www.grappa.univ-lille3.fr/tata` (release October 12, 2007)

[4] Day, L., Hutton, G.: Towards Modular Compilers For Effects. In: Proceedings of the Symposium on Trends in Functional Programming, Madrid, Spain (2011)

[5] Eilenberg, S., Wright, J.B.: Automata in general algebras. Inform. Control. 11(4), 452–470 (1967)
[6] Fokkinga, M.M.: Law and Order in Algorithmics. Ph.D. thesis, University of Twente, 7500 AE Enschede, Netherlands (1992)
[7] Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. Springer-Verlag New York, Inc. (1998)
[8] Gibbons, J.: Upwards and Downwards Accumulations on Trees. In: Bird, R.S., Morgan, C.C., Woodcock, J.C.P. (eds.) MPC 1992. LNCS, vol. 669, pp. 122–138. Springer, Heidelberg (1993)
[9] Gibbons, J.: Polytypic Downwards Accumulations. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 207–233. Springer, Heidelberg (1998)
[10] Gibbons, J.: Generic downwards accumulations. Sci. Comput. Program. 37(1-3), 37–65 (2000)
[11] Hasuo, I., Jacobs, B., Yu, H.-J.: Categorical Views on Computations on Trees (Extended Abstract). In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 619–630. Springer, Heidelberg (2007)
[12] Hinze, R., Löh, A., Oliveira, B.C.d.S.: "Scrap Your Boilerplate" Reloaded. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006)
[13] Hinze, R., Löh, A.: "Scrap Your Boilerplate" Revolutions. In: Yu, H.-J. (ed.) MPC 2006. LNCS, vol. 4014, pp. 180–208. Springer, Heidelberg (2006)
[14] Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 96–107. ACM Press (2004)
[15] Knuth, D.E.: Semantics of context-free languages. Theory Comput. Syst. 2(2), 127–145 (1968)
[16] Kühnemann, A.: Benefits of Tree Transducers for Optimizing Functional Programs. In: Arvind, V., Sarukkai, S. (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 146–158. Springer, Heidelberg (1998)
[17] Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, pp. 204–215. ACM, New York (2005)
[18] Lewis, J.R., Launchbury, J., Meijer, E., Shields, M.B.: Implicit parameters: dynamic scoping with static types. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 108–118. ACM, New York (2000)
[19] Marlow, S.: Haskell 2010 Language Report (2010)
[20] Meijer, E., Fokkinga, M., Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)
[21] Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 49–60. ACM, New York (2007)
[22] Paakki, J.: Attribute grammar paradigmsa high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (1995)
[23] Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4), 423–436 (2008)
[24] Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, pp. 245–256. ACM, New York (2009)

[25] Voigtländer, J.: Formal Efficiency Analysis for Tree Transducer Composition. Theory Comput. Syst. 41(4), 619–689 (2007)
[26] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
[27] Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, pp. 233–244. ACM, New York (2009)

Constructing Applicative Functors

Ross Paterson

City University London, UK

Abstract. Applicative functors define an interface to computation that is more general, and correspondingly weaker, than that of monads. First used in parser libraries, they are now seeing a wide range of applications. This paper sets out to explore the space of non-monadic applicative functors useful in programming. We work with a generalization, lax monoidal functors, and consider several methods of constructing useful functors of this type, just as transformers are used to construct computational monads. For example, coends, familiar to functional programmers as existential types, yield a range of useful applicative functors, including left Kan extensions. Other constructions are final fixed points, a limited sum construction, and a generalization of the semi-direct product of monoids. Implementations in Haskell are included where possible.

1 Introduction

This paper is part of a tradition of applying elementary category theory to the design of program libraries. Moggi [16] showed that the notion of monad could be used to structure denotational descriptions of programming languages, an idea carried over to program libraries by Wadler [20]. It turns out that the monads useful in semantics and programming can be constructed from a small number of monad transformers also identified by Moggi [17].

Applicative functors [15] provide a more limited interface than monads, but in return have more instances. All monads give rise to applicative functors, but our aim is to explore the space of additional instances with applications to programming. We are particularly interested in general constructions, with which programmers can build their own applicative functors, knowing that they satisfy the required laws. It is already known that applicative functors, unlike monads, can be freely composed. We identify a number of further general constructions, namely final fixed points, a limited sum construction, a generalization of semi-direct products of monoids, and coends (including left Kan extensions). By combining these constructions, one can obtain most of the computational applicative functors in the literature, with proofs of their laws. General constructions also clarify the relationships between seemingly unrelated examples, and suggest further applications.

Elementary category theory provides an appropriately abstract setting for the level of generality we seek. An idealized functional language corresponds to a type of category with first-class functions (a cartesian closed category). Applicative functors on such a category are equivalent to a simpler form called lax monoidal

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 300–323, 2012.
© Springer-Verlag Berlin Heidelberg 2012

functors, which are more convenient to work with. We can build up lax monoidal functors in more general ways by ranging across several different categories, as long as the end result acts on the category of our functional language, and is thus applicative. Familiarity with the basic definitions of categories and functors is assumed. The other notions used are mostly shallow, and will be explained along the way.

In the next section, we introduce applicative and lax monoidal functors. The rest of the paper describes the general constructions, illustrated with examples in Haskell where possible. Two proof styles are used throughout the paper. When making statements that apply to any category, we use standard commuting diagrams. However many statements assume a cartesian closed category, or at least a category with products. For these we use the internal language of the category, which provides a term language with equational reasoning that will be familiar to functional programmers.

2 Applicative Functors

The categorical notion of "functor" is modelled in Haskell with the type class

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Instances include a variety of computational concepts, including containers, in which `fmap` modifies elements while preserving shape. Another class of instances are "notions of computation", including both monads and applicative functors, in which terms of type $F\,a$ correspond to computations producing values of type a, but also having an "effect" described by the functor F, e.g. modifying a state, possibly throwing an exception, or non-determinism. The requirement that F be a functor allows one to modify the value returned without changing the effect.

The applicative interface adds pure computations (having no effect) and an operation to sequence computations, combining their results. It is described by a type class:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

If we compare this with the type class of monads:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

we see that `pure` corresponds to `return`; the difference lies in the sequencing operations. The more powerful `>>=` operation available with monads allows the choice of the second computation to depend on the result of the first, while in the applicative case there can be no such dependency. Every monad can be made an applicative functor in a uniform way, here illustrated with the `Maybe` monad:

```
instance Functor Maybe where
    fmap f m = m >>= \ x -> return (f x)

instance Applicative Maybe where
    pure = return
    mf <*> mx = mf >>= \ f -> mx >>= \ x -> return (f x)
```

For functors that are also monads the monadic interface is often more convenient, but here we shall be more interested in applicative functors that are not also monads. A simple example is a constant functor returning a monoid [15]. Here is that functor expressed in Haskell using the `Monoid` class, which defines an associative binary operation `<>` with identity `mempty`:

```
newtype Constant a b = Constant a

instance Functor (Constant a) where
    fmap f (Constant x) = Constant x

instance Monoid a => Applicative (Constant a) where
    pure _ = Constant mempty
    Constant x <*> Constant y = Constant (x <> y)
```

The more limited applicative interface has many more instances, some of which will be presented in later sections. For example, the constrained form of sequencing offered by the applicative interface makes possible instances in which part of the value is independent of the results of computations, e.g. parsers that pre-generate parse tables [18]. Unlike monads, applicative functors are closed under composition.

However many applications of monads, such as traversal of containers, can be generalized to the applicative interface [15].

2.1 Lax Monoidal Functors

The applicative interface is convenient for programming, but in order to explore relationships between functors we shall use an alternative form with a more symmetrical sequencing operation:

```
class Functor f => Monoidal f where
    unit :: f ()
    mult :: f a -> f b -> f (a, b)
```

This interface, with identity and associativity laws, is equivalent to the applicative interface—the operations are interdefinable:

```
pure x = fmap (const x) unit
a <*> b = fmap (uncurry id) (mult a b)

unit = pure ()
mult a b = fmap (,) a <*> b
```

If we uncurry the operation `mult` of the `Monoidal` class, we obtain an operation $\circledast : F\,a \times F\,b \rightarrow F\,(a \times b)$. This suggests generalizing from products to other binary type constructors, a notion known in category theory as a monoidal category.

A *monoidal category* [13] consists of a category $\mathcal{C}$, a functor $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ and an object $\top$ of $\mathcal{C}$, with coherent natural isomorphisms

$$\begin{array}{ll}
\lambda : \top \otimes a \cong a & \text{(left identity)} \\
\rho : a \otimes \top \cong a & \text{(right identity)} \\
\alpha : a \otimes (b \otimes c) \cong (a \otimes b) \otimes c & \text{(associativity)}
\end{array}$$

A *symmetric monoidal category* also has

$$\sigma : a \otimes b \cong b \otimes a \qquad \text{(symmetry)}$$

Both products and coproducts are examples of monoidal structures, and both are also symmetric. Given a monoidal category $\langle \mathcal{C}, \top, \otimes, \lambda, \rho, \alpha \rangle$, the category $\mathcal{C}^{\mathrm{op}}$, obtained by reversing all the morphisms of $\mathcal{C}$, also has a monoidal structure: $\langle \mathcal{C}^{\mathrm{op}}, \top, \otimes, \lambda^{-1}, \rho^{-1}, \alpha^{-1} \rangle$. The product of two monoidal categories is also monoidal, combining the isomorphisms of the two categories in parallel.

Often we simply refer to the category when the monoidal structure is clear from the context.

Some functors preserve this structure exactly, with $\top' = F\,\top$ and $F\,a \otimes' F\,b = F\,(a \otimes b)$; a trivial example is the identity functor. Others, such as the product functor $\times : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ preserve it up to isomorphism:

$$1 \cong 1 \times 1$$

$$(a_1 \times a_2) \times (b_1 \times b_2) \cong (a_1 \times b_1) \times (a_2 \times b_2)$$

We obtain a larger and more useful class of functors by relaxing further, requiring only morphisms between the objects in each pair, thus generalizing the class `Monoidal` above from products to any monoidal category.

A *lax monoidal functor* between monoidal categories $\langle \mathcal{C}, \otimes, \top \rangle$ and $\langle \mathcal{C}', \otimes', \top' \rangle$ consists of a functor $F : \mathcal{C} \rightarrow \mathcal{C}'$ with natural transformations

$$\begin{array}{ll}
u : \top' \rightarrow F\,\top & \text{(unit)} \\
\circledast : F\,a \otimes' F\,b \rightarrow F\,(a \otimes b) & \text{(multiplication)}
\end{array}$$

such that the following diagrams commute:

$$\begin{array}{ccc}
\top \otimes F\,a & \xrightarrow{\lambda} & F\,a \\
\downarrow{\scriptstyle u \otimes F\,a} & & \uparrow{\scriptstyle F\,\lambda} \\
F\,\top \otimes F\,a & \xrightarrow[\circledast]{} & F\,(\top \otimes a)
\end{array}
\qquad
\begin{array}{ccc}
F\,a \otimes \top & \xrightarrow{\rho} & F\,a \\
\downarrow{\scriptstyle F\,a \otimes u} & & \uparrow{\scriptstyle F\,\rho} \\
F\,a \otimes F\,\top & \xrightarrow[\circledast]{} & F\,(a \otimes \top)
\end{array}$$

$$\begin{array}{ccccc}
F\,a \otimes (F\,b \otimes F\,c) & \xrightarrow{F\,a \otimes \circledast} & F\,a \otimes F\,(b \otimes c) & \xrightarrow{\circledast} & F\,(a \otimes (b \otimes c)) \\
\downarrow{\scriptstyle \alpha} & & & & \downarrow{\scriptstyle F\,\alpha} \\
(F\,a \otimes F\,b) \otimes F\,c & \xrightarrow[\circledast \otimes F\,c]{} & F\,(a \otimes b) \otimes F\,c & \xrightarrow[\circledast]{} & F\,((a \otimes b) \otimes c)
\end{array}$$

The first two diagrams state that u is the left and right identity respectively of the binary operation $\circledast$, while the last diagram expresses the associativity of $\circledast$.

2.2 Weak Commutativity

Although the definition of a lax monoidal functor neatly generalizes the `Monoidal` class, it lacks the counterpart of `pure`. We will also want an associated axiom stating that pure computations can be commuted with other computations. (There is a notion of symmetric lax monoidal functor, but requiring the ability to swap any two computations would exclude too many functors useful in computation, where the order in which effects occur is often significant.)

Thus we define an *applicative functor* on a symmetric monoidal category $\mathcal{C}$ as consisting of a lax monoidal functor $F : \mathcal{C} \to \mathcal{C}$, with a natural transformation $p : a \to F\,a$ (corresponding to the `pure` function of the `Applicative` class) satisfying $p_\top = u$ and $p \circ \circledast = \circledast \circ p \otimes p$, plus a weak commutativity condition:

$$\begin{array}{ccccc}
a \otimes F\,b & \xrightarrow{p \otimes F\,b} & F\,a \otimes F\,b & \xrightarrow{\circledast} & F\,(a \otimes b) \\
\downarrow{\scriptstyle \sigma} & & & & \downarrow{\scriptstyle F\,\sigma} \\
F\,b \otimes a & \xrightarrow[F\,b \otimes p]{} & F\,b \otimes F\,a & \xrightarrow[\circledast]{} & F\,(b \otimes a)
\end{array}$$

We could also express the weak commutativity condition as a constraint on functors with a tensorial strength, but here we shall avoid such technicalities by assuming that function spaces are first-class types, with primitives to perform application and currying, or in categorical terms that we are working in a cartesian closed category (ccc). In particular, if $\mathcal{A}$ is a ccc, any lax monoidal functor $F : \mathcal{A} \to \mathcal{A}$ is also applicative. To show this, we make use of another advantage of working in a ccc, namely that we can conduct proofs in the internal λ-calculus of the category [12], in which variables of type a stand for arrows of $\mathcal{A}(1, a)$, and we write $f\,(e_1, \ldots, e_n)$ for $f \circ \langle e_1, \ldots, e_n \rangle$. The result is a convenient language that is already familiar to functional programmers. When working in categories with products we shall calculate using the internal language; when products are not assumed we shall use diagrams.

In the internal language, we can define $p : I \dot{\to} F$ with the counterpart of the above definition of `pure` for any `Monoidal` functor:

$$p\,x = F\,(\mathit{const}\,x)\,u$$

The proof of weak commutativity is then a simple calculation in the internal language:

$$
\begin{array}{rll}
F\,\sigma\,(p\,x \circledast y) & = F\,\sigma\,(F\,(\mathit{const}\,x)\,u \circledast y) & \text{definition of } p \\
& = F\,(\sigma \circ (\mathit{const}\,x) \times \mathit{id})\,(u \circledast y) & \text{naturality of } \circledast \\
& = F\,(\sigma \circ (\mathit{const}\,x) \times \mathit{id})\,(F\,\lambda^{-1}\,y) & \text{left identity} \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \sigma)\,(F\,\lambda^{-1}\,y) & \text{naturality of } \sigma \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \sigma \circ \lambda^{-1})\,y & \text{functor} \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \rho^{-1})\,y & \text{symmetry} \\
& = F\,(\mathit{id} \times \mathit{const}\,x)\,(F\,\rho^{-1}\,y) & \text{functor} \\
& = F\,(\mathit{id} \times \mathit{const}\,x)\,(y \circledast u) & \text{right identity} \\
& = y \circledast F\,(\mathit{const}\,x)\,u & \text{naturality of } \circledast \\
& = y \circledast p\,x & \text{definition of } p
\end{array}
$$

It is also known that lax monoidal functors in a ccc are equivalent to closed functors [5], which resemble the `Applicative` interface, but again the lax monoidal form is more convenient for defining derived functors.

Thus our strategy will be to construct a lax monoidal functor over the product structure of a ccc, but we may construct it from constituents involving other monoidal categories. As a simple example, we have seen that the product functor $\times : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is lax monoidal, and we can compose it with the diagonal functor from $\mathcal{A}$ to $\mathcal{A} \times \mathcal{A}$ (also lax monoidal) to obtain a lax monoidal functor from $\mathcal{A}$ to $\mathcal{A}$:

$$F\,a = a \times a$$

though in this case the resulting functor is also monadic. In Section 5 we also use auxiliary categories with monoidal structures other than products.

3 Fixed Points, Limits and Colimits

A standard example of a computational monad is the list monad, which may be used to model backtracking. There is another lax monoidal functor on lists, with a unit constructing infinite lists and multiplication forming the zip of two lists [15]:

```
data ZipList a = Nil | Cons a (ZipList a)

instance Functor ZipList where
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Monoidal ZipList where
    unit = Cons () unit
    mult (Cons x xs) (Cons y ys) = Cons (x,y) (mult xs ys)
    mult _ _ = Nil
```

It turns out that this instance, and the proof that it satisfies the lax monoidal laws, follow from a general construction. We can observe that `ZipList` is a fixed point through the second argument of the binary functor $F(a, b) = 1 + a \times b$.

That is, F is the functor $\texttt{Maybe} \circ \times$, a composition of two lax monoidal functors and therefore lax monoidal.

There are two canonical notions of the fixed point of a functor, the initial and final fixed points, also known as data and codata. Initial fixed points can be used to define monads; here we use final fixed points to define lax monoidal functors. Recall that a parameterized final fixed point of a functor $F : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{B}$ consists of a functor $\nu F : \mathcal{A} \rightarrow \mathcal{B}$ with an isomorphism $c : F\,(a, \nu F\, a) \cong \nu F\, a$ and an unfold operator $[\![\cdot]\!]$ constructing the unique morphism satisfying

$$\begin{array}{ccc} b & \overset{[\![f]\!]}{\dashrightarrow} & \nu F\, a \\ {\scriptstyle f}\downarrow & & \uparrow{\scriptstyle c} \\ F\,(a, b) & \underset{F\,(a, [\![f]\!])}{\dashrightarrow} & F\,(a, \nu F\, a) \end{array}$$

If F is lax monoidal, we can define the unit and multiplication morphisms of a lax monoidal structure on νF as two of these unfolds:

$$\begin{array}{ccc} \top & \overset{u_{\nu F} = [\![u_F]\!]}{\dashrightarrow} & \nu F\, \top \\ {\scriptstyle u_F}\downarrow & & \uparrow{\scriptstyle c} \\ F\,(\top, \top) & \underset{F\,(\top, u_{\nu F})}{\dashrightarrow} & F\,(\top, \nu F) \end{array}$$

$$\begin{array}{ccc} \nu F\, a_1 \otimes \nu F\, a_2 & \overset{\circledast_{\nu F} = [\![\circledast_F \circ c^{-1} \otimes c^{-1}]\!]}{\dashrightarrow} & \nu F\,(a_1 \otimes a_2) \\ {\scriptstyle c^{-1} \otimes c^{-1}}\downarrow & & \\ F\,(a_1, \nu F\, a_1) \otimes F\,(a_2, \nu F\, a_2) & & \uparrow{\scriptstyle c} \\ {\scriptstyle \circledast_F}\downarrow & & \\ F\,(a_1 \otimes a_2, \nu F\, a_1 \otimes \nu F\, a_2) & \underset{F\,(a_1 \otimes a_2, \circledast_{\nu F})}{\dashrightarrow} & F\,(a_1 \otimes a_2, \nu F\,(a_1 \otimes a_2)) \end{array}$$

In particular, for $F = \texttt{Maybe} \circ \times$, this construction yields a lax monoidal functor equivalent to `ZipList` above.

One can prove using fusion that this definition does indeed satisfy the lax monoidal laws, but we shall prove a more general result instead.

3.1 Limits

Ignoring the parameter $\mathcal{A}$ for the moment, another way to define the final fixed point of a functor $F : \mathcal{B} \rightarrow \mathcal{B}$ starts with the terminal object 1. Using with the unique morphism $!_{F\,1} : F\,1 \rightarrow 1$, we can define a chain of objects and morphisms:

$$\cdots \longrightarrow F^3\,1 \xrightarrow[F^2\,!_{F\,1}]{} F^2\,1 \xrightarrow[F\,!_{F\,1}]{} F\,1 \xrightarrow[!_{F\,1}]{} 1$$

The final fixed point νF is defined as the limit of this chain, an object with a commuting family of morphisms (a *cone*) to the objects of the chain:

$$\begin{array}{l} \nu F \to F^3\,1,\ \nu F \to F^2\,1,\ \nu F \to F\,1,\ \nu F \to 1 \\ \cdots \longrightarrow F^3\,1 \xrightarrow[F^2\,!_{F\,1}]{} F^2\,1 \xrightarrow[F\,!_{F\,1}]{} F\,1 \xrightarrow[!_{F\,1}]{} 1 \end{array}$$

such that any other such cone, say from an object B, can be expressed as a composition of a unique morphism $B \to \nu F$ and the cone from νF.

This construction is sufficient for final fixed points of regular functors like `ZipList`, but for the general case we need to lift the whole thing to the category $\mathbf{Fun}(\mathcal{A}, \mathcal{B})$, whose objects are functors $\mathcal{A} \to \mathcal{B}$, and whose morphisms are natural transformations. Given a functor Φ on this category, we can repeat the above construction in the functor category, starting with the constant functor $\overline{1}$:

$$\begin{array}{l} \nu\Phi \to \Phi^3\,\overline{1},\ \nu\Phi \to \Phi^2\,\overline{1},\ \nu\Phi \to \Phi\,\overline{1},\ \nu\Phi \to \overline{1} \\ \cdots \longrightarrow \Phi^3\,\overline{1} \xrightarrow[\Phi^2\,!_{\Phi\,\overline{1}}]{} \Phi^2\,\overline{1} \xrightarrow[\Phi\,!_{\Phi\,\overline{1}}]{} \Phi\,\overline{1} \xrightarrow[!_{\Phi\,\overline{1}}]{} \overline{1} \end{array}$$

A standard result holds that limits in $\mathbf{Fun}(\mathcal{A}, \mathcal{B})$ may be constructed from pointwise limits in $\mathcal{B}$ [13, p. 112].

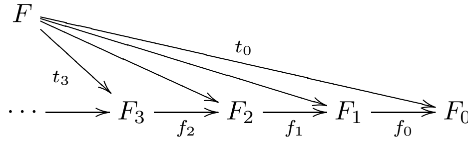
On the way to defining the final fixed point of Φ as a lax monoidal functor, we wish to require that Φ preserve lax monoidal functors. To state this, we need a specialized notion of natural transformation for lax monoidal functors: a *monoidal transformation* between lax monoidal functors $\langle F, \circledast, \top \rangle$ and $\langle F', \circledast', \top' \rangle$ is a natural transformation $h : F \dot{\to} F'$ that preserves the lax monoidal operations:

$$\begin{array}{ccc} & \top & \\ {}^{u}\swarrow & & \searrow^{u'} \\ F\,\top & \xrightarrow[h]{} & F'\,\top \end{array} \qquad \begin{array}{ccc} F\,a \otimes F\,b & \xrightarrow{h \otimes h} & F'\,a \otimes F'\,b \\ \downarrow{\scriptstyle \circledast} & & \downarrow{\scriptstyle \circledast'} \\ F\,(a \otimes b) & \xrightarrow[h]{} & F'\,(a \otimes b) \end{array}$$

Then given monoidal categories $\mathcal{A}$ and $\mathcal{B}$, we can define a category $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$ with lax monoidal functors as objects and monoidal transformations between them as morphisms. Now suppose we have a diagram in $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$, e.g. the chain

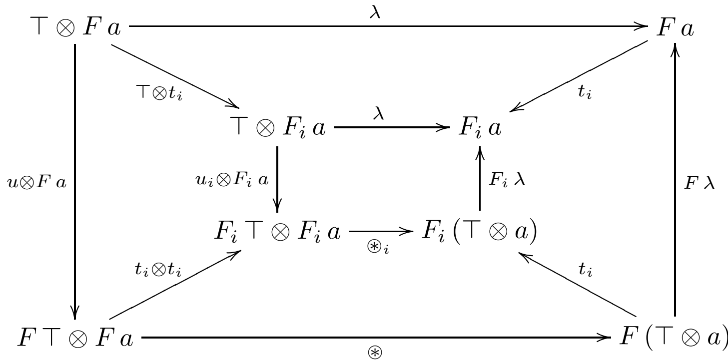
$$\cdots \longrightarrow F_3 \xrightarrow[f_2]{} F_2 \xrightarrow[f_1]{} F_1 \xrightarrow[f_0]{} F_0$$

We can construct a limit in **Fun**$(\mathcal{A}, \mathcal{B})$ (from pointwise limits in $\mathcal{B}$):



To extend F to a limit of this diagram in **Mon**$(\mathcal{A}, \mathcal{B})$, we want to define operations u and $\circledast$ on F such that the t_i are monoidal transformations, i.e. satisfying the following equations in $\mathcal{B}$:

These equations imply that u and $\circledast$ are mediating morphisms to the limits in $\mathcal{B}$, and thus uniquely define them. It remains to show that $\circledast$ is a natural transformation, and that u and $\circledast$ satisfy the identity and associativity laws. Each of these four equations is proven in the same way: we show that the two sides of the equation are equalized by each t_i, as a consequence of the corresponding equation on F_i, and thus, by universality, must be equal. For example, for the left identity law we have the diagram



The central panel is the left identity law for F_i, while the four surrounding panels follow from the definitions of u and $\circledast$ and the naturality of λ and t_i. Thus the two morphisms $\top \otimes F\,a \to F\,a$ on the perimeter of the diagram is equalized by t_i. Since the universality of the limit implies that such a morphism is unique, they must be equal. We have proven:

Proposition 1. *If $\mathcal{B}$ is complete, then so is* **Mon**$(\mathcal{A}, \mathcal{B})$.

Applying this to the chain of the fixed point construction, we have the immediate corollary that the final fixed point of a higher-order functor Φ on lax monoidal

functors is a uniquely determined extension of the final fixed point of Φ on ordinary functors. For example, `ZipList` is the final fixed point of the higher-order functor Φ is defined by $\Phi\, Z\, a = \mathtt{Maybe}\,(a \times Z\, a)$.

3.2 Sums

The dual notion, colimits, is not as easily handled. We can construct sums in special cases, such as adding the identity functor to another lax monoidal functor:

```
data Lift f a = Return a | Others (f a)

instance Functor f => Functor (Lift f) where
    fmap f (Return x) = Return (f x)
    fmap f (Others m) = Others (fmap f m)

instance Monoidal f => Monoidal (Lift f) where
    unit = Return ()
    mult (Return x) (Return y) = Return (x, y)
    mult (Return x) (Others my) = Others (fmap ((,) x) my)
    mult (Others mx) (Return y) = Others (fmap (flip (,) y) mx)
    mult (Others mx) (Others my) = Others (mult mx my)
```

Here pure computations (represented by the identity functor and the constructor `Return`) may be combined with `mult`, but are converted to the other functor if either computation involves that functor.

Applying this construction to the constant functor yields a form of computations with exceptions that collects errors instead of failing at the first error [4,15]:

```
type Except err a = Lift (Constant err) a
```

That is, in a computation `mult e1 e2`, after a failure in `e1`, the whole computation will fail, but not before executing `e2` in case it produces errors that should be reported together with those produced by `e1`.

The fixed point $L \cong \mathtt{Lift}\,(I \times L)$ expands to non-empty lists combined with a "long zip", in which the shorter list is padded with copies of its last element to pair with the remaining elements of the longer list, as suggested by Jeremy Gibbons and Richard Bird[1]:

```
data PadList a = Final a | NonFinal a (PadList a)

instance Functor PadList where
    fmap f (Final x) = Final (f x)
    fmap f (NonFinal x xs) = NonFinal (f x) (fmap f xs)

instance Monoidal PadList where
    unit = Final ()
```

[1] Personal communication, 5 July 2011.

```
mult (Final x) (Final y) = Final (x, y)
mult (Final x) (NonFinal y ys) =
    NonFinal (x, y) (fmap ((,) x) ys)
mult (NonFinal x xs) (Final y) =
    NonFinal (x, y) (fmap (flip (,) y) xs)
mult (NonFinal x xs) (NonFinal y ys) =
    NonFinal (x, y) (mult xs ys)
```

A straightforward generalization is $L \cong \texttt{Lift}\,(I \times (F \circ L))$ for any lax monoidal F, defining forms of long zip for various kinds of tree.

The `Lift` construction is able to combine computations in the identity functor with those of another lax monoidal functor F because there is a monoidal transformation between the two, namely the arrow `pure`. We can generalize:

Proposition 2. *If $\mathcal{J}$ is an upper semi-lattice and $\mathcal{B}$ is a ccc with finite coproducts, a diagram $\Delta : \mathcal{J} \to \mathbf{Mon}(\mathcal{A}, \mathcal{B})$ has a colimit.*

Proof. Define a functor F by

$$F\,a = \sum_{j \in \mathcal{J}} C_j\,(\Delta_j\,a)$$
$$F\,f\,(C_j\,x) = C_j\,(f\,x)$$

where the $C_j : \Delta_j \dot{\to} F$ are tagging injections (constructors) marking the terms of the sum. Then we can define a lax monoidal structure on F as follows:

$$u = C_\bot\,u_\bot$$
$$C_j\;a \circledast C_k\;b = C_{j \sqcup k}\;(\Delta_{j \leq j \sqcup k}\;a, \Delta_{k \leq j \sqcup k}\;b)$$

Naturality of $\circledast$ and the identity and associativity laws follow from simple calculations. □

For example, in the case of `Lift`, $\mathcal{J}$ is a two-element lattice $0 \leq 1$, with $\Delta_0 = I$, $\Delta_1 = F$ and $\Delta_{0 \leq 1} = p$.

4 Generalized Semi-direct Products

A pioneering instance of the applicative interface was the parser combinator library of Swierstra and Duponcheel [18], which we here rehearse in a greatly cut-down form.

These parsers are applied to the output of lexical analysis. Given a type `Symbol` enumerating symbol types, parser input consists of a list of `Token`s, recording the symbol type and its text:

```
type Token = (Symbol, String)
```

For example, there might be a `Symbol` for numeric literals, in which case the corresponding `String` would record the text of the number. Parsers take a list of tokens and return either an error string or a parsed value together with the unparsed remainder of the input:

```
newtype Parser a = P ([Token] -> Either String (a, [Token]))
```

This type is a monad (built by adding to an exception monad a state consisting of a list of tokens), and therefore also an applicative functor. Parsers can be built using primitives to peek at the next symbol, to move to the next token returning the string value of the token read, and to abort parsing reporting an error:

```
nextSymbol :: Parser Symbol
advance :: Parser String
throwError :: String -> Parser a
```

In order to construct recursive descent parsers corresponding to phrases of a grammar, one needs to keep track of whether a phrase can generate the empty string, and also the set of symbols that can begin a phrase (its *first* set). Swierstra and Duponcheel's idea was to define a type containing this information about a phrase, from which a deterministic parser for the phrase could be constructed:

```
data Phrase a = Phrase (Maybe a) (Map Symbol (Parser a))
```

The two components are:

- The type `Maybe a` indicates whether the phrase can generate the empty string, and if so provides a default output value.
- The type `Map Symbol (Parser a)` records which symbols can start the phrase, and provides for each a corresponding deterministic parser.

The `Functor` instance for this type follows from the structure of the type:

```
instance Functor Phrase where
    fmap f (Phrase e t) = Phrase (fmap f e) (fmap (fmap f) t)
```

The idea, then, is to build a value of this type for each phrase of the grammar, with the following conversion to a deterministic parser:

```
parser :: Phrase a -> Parser a
parser (Phrase e t)
  | null t = def
  | otherwise = do
        s <- nextSymbol
        findWithDefault def s t
  where
    def = case e of
        Just x -> return x
        Nothing -> throwError ("expected " ++ show (keys t))
```

A parser for a single symbol, returning its corresponding text, is

```
symbol :: Symbol -> Phrase String
symbol s = Phrase Nothing (singleton s advance)
```

Alternatives are easily built:

```
(<|>) :: Phrase a -> Phrase a -> Phrase a
Phrase e1 t1 <|> Phrase e2 t2 =
    Phrase (e1 `mplus` e2) (t1 `union` t2)
```

In a realistic library, one would want to check that at most one of the alternatives could generate the empty string, and that the first sets were disjoint. The information in the `Phrase` type makes it possible to determine this check before parsing, but we omit this in our simplified presentation.

Now the lax monoidal structure corresponds to the empty phrase and concatenation of phrases. A phrase $\alpha\beta$ can generate the empty string only if both the constituent phrases can, but the emptiness information for α also determines whether the initial symbols of $\alpha\beta$ include those of β in addition to those of α:

```
instance Monoidal Phrase where
    unit = Phrase unit empty
    mult (Phrase e1 t1) (~p2@(Phrase e2 t2)) =
        Phrase (mult e1 e2) (union t1' t2')
      where
        t1' = fmap (`mult` parser p2) t1
        t2' = maybe empty (\ x -> fmap (fmap ((,) x)) t2) e1
```

In Haskell, a tilde marks a pattern as lazy, meaning it is not matched until its components are used. It is used here so that `Phrase` values can be recursively defined, as long as one avoids left recursion.

We might wonder whether this definition is an instance of a general construction. We note that the `Phrase` type is a pair, and the first components are combined using the lax monoidal operations on `Maybe`, independent of the second components. This is similar to a standard construction on monoids, the semi-direct product, which takes a pair of monoids $\langle A, *, 1\rangle$ and $\langle X, +, 0\rangle$ with an action $(\cdot) : A \times X \to X$, and defines a monoid on $A \times X$, with binary operation

$$(a, x) \odot (b, y) = (a * b, x + (a \cdot y))$$

and identity $(1, 0)$. For example Horner's Rule for the evaluation of a polynomial $a_n x^n + \cdots + a_1 x + a_0$ can be expressed as a fold of such an operation over the list $[(x, a_0), (x, a_1), \ldots, (x, a_n)]$, with the immediate consequence that the calculation can be performed in parallel (albeit with repeated calculation of the powers of x).

We shall consider a generalization of the semi-direct product on lax monoidal functors, requiring

- a lax monoidal functor $\langle F, \circledast, u\rangle : \langle \mathcal{A}, \otimes, \top\rangle \to \langle \mathcal{B}, \times, 1\rangle$
- a functor $G : \mathcal{A} \to \mathcal{B}$ with a natural family of monoids $\oplus : G\,a \times G\,a \to G\,a$ and $\emptyset : 1 \to G\,a$.

– an operation $\rtimes : G\,a \times (F\,b \times G\,b) \to G\,(a \otimes b)$ distributing over G:

$$\emptyset \rtimes q = \emptyset \tag{1}$$

$$(x \oplus y) \rtimes q = (x \rtimes q) \oplus (y \rtimes q) \tag{2}$$

– an operation $\ltimes : (F\,a \times G\,a) \times G\,b \to G\,(a \otimes b)$ distributing over G:

$$p \ltimes \emptyset = \emptyset \tag{3}$$

$$p \ltimes (x \oplus y) = (p \ltimes x) \oplus (p \ltimes y) \tag{4}$$

also satisfying

$$(p \ltimes y) \rtimes r = p \ltimes (y \rtimes r) \tag{5}$$

Proposition 3. *Given the above functors and operations, there is a lax monoidal functor* $\langle H, \circledast_H, u_H \rangle : \langle \mathcal{A}, \otimes, \top \rangle \to \langle \mathcal{B}, \times, 1 \rangle$ *defined by*

$$\begin{aligned} H\,a &= F\,a \times G\,a \\ u_H &= (u, \emptyset) \\ (a, x) \circledast_H (b, y) &= (a \circledast b, (x \rtimes (b, y)) \oplus ((a, x) \ltimes y)) \end{aligned}$$

provided that $\ltimes$ *and* $\rtimes$ *are left and right actions on* G*, i.e.*

$$u_H \ltimes z = z \tag{6}$$

$$(p \circledast_H q) \ltimes z = p \ltimes (q \ltimes z) \tag{7}$$

$$x \rtimes u_H = x \tag{8}$$

$$x \rtimes (q \circledast_H r) = (x \rtimes q) \rtimes r \tag{9}$$

Proof. It follows from their definitions that H is a functor and $\circledast_H$ a natural transformation. Next, we show that u_H is the left and right identity of $\circledast_H$:

$$\begin{aligned} u_H \circledast_H (b, y) &= (1 \circledast b, (\emptyset \rtimes (b, y)) \oplus (u_H \ltimes y)) && \text{definition of } \circledast_H,\ u_H \\ &= (1 \circledast b, \emptyset \oplus y) && \text{equations (1) and (6)} \\ &= (b, y) && \text{monoid laws} \end{aligned}$$

$$\begin{aligned} (a, x) \circledast_H u_H &= (a \circledast 1, (x \rtimes u_H) \oplus ((a, x) \ltimes \emptyset)) && \text{definition of } \circledast_H,\ u_H \\ &= (a \circledast 1, x \oplus \emptyset) && \text{equations (8) and (3)} \\ &= (a, x) && \text{monoid laws} \end{aligned}$$

Finally, we must show that $\circledast_H$ is associative:

$$\begin{aligned} &((a, x) \circledast_H (b, y)) \circledast_H (c, z) \\ &= (a \circledast b \circledast c, (((x \rtimes (b, y)) \oplus ((a, x) \ltimes y)) \rtimes (c, z)) \oplus && \text{definition of } \circledast_H \\ &\qquad (((a, x) \circledast_H (b, y)) \ltimes z)) \\ &= (a \circledast b \circledast c, ((x \rtimes (b, y)) \rtimes (c, z)) \oplus (((a, x) \ltimes y) \rtimes (c, z)) \oplus && \text{equation (2)} \\ &\qquad (((a, x) \circledast_H (b, y)) \ltimes z)) \\ &= (a \circledast b \circledast c, (x \rtimes ((b, y) \circledast_H (c, z))) \oplus && \text{(9), (5) and (7)} \\ &\qquad ((a, x) \ltimes (y \rtimes (c, z))) \oplus ((a, x) \ltimes ((b, y) \ltimes z))) \\ &= (a \circledast b * c, (x \rtimes ((b, y) \circledast_H (c, z))) \oplus && \text{equation (4)} \\ &\qquad ((a, x) \ltimes (y \rtimes (c, z)) \oplus ((b, y) \ltimes z))) \\ &= (a, x) \circledast_H ((b, y) \circledast_H (c, z)) && \text{definition of } \circledast_H \end{aligned}$$

□

In particular, Proposition 3 identifies the properties we need to establish to demonstrate that `Phrase` is lax monoidal, and thus applicative.

5 Existentials and Coends

Many applicative functors are constructed using existential quantification, hiding a private representation type. We shall consider the corresponding categorical notion, called a coend.

Abbott, Altenkirch and Ghani [1] consider containers of the form

$$L\,c = \exists\, m.\, K\,m \to c$$

Here m is drawn from a set of shapes $\mathcal{M}$ (a discrete category), the functor $K : \mathcal{M} \to \mathcal{C}$ assigns to each shape a set of positions within containers of that shape, and the function provides a value for each of these positions. For example, the shape of a list is a natural number n giving its length, which K maps to the set of positions $\{0, 1, \ldots, n-1\}$, the indices of the list.

There are several ways that we can extend container functors to obtain useful lax monoidal functors.

If we let $\mathcal{M}$ be the natural numbers plus an upper bound ω, and $K\,n = \{i \mid i < n\}$, then L represents finite and infinite lists. We can define lax monoidal operations:

$$\begin{aligned} u &= (\omega, \mathit{const}\,\top) \\ (m, f) \circledast (n, g) &= (m \sqcap n, \langle f, g \rangle) \end{aligned}$$

That is, u yields an infinite list, and $\circledast$ constructs a list of pairs, whose length is the smaller of the lengths of the arguments. We recognize this functor as another version of the `ZipList` functor defined in Section 3. More generally, if $\mathcal{M}$ has a monoidal structure that is a lower semi-lattice, and $K\,(m_1 \sqcap m_2) \subseteq K\,m_i$, then the lax monoidal structure on L computes zips on containers.

A type comprising arrays of different dimensions can be represented using a shape functor K satisfying $K\,\top \cong 1$ and $K\,(a \otimes b) \cong K\,a \times K\,b$. Then we can define lax monoidal operations with with u constructing a scalar and $\circledast$ being cartesian product:

$$\begin{aligned} u &= (\top, \mathit{const}\,()) \\ (m, f) \circledast (n, g) &= (m \otimes n, f \times g) \end{aligned}$$

We can approximate such multi-dimensional arrays using a Haskell existential type (specified by using the quantifier keyword `forall` before the data constructor):

```
data MultiArray a = forall i. Ix i => MA (Array i a)

instance Functor MultiArray where
    fmap f (MA a) =
        MA (array (bounds a) [(i, f e) | (i, e) <- assocs a])
```

The `unit` operation constructs a scalar, while `mult` forms the cartesian product of two arrays:

```
instance Monoidal MultiArray where
    unit = MA (array ((), ()) [((), ())])
    mult (MA xs) (MA ys) =
        MA (array ((lx, ly), (hx, hy))
            [((i, j), (x, y)) | (i, x) <- assocs xs,
                                (j, y) <- assocs ys])
          where
            (lx, hx) = bounds xs
            (ly, hy) = bounds ys
```

We could extend multi-dimensional arrays by adding a distinguished position, i.e. a cursor within the container:

$$L\,c = \exists\, m.\, K\, m \times (K\, m \to c)$$

When two arrays are combined with `mult`, their cursors are also paired to form a cursor on the product array.

Another example arises in Elliott's analysis of fusion [6], where folds are reified using a type

```
data FoldL b a = FoldL (a -> b -> a) a
```

The type constructor `FoldL` is not a functor, because its argument `a` occurs in both the domain and range of function types. Wishing to apply functorial machinery to these reified folds, Elliott introduced a related type that could be defined as a functor:

```
data WithCont z c = forall a. WC (z a) (a -> c)

instance Functor (WithCont z) where
    fmap g (WC z k) = WC z (g . k)
```

Although `FoldL` is not a functor, it nevertheless has operations similar to `unit` and `mult`. These can be described using a type class similar to `Monoidal`, but without the `Functor` superclass:

```
class Zip z where
    zunit :: z ()
    zmult :: z a -> z b -> z (a, b)
```

The above type constructor `FoldL` is an instance:

```
instance Zip (FoldL b) where
    zunit = FoldL const ()
    zmult (FoldL f1 z1) (FoldL f2 z2) =
        FoldL (\ (x,y) b -> (f1 x b, f2 y b)) (z1, z2)
```

This class is sufficient to define `WithCont` as a lax monoidal functor:

```
instance Zip z => Monoidal (WithCont z) where
    unit = WC zunit (const ())
    mult (WC t1 k1) (WC t2 k2) =
        WC (zmult t1 t2) (\ (x,y) -> (k1 x, k2 y))
```

5.1 Left Kan Extensions

We now consider the general case. Kan extensions are general constructions that have also found applications in programming. The right Kan has been used to construct generalized folds on nested types [10], to fuse types [7], and to construct a monad (the codensity monad) that can be used for program optimization [19,8]. The lax monoidal functors discussed above are instances of the other variety, the left Kan.

Kan extensions have an elegant description at the level of functors. Given a functor $K : \mathcal{M} \to \mathcal{C}$, the left and right Kan extensions along K are defined as the left and right adjoints of the higher-order functor $(\circ K)$ that maps to each functor $\mathcal{C} \to \mathcal{A}$ to a functor $\mathcal{M} \to \mathcal{C}$ [13]. That is, the left Kan extension a functor $T : \mathcal{M} \to \mathcal{A}$ along K is a functor $L : \mathcal{C} \to \mathcal{A}$ with a universal natural transformation $\eta : T \dot{\to} L \circ K$:

$$\begin{array}{ccc} \mathcal{M} & \xrightarrow{K} & \mathcal{C} \\ & {\scriptstyle T}\searrow \;\overset{\eta}{\Longrightarrow}\; \swarrow{\scriptstyle L} & \\ & \mathcal{A} & \end{array}$$

For our purposes, it will be more convenient to use the standard pointwise construction of the left Kan extension as a coend, corresponding to existential quantification in programming languages. For convenience, we assume that the category $\mathcal{A}$ is cartesian closed, and that $\mathcal{C}$ is an $\mathcal{A}$-category [11], i.e. that the “hom-sets” of $\mathcal{C}$ are objects of $\mathcal{A}$, with identity and composition morphisms satisfying the usual laws. Using the more familiar notation $\exists$ in place of the integral sign favoured by category theorists, the left Kan extension of $T : \mathcal{M} \to \mathcal{A}$ along $K : \mathcal{M} \to \mathcal{C}$ is the functor $L : \mathcal{C} \to \mathcal{A}$ defined by

$$L\,c = \exists\, m.\, T\, m \times \mathcal{C}\,(K\,m, c)$$

The examples discussed above are instances of left Kan extensions:

- In the container example, T is the constant functor mapping to 1, and the monoidal structure on $\mathcal{M}$ has $\top = \omega$ and with $\otimes$ as minimum.
- In the example of arrays with cursors, T is identified with K.
- In the `WithCont` example, $\mathcal{M}$ is the subcategory of isomorphisms of $\mathcal{A}$. T can model any type constructor, as although type constructors (like `FoldL` above) need not be functorial, they still preserve isomorphisms.

To state that $L\,c$ is a coend is to say that there is an initial dinatural transformation $\omega : T\,m \times \mathcal{C}\,(K\,m, c) \to L\,c$. This dinaturality of ω is expressed by the equation

$$\omega\,(T\,h\,x, k) = \omega\,(x, k \circ K\,h)$$

That is, the existentially qualified type m is abstract: we can change the representation without affecting the constructed value. The natural transformation $\eta : T \dot{\to} L \circ K$ is defined as

$$\eta\,x = \omega\,(x, id)$$

Initiality of ω means that a natural transformation from L is uniquely determined by its action on terms of the internal language of the form $\omega\,(x, k)$. For example, we can define the action of L on arrows as

$$L\,f\,(\omega\,(x, k)) = \omega\,(x, f \circ k)$$

In order to make L lax monoidal, we shall assume that the functor $K : \mathcal{M} \to \mathcal{C}$ is a *colax monoidal functor*, or equivalently a lax monoidal functor $\mathcal{M}^{\mathrm{op}} \to \mathcal{C}^{\mathrm{op}}$. That is, there are natural transformations

$$\begin{aligned} s &: K\,(a \otimes_{\mathcal{M}} b) \to K\,a \otimes_{\mathcal{C}} K\,b \\ n &: K\,\top_{\mathcal{M}} \to \top_{\mathcal{C}} \end{aligned}$$

such that the following diagrams commute:

$$\begin{array}{ccc} K\,(\top \otimes a) & \xrightarrow{s} & K\,\top \otimes K\,a \\ \downarrow{\scriptstyle K\,\lambda} & & \downarrow{\scriptstyle n \otimes K\,a} \\ K\,a & \xleftarrow[\lambda]{} & \top \otimes K\,a \end{array} \qquad \begin{array}{ccc} K\,(a \otimes \top) & \xrightarrow{s} & K\,a \otimes K\,\top \\ \downarrow{\scriptstyle K\,\rho} & & \downarrow{\scriptstyle K\,a \otimes n} \\ K\,a & \xleftarrow[\rho]{} & K\,a \otimes \top \end{array}$$

$$\begin{array}{ccccc} K\,(a \otimes (b \otimes c)) & \xrightarrow{s} & K\,a \otimes K\,(b \otimes c) & \xrightarrow{K\,a \otimes s} & K\,a \otimes (K\,b \otimes K\,c) \\ \downarrow{\scriptstyle K\,\alpha} & & & & \downarrow{\scriptstyle \alpha} \\ K\,((a \otimes b) \otimes c) & \xrightarrow[s]{} & K\,(a \otimes b) \otimes K\,c & \xrightarrow[s \otimes K\,c]{} & (K\,a \otimes K\,b) \otimes K\,c \end{array}$$

In the special case where the monoidal structure on $\mathcal{C}$ is that of products, there is only one choice for n, namely the unique arrow $K\,\top \to 1$. Moreover in that case $s : K\,(a \otimes b) \to K\,a \times K\,b$ can be broken down into two components: $s = \langle s_1, s_2 \rangle$.

Proposition 4. *If $\mathcal{M}$ and $\mathcal{C}$ are monoidal and $\mathcal{A}$ has finite products, K is colax monoidal and T is lax monoidal, then L is lax monoidal, with*

$$\begin{aligned} u_L &= \omega\,(u_T, n_K) \\ \omega\,(x_1, k_1) \circledast_L \omega\,(x_2, k_2) &= \omega\,(x_1 \circledast_T x_2, k_1 \times k_2 \circ s_K) \end{aligned}$$

This is a special case of Proposition 5, which we shall prove in the next section.

A degenerate example has $\mathcal{M}$ as the trivial category with one object and one morphism, so that T defines a monoid and K selects some object, with $s_i = id$. This describes computations that write output and also read an environment, but in which the output is independent of the environment:

$$L\,c = T \times (K \to c)$$

This applicative functor is a composition of two applicative functors that are also monads, but the composition is not a monad.

Another simple case arises when $\mathcal{M}$ is a cartesian category, in which case an arbitrary functor $K : \mathcal{M} \to \mathcal{C}$ can be made colax monoidal by setting $s_i = K\,\pi_i$. Thus we obtain the following Haskell version of the left Kan:[2]

```
data Lan t k c = forall m. Lan (t m) (k m -> c)

instance (Functor t, Functor k) => Functor (Lan t k) where
    fmap f (Lan x k) = Lan x (f . k)

instance (Monoidal t, Functor k) => Monoidal (Lan t k) where
    unit = Lan unit (const ())
    mult (Lan x1 k1) (Lan x2 k2) =
        Lan (mult x1 x2)
            (\ y -> (k1 (fmap fst y), k2 (fmap snd y)))
```

Although this implementation has the form of a general left Kan extension, it is limited to the Haskell category.

A richer example occurs in the modelling of behaviours of animations using applicative functors by Matlage and Gill [14]. The basic functor comprises a function over a closed interval of time, which can be modelled as pairs of times:

```
data Interval = Between Time Time

instance Monoid Interval where
    mempty = Between inf (-inf)
    Between start1 stop1 <> Between start2 stop2 =
        Between (min start1 start2) (max stop1 stop2)
```

We would like to represent continuous behaviours by a type $\exists\, i.\, K\, i \to T$, for a functor K mapping pairs of times to closed intervals of time, with s_i mapping from larger intervals to smaller by truncation. We cannot express this directly in Haskell, which lacks dependent types, but we can approximate it with a type

```
data Behaviour a = B Interval (Time -> a)
```

provided we hide the representation and provide only an accessor function:

[2] In fact the `Functor` instance requires no assumptions about `t` and `k`, and in the `Monoidal` instance `Zip t` could replace `Monoidal t`.

```
observe :: Behaviour a -> Time -> a
observe (B (Between start stop) f) t =
    f (max start (min stop t))
```

Now this type can be made monoidal with the following definitions, which preserve the abstraction:

```
instance Functor Behaviour where
    fmap f (B i g) = B i (f . g)

instance Monoidal Behaviour where
    unit = B mempty (const ())
    mult b1@(B i1 f1) b2@(B i2 f2) =
        B (i1 <> i2) (\ t -> (observe b1 t, observe b2 t))
```

The final functor used by Matlage and Gill can be obtained by adding constant behaviour using `Lift`:

```
type Active = Lift Behaviour
```

Thus a value of type `Active a` is either constant or a function of time over a given interval. A combination of such behaviours is constant only if both the arguments were.

5.2 The General Case

Our final example is a generalization of the type used by Baars, Löh and Swierstra [3] to construct parsers for permutations of phrases, which we express as

```
data Perms p a = Choice (Maybe a) [Branch p a]
data Branch p a = forall b. Branch (p b) (Perms p (b -> a))
```

This implementation is too subtle to explain in full detain here, but the `Perms` type is essentially an efficient representation of a collection of all the permutations of a set of elementary parsers (or actions, in other applications). The type in the original paper is equivalent to restricting our version of the `Perms` type to values of the forms `Choice (Just x) []` and `Choice Nothing bs`, allowing a single elementary parser to be added to the collection at a time. In contrast, the `mult` methods allows the interleaving of arbitrary collections of actions, allowing us to build them in any order.

The functor instances for these two types are straightforward:

```
instance Functor p => Functor (Perms p) where
    fmap f (Choice def bs) =
        Choice (fmap f def) (map (fmap f) bs)

instance Functor p => Functor (Branch p) where
    fmap f (Branch p perm) = Branch p (fmap (f .) perm)
```

Assuming that `p` is lax monoidal, we will construct instances for `Perms p` and `Branch p`. These types are mutually recursive, but we know that final fixed points preserve applicative functors.

We define an operator `***` as

```
(***) :: Monoidal f => f (a1 -> b1) -> f (a2 -> b2) ->
                        f ((a1,a2) -> (b1,b2))
p *** q = fmap (\ (f,g) (x,y) -> (f x, g y)) (mult p q)
```

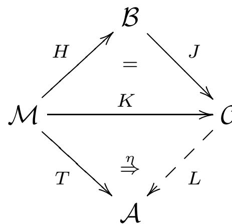
This is an example of the construction of static arrows from applicative functors [15]. Now, assuming that `Perms p` is lax monoidal, we can construct an instance for `Branch p` as a generalized left Kan extension:

```
instance Monoidal p => Monoidal (Branch p) where
    unit = Branch unit (pure id)
    mult (Branch p1 perm1) (Branch p2 perm2) =
        Branch (mult p1 p2) (perm1 *** perm2)
```

The instance for `Perms p` is constructed from the instance for `Branch p` as a generalized semi-direct product, which builds all the interleavings of the two collections of permutations:

```
instance Monoidal p => Monoidal (Perms p) where
    unit = Choice unit []
    mult (t1@(Choice d1 bs1)) (t2@(Choice d2 bs2)) =
        Choice (mult d1 d2)
            (map (`mult` include t2) bs1 ++
             map (include t1 `mult`) bs2)
      where
        include :: Monoidal p => Perms p a -> Branch p a
        include p = Branch unit (fmap const p)
```

To encompass examples such as this, we need a generalization of the left Kan. Suppose the functor K factors through a monoidal category $\mathcal{B}$:



We also assume a natural operator

$$\boxtimes : \mathcal{C}(J\,a, J\,b) \times \mathcal{C}(J\,c, J\,d) \to \mathcal{C}(J\,(a \otimes c), J\,(b \otimes d))$$

(corresponding to `***` above) satisfying unit and associativity laws:

$$J\,\lambda \circ f \boxtimes \top = f \circ J\,\lambda$$
$$J\,\rho \circ \top \boxtimes f = f \circ J\,\rho$$
$$J\,\alpha \circ f \boxtimes (g \boxtimes h) = (f \boxtimes g) \boxtimes h \circ J\,\alpha$$

The situation of ordinary left Kan extensions is the special case where J is the identity functor and $\boxtimes$ is $\otimes_{\mathcal{C}}$. However in general we do not require that $\boxtimes$ be a functor. The key example of a structure with such an operator is an enriched premonoidal category, or "arrow" [2,9].

Proposition 5. *If $\mathcal{M}$ and $\mathcal{B}$ are monoidal, $\mathcal{A}$ has finite products, H is colax monoidal and T is lax monoidal, then $F = L \circ J$ is lax monoidal, with*

$$\begin{aligned} F\,a &= \exists\, m.\, T\,m \times (J\,(H\,m) \to J\,a) \\ F\,f\,(\omega\,(x,k)) &= \omega\,(x, J\,f \circ k) \\ u_F &= \omega\,(u_T, J\,n_H) \\ \omega\,(x_1,k_1) \circledast_F \omega\,(x_2,k_2) &= \omega\,(x_1 \circledast_T x_2, k_1 \boxtimes k_2 \circ J\,s_H) \end{aligned}$$

Instead of proving this directly, we show that the functor $G : \mathcal{M}^{\mathrm{op}} \times \mathcal{M} \times \mathcal{A}$ defined by

$$G\,(m', m, a) = T\,m \times (J\,(H\,m') \to J\,a)$$

is itself lax monoidal, and then use a general result about coends of lax monoidal functors. To see that G is lax monoidal, we note that T is lax monoidal, so we only need to show that the second component is. The left identity case is

$$\begin{aligned} F\,\lambda \circ id \boxtimes k \circ J\,(n_H \times 1 \circ s_H) &= k \circ J\,(\lambda \circ n_H \times 1 \circ s_H) && \text{left identity of } \boxtimes \\ &= k \circ J\,(H\,\lambda) && \text{left identity of } H \end{aligned}$$

The right identity case is similar. Associativity relies on the associativity of $\boxtimes$:

$$\begin{aligned} & J\,\alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3 \circ J\,s_H) \circ J\,s_H \\ &= J\,\alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3) \circ J\,(id \times s_H \circ s_H) && \text{naturality of } \boxtimes \\ &= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J\,(\alpha \circ id \times s_H \circ s_H) && \text{associativity of } \boxtimes \\ &= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J\,(s_H \times id \circ s_H \circ H\,\alpha) && \text{associativity of } s_H \end{aligned}$$

Thus it suffices to show that coends preserve lax monoidal functors, which is our final result.

Proposition 6. *Given monoidal categories $\mathcal{A}$ and $\mathcal{B}$ and a ccc $\mathcal{C}$, with a lax monoidal functor $G : \mathcal{A}^{\mathrm{op}} \times \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, then the coend $F\,b = \exists\, a.\, G\,(a, a, b)$ is also lax monoidal, with*

$$\begin{aligned} F\,b &= \exists\, a.\, G\,(a, a, b) \\ F\,f\,(\omega\,x) &= \omega\,(G\,(id, id, f)\,x) \\ u_F &= \omega\,u_G \\ \omega\,x_1 \circledast_F \omega\,x_2 &= \omega\,(x_1 \circledast_G x_2) \end{aligned}$$

Proof. It is a standard result that a parameterized coend such as F defines a functor. Naturality of $\circledast_F$ follows from naturality of $\circledast_G$:

$$\begin{aligned} & F\,(f_1 \otimes f_2)\,(\omega\,x_1 \circledast_F \omega\,x_2) \\ &= F\,(f_1 \otimes f_2)\,(\omega\,(x_1 \circledast_G x_2)) && \text{definition of } \circledast_F \\ &= \omega\,(G\,(id, id, f_1 \otimes f_2)\,(x_1 \circledast_G x_2)) && \text{definition of } F \\ &= \omega\,(G\,(id, id, f_1)\,x_1 \circledast_G G\,(id, id, f_2)\,x_2) && \text{naturality of } \circledast_G \\ &= \omega\,(G\,(id, id, f_1)\,x_1) \circledast_F \omega\,(G\,(id, id, f_2)\,x_2) && \text{definition of } \circledast_F \\ &= F\,f_1\,(\omega\,x_1) \circledast_F F\,f_2\,(\omega\,x_2) && \text{definition of } F \end{aligned}$$

Similarly the left identity law for F follows from the corresponding law for G:

$$\begin{array}{llr} F\,\lambda\,(u_F \circledast_F \omega\,x) & = F\,\lambda\,(\omega\,u_G \circledast_F \omega\,x) & \text{definition of } u_F \\ & = F\,\lambda\,(\omega\,(u_G \circledast_G x)) & \text{definition of } \circledast_F \\ & = \omega\,(G\,(id, id, \lambda)\,(u_G \circledast_G x)) & \text{definition of } F \\ & = \omega\,(G\,(id, \lambda \circ \lambda^{-1}, \lambda)\,(u_G \circledast_G x)) & \text{isomorphism} \\ & = \omega\,(G\,(\lambda^{-1}, \lambda, \lambda)\,(u_G \circledast_G x)) & \text{dinaturality of } \omega \\ & = \omega\,x & \text{left identity of } G \end{array}$$

The right identity case is similar.

Finally, the associativity law for $\circledast_F$ follows from the associativity of $\circledast_G$:

$$\begin{array}{lr} F\,\alpha\,(\omega\,x \circledast_F (\omega\,y \circledast_F \omega\,z)) & \\ = F\,\alpha\,(\omega\,x \circledast_F \omega\,(y \circledast_G z)) & \text{definition of } \circledast_F \\ = F\,\alpha\,(\omega\,(x \circledast_G (y \circledast_G z))) & \text{definition of } \circledast_F \\ = \omega\,(G\,(id, id, \alpha)\,(x \circledast_G (y \circledast_G z))) & \text{definition of } F \\ = \omega\,(G\,(id, \alpha \circ \alpha^{-1}, \alpha)\,(x \circledast_G (y \circledast_G z))) & \text{isomorphism} \\ = \omega\,(G\,(\alpha^{-1}, \alpha, \alpha)\,(x \circledast_G (y \circledast_G z))) & \text{dinaturality of } \omega \\ = \omega\,((x \circledast_G y) \circledast_G z) & \text{associativity of } G \\ = \omega\,(x \circledast_G y) \circledast_F \omega\,z & \text{definition of } \circledast_F \\ = (\omega\,x \circledast_F \omega\,y) \circledast_F \omega\,z & \text{definition of } \circledast_F \end{array}$$

□

As a further example, we have the coend encoding of the final fixed point $\nu\,F$ of a functor $F : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$:

$$\nu\,F\,a \cong \exists\,b.\,b \times (b \to F\,(a, b))$$

which is a coend of $G\,(b', b, a) = b \times (b' \to F\,(a, b))$, and yields the same applicative functor as discussed in Section 3.

6 Conclusion

We have established a number of general constructions of lax monoidal functors, and therefore of applicative functors. In examples such as the permutation phrases of Section 5.2, we showed that by combining these constructions we could account for quite complex (and useful) applicative functors, avoiding the need for specific proofs of their laws. By breaking the functors down into simple building blocks, we have clarified their relationships, as well providing the tools to build more applications. The next stage is to examine the possible combinations, and to consider other constructions.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Categories of Containers. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 23–38. Springer, Heidelberg (2003)
2. Atkey, R.: What is a categorical model of arrows? Electronic Notes on Theoretical Computer Science 229(5), 19–37 (2011)

3. Baars, A.I., Löh, A., Doaitse Swierstra, S.: Parsing permutation phrases. Journal of Functional Programming 14(6), 635–646 (2004)
4. Coutts, D.: Arrows for errors: Extending the error monad (2002); unpublished presentation at the Summer School on Advanced Functional Programming
5. Eilenberg, S., Kelly, G.M.: Closed categories. In: Eilenberg, S., Harrison, D.K., Röhrl, H., MacLane, S. (eds.) Proceedings of the Conference on Categorical Algebra, pp. 421–562. Springer (1966)
6. Elliott, C.: Denotational design with type class morphisms. Technical Report 2009-01, LambdaPix (2009)
7. Hinze, R.: Type Fusion. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 92–110. Springer, Heidelberg (2011)
8. Hinze, R.: Kan extensions for program optimisation, or: Art and dan explain an old trick. In: Gibbons, J., Nogueira, P. (eds.) Mathematics of Program Construction (2012)
9. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37(1-3), 67–111 (2000)
10. Johann, P., Ghani, N.: A principled approach to programming with nested types in Haskell. Higher-Order and Symbolic Computation 22(2), 155–189 (2009)
11. Kelly, G.M.: Basic concepts of enriched category theory. London Mathematical Society Lecture Note Series, vol. 64. Cambridge University Press (1982)
12. Lambek, J., Scott, P.J.: Introduction to Higher Order Categorical Logic. Cambridge Studies in Advanced Mathematics, vol. 7. Cambridge University Press, Cambridge (1986)
13. Lane, S.M.: Categories for the Working Mathematician. Springer, New York (1971)
14. Matlage, K., Gill, A.: Every Animation Should Have a Beginning, a Middle, and an End: A Case Study of Using a Functor-Based Animation Language. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 150–165. Springer, Heidelberg (2011)
15. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)
16. Moggi, E.: Computational lambda-calculus and monads. In: Logic in Computer Science, pp. 14–23. IEEE Computer Society Press (1989)
17. Moggi, E.: An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh (1990)
18. Swierstra, S.D., Duponcheel, L.: Deterministic, Error-Correcting Combinator Parsers. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 184–207. Springer, Heidelberg (1996)
19. Voigtländer, J.: Asymptotic Improvement of Computations over Free Monads. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 388–403. Springer, Heidelberg (2008)
20. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2(4), 461–493 (1992)

Kan Extensions for Program Optimisation *Or:* Art and Dan Explain an Old Trick

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@cs.ox.ac.uk
http://www.cs.ox.ac.uk/ralf.hinze/

Abstract. Many program optimisations involve transforming a program in direct style to an equivalent program in continuation-passing style. This paper investigates the theoretical underpinnings of this transformation in the categorical setting of monads. We argue that so-called absolute Kan Extensions underlie this program optimisation. It is known that every Kan extension gives rise to a monad, the codensity monad, and furthermore that every monad is isomorphic to a codensity monad. The end formula for Kan extensions then induces an implementation of the monad, which can be seen as the categorical counterpart of continuation-passing style. We show that several optimisations are instances of this scheme: Church representations and implementation of backtracking using success and failure continuations, among others. Furthermore, we develop the calculational properties of Kan extensions, powers and ends. In particular, we propose a two-dimensional notation based on string diagrams that aims to support effective reasoning with Kan extensions.

Keywords: Haskell, CPS, adjunction, Kan extension, codensity monad, power, end, Church representation, backtracking, string diagram.

1 Introduction

Say you have implemented some computational effect using a monad, and you note that your monadic program is running rather slow. There is a folklore trick to speed it up: transform the monad M into continuation-passing style.

$$\begin{aligned}
&\textbf{type}\,\mathsf{C}\,a = \forall z\,.\,(a \to \mathsf{M}\,z) \to \mathsf{M}\,z\\
&\textbf{instance}\,\mathit{Monad}\,\mathsf{C}\,\textbf{where}\\
&\quad \mathit{return}\,a = \lambda c \to c\,a\\
&\quad m \mathbin{>\!\!>\!\!=} k = \lambda c \to m\,(\lambda a \to k\,a\,c)
\end{aligned}$$

The type constructor C is a monad, regardless of M. The origins of this trick seem to be unknown. It is implicit in Hughes' tutorial on designing a pretty-printing library [18], which introduces a related construction called context-passing style. Interestingly, Hughes makes C parametric in the type variable z, rather than

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 324–362, 2012.
© Springer-Verlag Berlin Heidelberg 2012

locally quantifying over z. Presumably, this is because no Haskell system supported rank-2 types at the time of writing the paper. Only in 1996 Augustsson added support for local universal quantification to the Haskell B. Compiler (hbc 0.9999.0) and I started using it.

My goal was to provide a fast implementation of backtracking in Haskell—the first promising results were detailed in a long technical report [12]. Briefly, the idea is to use two continuations, a success and a failure continuation. Failure and choice can then be implemented as follows.

$$\begin{aligned}
&\textbf{type}\,\mathsf{B}\,a = \forall z\,.\,(a \to z \to z) \to z \to z\\
&\mathit{fail}\ :\ \mathsf{B}\,a\\
&\mathit{fail} = \lambda s\,f \to f\\
&(\shortmid) : \mathsf{B}\,a \to \mathsf{B}\,a \to \mathsf{B}\,a\\
&m \shortmid n = \lambda s\,f \to m\,s\,(n\,s\,f)
\end{aligned}$$

We shall see later that this implementation of backtracking is an instance of the trick. This particular application can be traced back to a paper by Mellish and Hardy [29], who showed how to integrate Prolog into the POPLOG environment. Their setting is an imperative one; Danvy and Filinski [9] explained how to recast the approach in purely functional terms. Since then the trick has made several appearances in the literature, most notably [13,8,33,20,28].

The purpose of this paper is to justify the trick and explain its far-reaching applications. There is no shortage of proofs in the aforementioned papers, but no work relates the original monad M to the improved monad C. Since the transformation is labelled 'program optimisation', one would hope that M is isomorphic to C, but sadly this is not the case. We shall see that $\mathsf{M}\,a$ is instead isomorphic to $\forall z\,.\,(a \to \mathsf{R}\,z) \to \mathsf{R}\,z$ for some magic functor R related to M.

The proofs will be conducted in a categorical setting. We will argue that continuations are an *implementation* of a categorical concept known as a right Kan extension, Kan extension for short. For the most part, we will prove and program against the *specification* of a Kan extension. This is in contrast to the related work, including my papers, which take the rank-2 types as the point of departure. (One could argue that this violates one of the fundamental principles of computer science, that we should program against an interface, not an implementation.) It should come as little surprise that all of the necessary categorical concepts and results appear either explicitly or implicitly in Mac Lane's masterpiece [27]. In fact, the first part of this paper solves Exercise X.7.3 of the textbook. Specifically, we show that

– a Kan extension gives rise to a monad, the so-called codensity monad, thereby solving Exercise X.7.3(a);
– every monad is isomorphic to a codensity monad, solving Exercise X.7.3(c);
– we show that Kan extensions can be implemented using ends and powers [27, Section X.4], which we argue is the gist of continuation-passing style.

Combined these results provide a powerful optimisation scheme. Although the categorical results are known, none of the papers cited above seems to note

the intimate relationship. This paper sets out to fill this gap, showing the relevance of the categorical construction to programming. Furthermore, it aims to complement Mac Lane's diagrammatic reasoning by a calculational approach. Specifically, the paper makes the following original contributions:

- we demonstrate that many program optimisations are instances of the optimisation scheme: Church representations etc;
- we develop the calculational properties of Kan extensions, powers and ends;
- to support effective reasoning, we propose a two-dimensional notation for Kan extensions based on string diagrams.

It is the last aspect I am most excited about. The algebra of programming has aptly demonstrated the power of equational reasoning for program calculation. However, one-dimensional notation reaches its limits when it comes to reasoning about natural transformations, as we will set out to do. Natural transformations are a 2-categorical concept, which lends itself naturally to a two-dimensional notation. Many laws, which otherwise have to be invoked explicitly, are built into the notation.

The remainder of the paper is structured as follows. Section 2 introduces some background, notably adjunctions and monads. The knowledgeable reader may safely skip the material, except perhaps for Section 2.2, which introduces string diagrams. Section 3 defines the notion of a Kan extension and suggests a two-dimensional notation based on string diagrams. Section 4 applies the notation to show that every Kan extension induces a monad, the codensity monad. Sections 5 and 6 move on to discuss the existence of Kan extensions. Section 5 proves that every adjunction induces a Kan extension, and that every monad is isomorphic to a codensity monad. Section 6 derives the so-called end formula for Kan extensions. The development requires the categorical notions of powers and ends, which are introduced in Sections 6.1 and 6.2, respectively. The framework has a multitude of applications, which Section 7 investigates. Finally, Section 8 reviews related work and Section 9 concludes.

A basic knowledge of category theory is assumed. Appendix A summarises the main facts about composition of functors and natural transformations.

2 Background

2.1 Adjunction

The notion of an adjunction was introduced by Daniel Kan in 1958 [23]. Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane [27, p.vii] famously saying, "Adjoint functors arise everywhere." From the perspective of program calculation, adjunctions provide a unified framework for program transformation. As with every deep concept, there are various ways to define the notion of an adjunction. The simplest is perhaps the following:

Let $\mathscr{L}$ and $\mathscr{R}$ be categories. The functors $\mathsf{L} : \mathscr{L} \leftarrow \mathscr{R}$ and $\mathsf{R} : \mathscr{L} \rightarrow \mathscr{R}$ are *adjoint*, written $\mathsf{L} \dashv \mathsf{R}$ and depicted

$$\mathscr{L} \underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{R} \ ,$$

if and only if there is a bijection between the hom-sets

$$\lfloor - \rfloor : \mathscr{L}(\mathsf{L}\, A, B) \cong \mathscr{R}(A, \mathsf{R}\, B) : \lceil - \rceil \ , \tag{1}$$

that is natural both in A and B. The functor L is said to be a *left adjoint* for R, while R is L's *right adjoint*. The isomorphism $\lfloor - \rfloor$ is called the *left adjunct* with $\lceil - \rceil$ being the *right adjunct*. (The notation $\lfloor - \rfloor$ for the left adjunct is chosen as the opening bracket resembles an 'L'. Likewise—but this is admittedly a bit laboured—the opening bracket of $\lceil - \rceil$ can be seen as an angular 'r'.)

That $\lfloor - \rfloor$ and $\lceil - \rceil$ are mutually inverse can be captured using an equivalence.

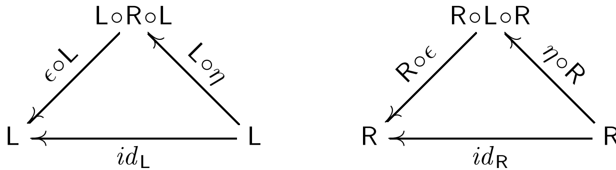
$$f = \lceil g \rceil \iff \lfloor f \rfloor = g \tag{2}$$

The left-hand side lives in $\mathscr{L}$, and the right-hand side in $\mathscr{R}$.

Let us spell out the naturality properties of the adjuncts: $\lceil g \rceil \cdot \mathsf{L}\, h = \lceil g \cdot h \rceil$ and $\mathsf{R}\, k \cdot \lfloor f \rfloor = \lfloor k \cdot f \rfloor$. The formulæ imply $\lceil id \rceil \cdot \mathsf{L}\, h = \lceil h \rceil$ and $\mathsf{R}\, k \cdot \lfloor id \rfloor = \lfloor k \rfloor$. Consequently, the adjuncts are uniquely defined by their images of the identity: $\epsilon = \lceil id \rceil$ and $\eta = \lfloor id \rfloor$. An alternative definition of adjunctions is based on these two natural transformations, which are called the *counit* $\epsilon : \mathsf{L} \circ \mathsf{R} \mathrel{\dot{\to}} \mathsf{Id}$ and the *unit* $\eta : \mathsf{Id} \mathrel{\dot{\to}} \mathsf{R} \circ \mathsf{L}$ of the adjunction. The units must satisfy the so-called *triangle identities*:

$$\epsilon \circ \mathsf{L} \cdot \mathsf{L} \circ \eta = id_{\mathsf{L}} \ , \tag{3a}$$
$$\mathsf{R} \circ \epsilon \cdot \eta \circ \mathsf{R} = id_{\mathsf{R}} \ . \tag{3b}$$

The diagrammatic rendering explains the name triangle identities.



Remark 1. To understand concepts in category theory it is helpful to look at a simple class of categories: *preorders*, reflexive and transitive relations. Every preorder gives rise to a category whose objects are the elements of the preorder and whose arrows are given by the ordering relation. These categories are special as there is at most one arrow between two objects. Reflexivity provides the identity arrow, transitivity allows us to compose two arrows. A functor between two preorders is a *monotone function*, a mapping on objects that respects the

underlying ordering: $a \leqslant b \Longrightarrow f\ a \leqslant f\ b$. A natural transformation between two monotone functions corresponds to a point-wise ordering: $f \mathrel{\dot{\leqslant}} g \Longleftrightarrow \forall x\ .\ f\ x \leqslant g\ x$. When appropriate we shall specialise the development to preorders.

The preorder equivalent of an adjunction is a *Galois connection*. Let L and R be preorders. The maps $l : L \leftarrow R$ and $r : L \to R$ form a Galois connection between L and R if and only if

$$l\ a \leqslant b \text{ in } L \quad \Longleftrightarrow \quad a \leqslant r\ b \text{ in } R \ , \tag{4}$$

for all $a \in R$ and $b \in L$.

An instructive example of a right adjoint is the floor function $\lfloor - \rfloor : \mathbb{R} \to \mathbb{Z}$ (not to be confused with the notation for left adjuncts), whose left adjoint is the inclusion map $\iota : \mathbb{Z} \to \mathbb{R}$. We have

$$\iota\ n \leqslant x \text{ in } \mathbb{R} \quad \Longleftrightarrow \quad n \leqslant \lfloor x \rfloor \text{ in } \mathbb{Z} \ ,$$

for all $n \in \mathbb{Z}$ and $x \in \mathbb{R}$. The inclusion map also has a left adjoint, the ceiling function $\lceil - \rceil : \mathbb{R} \to \mathbb{Z}$.

The definition of an adjunction in terms of the units corresponds to the following property: the maps $l : L \leftarrow R$ and $r : L \to R$ form a Galois connection between L and R if and only if l and r are monotone, $l \cdot r \mathrel{\dot{\leqslant}} id$ and $id \mathrel{\dot{\leqslant}} r \cdot l$. Since in a preorder there is at most one arrow between two objects, we furthermore have $r \cdot l \cdot r \mathrel{\dot{\cong}} r$ and $l \mathrel{\dot{\cong}} l \cdot r \cdot l$.

In general, to interpret a category-theoretic result in the setting of preorders, we only consider the types of the arrows: for example, the bijection (1) simplifies to (4). Conversely, an order-theoretic proof can be interpreted as a typing derivation. Category theory has been characterised as *coherently constructive lattice theory* [2], and to generalise an order-theoretic result we additionally have to impose coherence conditions—the triangle identities in the case of adjunctions. □

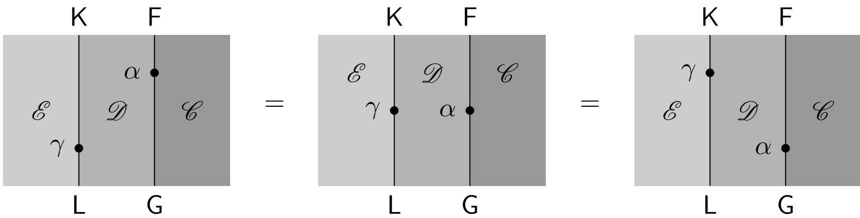
2.2 String Diagram

Throughout the paper we shall recast natural transformations and their properties in two-dimensional notation, based on *string diagrams* [31]. Categories, functors and natural transformations form a so-called 2-category, which lends itself naturally to a two-dimensional notation. From a calculational point of view, two-dimensional notation is attractive because several laws, notably the interchange law (56), are built into the notation. When we use one-dimensional notation, we have to invoke these laws explicitly. (For similar reasons we routinely use one-dimensional notation for objects and arrows: the monoidal properties of identity and composition are built into the notation.)

Here are the string diagrams for the units of an adjunction.

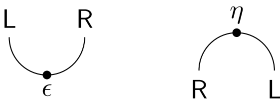
A string diagram is a planar graph. A region in the graph corresponds to a category, a line corresponds to a functor, and a point (usually drawn as a small circle) corresponds to a natural transformation. For readability lines are implicitly directed, and we stipulate that the flow is from right to left for horizontal composition, $\beta \circ \alpha$, and from top to bottom for vertical composition $\beta \cdot \alpha$. The counit ϵ has two incoming functors, L and R, and no outgoing functor—the dotted line hints at the identity functor, which is usually omitted. A natural transformation of type $\mathsf{F} \circ \mathsf{G} \circ \mathsf{H} \dot{\to} \mathsf{T} \circ \mathsf{U}$, for example, would be shown as a point with three incoming arrows (from above) and two outgoing arrows (to below).

Diagrams that differ only in the vertical position of natural transformations are identified—this is the import of the interchange law (56).



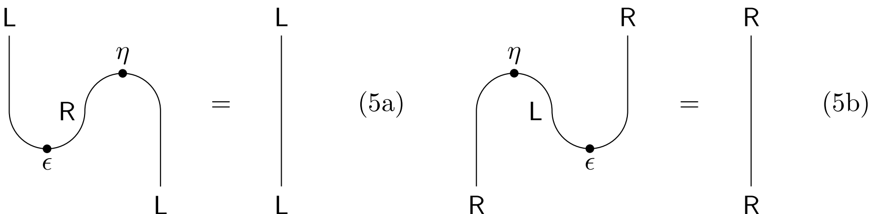
Thus, $\gamma \circ \mathsf{G} \cdot \mathsf{K} \circ \alpha$, $\gamma \circ \alpha$ and $\mathsf{L} \circ \alpha \cdot \gamma \circ \mathsf{F}$ correspond to the same diagram. For turning a string diagram into standard notation it is helpful to draw horizontal lines through the points that denote natural transformations. Each of these lines corresponds to a horizontal composition, where a vertical line that crosses the horizontal line is interpreted as the identity on the respective functor. This step yields $\gamma \circ \mathsf{G}$ and $\mathsf{K} \circ \alpha$ for the diagram on the left. The vertical composition of these terms then corresponds to the diagram.

To reduce clutter we shall usually not label or colour the regions. Also, identity functors (drawn as dotted lines above) and identity natural transformations are omitted. With these conventions the string diagrams for the units simplify to half circles.



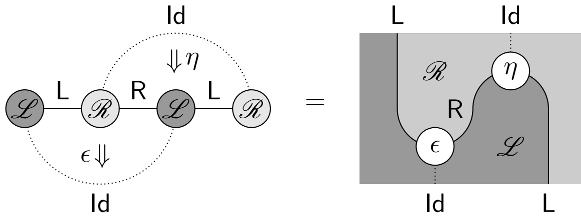
A cup signifies the counit ϵ and a cap the unit η.

It is important to keep in mind that, unlike a commutative diagram, a string diagram is a term, not a property. Properties such as the triangle identities (3a)-(3b) are still written as equations.



The triangle identities have an appealing visual interpretation: they allow us to pull a twisted string straight.

Remark 2. There is an alternative, perhaps more traditional two-dimensional notation, where categories are shown as points, functors as lines and natural transformations as regions (often labelled with a double arrow).



The traditional diagram on the left is the Poincaré dual of the string diagram on the right: d-dimensional objects on the left are mapped to $(2 - d)$-dimensional objects on the right, and vice versa. □

2.3 Monad

To incorporate computational effects such as IO, Haskell has adopted the categorical concept of a monad [30]. As with adjunctions, there are several ways to define the notion. The following is known as the monoidal definition.

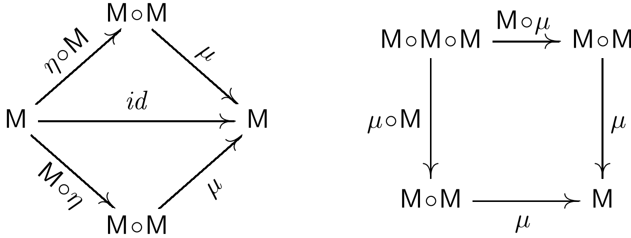
A monad consists of an endofunctor M and natural transformations

$$\eta : \mathsf{Id} \to \mathsf{M} \ ,$$
$$\mu : \mathsf{M} \circ \mathsf{M} \to \mathsf{M} \ .$$

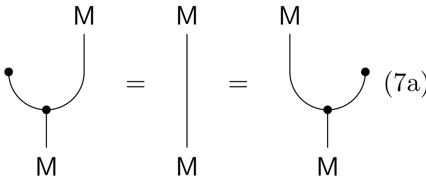
From the perspective of Haskell, a monad is a mechanism that supports effectful computations. A monadic program is an arrow of type $A \to \mathsf{M}\, B$, where the monad is wrapped around the target. The operations that come with a monad organise effects: the unit η (also called "return") creates a pure computation, the multiplication μ (also called "join") merges two layers of effects. The two operations have to work together:

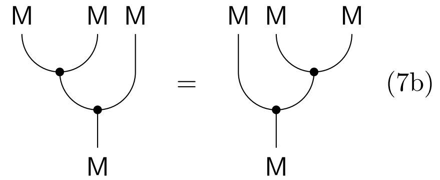
$$\mu \cdot \eta \circ \mathsf{M} = id_{\mathsf{M}} \ , \tag{6a}$$
$$\mu \cdot \mathsf{M} \circ \eta = id_{\mathsf{M}} \ , \tag{6b}$$
$$\mu \cdot \mu \circ \mathsf{M} = \mu \cdot \mathsf{M} \circ \mu \ . \tag{6c}$$

The unit laws (6a) and (6b) state that merging a pure with a potentially effectful computation gives the effectful computation. The associative law (6c) expresses that the two ways of merging three layers of effects are equivalent.



In two-dimensional notation, the natural transformations correspond to constructors of binary leaf trees: η creates a leaf, μ represents a fork. The monad laws correspond to transformations on binary trees: the unit laws allow us to prune or to add leaves and the associative law captures a simple tree rotation.





Every adjunction $\mathsf{L} \dashv \mathsf{R}$ induces a monad [17]:

$$\mathsf{M} = \mathsf{R}{\circ}\mathsf{L} \ , \tag{8a}$$

$$\eta = \eta \ , \tag{8b}$$

$$\mu = \mathsf{R}{\circ}\epsilon{\circ}\mathsf{L} \ . \tag{8c}$$

The monad operations have simple implementations in terms of the units: the unit of the adjunction serves as the unit of the monad; the multiplication is defined in terms of the counit. We will prove this result twice, a first time using one-dimensional notation and a second time using two-dimensional notation.

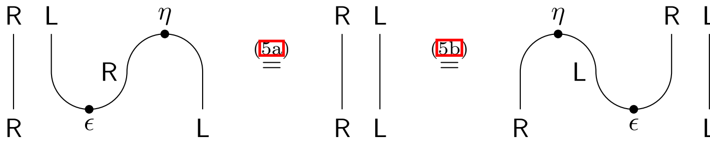
The unit laws (6a)–(6b) are consequences of the triangle identities (3a)–(3b).

$$
\begin{array}{ll}
& \mu \cdot \eta{\circ}\mathsf{M} \\
= & \{ \text{ definitions (8a)–(8c) } \} \\
& \mathsf{R}{\circ}\epsilon{\circ}\mathsf{L} \cdot \eta{\circ}\mathsf{R}{\circ}\mathsf{L} \\
= & \{ \ -{\circ}\mathsf{L} \text{ functor (54c) } \} \\
& (\mathsf{R}{\circ}\epsilon \cdot \eta{\circ}\mathsf{R}){\circ}\mathsf{L} \\
= & \{ \text{ triangle identity (3b) } \} \\
& id_{\mathsf{R}}{\circ}\mathsf{L} \\
= & \{ \ -{\circ}\mathsf{L} \text{ functor (54c) } \} \\
& id_{\mathsf{R}{\circ}\mathsf{L}} \\
= & \{ \text{ definition of } \mathsf{M} \text{ (8a) } \} \\
& id_{\mathsf{M}}
\end{array}
\qquad
\begin{array}{ll}
& \mu \cdot \mathsf{M}{\circ}\eta \\
= & \{ \text{ definitions (8a)–(8c) } \} \\
& \mathsf{R}{\circ}\epsilon{\circ}\mathsf{L} \cdot \mathsf{R}{\circ}\mathsf{L}{\circ}\eta \\
= & \{ \ \mathsf{R}{\circ}- \text{ functor (54a) } \} \\
& \mathsf{R}{\circ}(\epsilon{\circ}\mathsf{L} \cdot \mathsf{L}{\circ}\eta) \\
= & \{ \text{ triangle identity (3a) } \} \\
& \mathsf{R}{\circ}id_{\mathsf{L}} \\
= & \{ \ \mathsf{R}{\circ}- \text{ functor (54a) } \} \\
& id_{\mathsf{R}{\circ}\mathsf{L}} \\
= & \{ \text{ definition of } \mathsf{M} \text{ (8a) } \} \\
& id_{\mathsf{M}}
\end{array}
$$

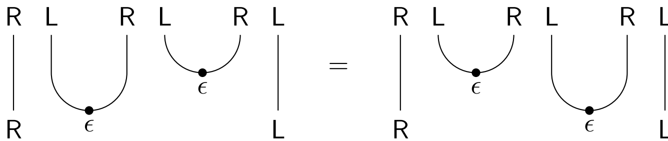
The associative law (6c) follows from the coherence property of horizontal composition, the interchange law (56).

$$
\begin{aligned}
& \mu \cdot \mu \circ \mathsf{M} \\
= \; & \{ \text{ definition of } \mathsf{M} \text{ (8a) and } \mu \text{ (8c) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{R} \circ \epsilon \circ \mathsf{L} \circ \mathsf{R} \circ \mathsf{L} \\
= \; & \{ \; \mathsf{R} \circ - \text{ and } - \circ \mathsf{L} \text{ functors (54b) and (54d) } \} \\
& \mathsf{R} \circ (\epsilon \cdot \epsilon \circ \mathsf{L} \circ \mathsf{R}) \circ \mathsf{L} \\
= \; & \{ \text{ interchange law (56): } \mathsf{Id} \circ \epsilon \cdot \epsilon \circ (\mathsf{L} \circ \mathsf{R}) = \epsilon \circ \epsilon = \epsilon \circ \mathsf{Id} \cdot (\mathsf{L} \circ \mathsf{R}) \circ \epsilon \; \} \\
& \mathsf{R} \circ (\epsilon \cdot \mathsf{L} \circ \mathsf{R} \circ \epsilon) \circ \mathsf{L} \\
= \; & \{ \; \mathsf{R} \circ - \text{ and } - \circ \mathsf{L} \text{ functors (54b) and (54d) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{R} \circ \mathsf{L} \circ \mathsf{R} \circ \epsilon \circ \mathsf{L} \\
= \; & \{ \text{ definition of } \mathsf{M} \text{ (8a) and } \mu \text{ (8c) } \} \\
& \mu \cdot \mathsf{M} \circ \mu
\end{aligned}
$$

The proofs using one-dimensional notation exhibit a lot of noise. In contrast, the proofs in two-dimensional notation carve out the essential steps. For the unit laws, we use the triangle identities.



The associative law requires no proof as the diagrams for the left- and the right-hand side are identified.



In other words, the one-dimensional proof only contains administrative steps.

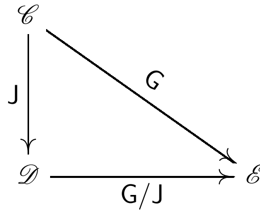
Remark 3. The preorder equivalent of a monad is a *closure operator*. Let P be a preorder. A map $m : P \to P$ is a closure operator on P if it is extensive, $id \mathrel{\dot{\leqslant}} m$, and idempotent, $m \cdot m \mathrel{\dot{\cong}} m$. (The latter condition can be weakened to $m \cdot m \mathrel{\dot{\leqslant}} m$ since $m \mathrel{\dot{\leqslant}} m \cdot id \mathrel{\dot{\leqslant}} m \cdot m$ as composition is monotone.)

A Galois connection $l \dashv r$ between L and R induces a closure operator $m = r \cdot l$ on R. For example, the composition of inclusion $\iota : \mathbb{Z} \to \mathbb{R}$ and the ceiling function $\lceil - \rceil : \mathbb{R} \to \mathbb{Z}$ is a closure operator on $\mathbb{R}$. □

3 Kan Extension—Specification

The continuation types shown in the introduction *implement* so-called right Kan extensions. This section *specifies* the concept formally. As to be expected, the specification will be quite different from the implementation.

Let $\mathsf{J} : \mathscr{C} \to \mathscr{D}$ be a functor. You may want to think of J as an inclusion functor. The functor part of the right Kan extension $\mathsf{G}/\mathsf{J} : \mathscr{D} \to \mathscr{E}$ extends a functor $\mathsf{G} : \mathscr{C} \to \mathscr{E}$ to the whole of $\mathscr{D}$.



It is worth pointing out that the functors J and G play quite different roles (see also Remark 4), which is why G/J is called the right Kan extension *of* G *along* J. The notation G/J is taken from relation algebra (see also Remark 5) and emphasises the algebraic properties of Kan extensions. (Mac Lane [27] writes $\mathsf{Ran}_\mathsf{J}\ \mathsf{G}$ for right Kan extensions and $\mathsf{Lan}_\mathsf{J}\ \mathsf{G}$ for left ones, a notation we do not use). Again, there are various ways to define the concept. The shortest is this:

The functor G/J is the (functor part of the) *right Kan extension of* G *along* J if and only if there is a bijection between the hom-sets

$$\mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/\mathsf{J}) \ , \tag{9}$$

that is natural in the functor $\mathsf{F} : \mathscr{D} \to \mathscr{E}$.

If we instantiate the bijection to $\mathsf{F} := \mathsf{G}/\mathsf{J}$, we obtain as the image of the identity $id : \mathscr{E}^{\mathscr{D}}(\mathsf{G}/\mathsf{J}, \mathsf{G}/\mathsf{J})$ a natural transformation $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G}/\mathsf{J}) \circ \mathsf{J}, \mathsf{G})$. The transformation eliminates a right Kan extension and is called the *unit* of the extension. An alternative definition of Kan extensions builds solely on the unit, which is an example of a universal arrow:

The *right Kan extension of* G *along* J consists of a functor written $\mathsf{G}/\mathsf{J} : \mathscr{D} \to \mathscr{E}$ and a natural transformation $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G}/\mathsf{J}) \circ \mathsf{J}, \mathsf{G})$. These two things have to satisfy the following *universal property*: for each functor $\mathsf{F} : \mathscr{D} \to \mathscr{E}$ and for each natural transformation $\alpha : \mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G})$ there exists a natural transformation $[\alpha] : \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/\mathsf{J})$ (pronounce "shift α") such that

$$\alpha = run \cdot \beta \circ \mathsf{J} \iff [\alpha] = \beta \ , \tag{10}$$

for all $\beta : \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/\mathsf{J})$. The equivalence witnesses the bijection (9) and expresses that there is a unique way to factor α into a composition of the form $run \cdot \beta \circ \mathsf{J}$.

A universal property such as (10) has three immediate consequences that are worth singling out. If we substitute the right-hand side into the left-hand side, we obtain the *computation law*:

$$\alpha = run \cdot [\alpha] \circ \mathsf{J} \ . \tag{11}$$

Instantiating β in (10) to the identity $id_{\mathsf{G}/\mathsf{J}}$ and substituting the left- into the right-hand side, yields the *reflection law*:

$$[run] = id \quad . \tag{12}$$

Finally, the *fusion law* allows us to fuse a shift with a natural transformation to form another shift:

$$[\alpha] \cdot \gamma = [\alpha \cdot \gamma \circ \mathsf{J}] \quad , \tag{13}$$

for all $\gamma : \mathscr{E}^{\mathscr{D}}(\hat{\mathsf{F}}, \check{\mathsf{F}})$. The fusion law states that shift is natural in the functor F. For the proof we reason

$$\begin{aligned}
& [\alpha] \cdot \gamma = [\alpha \cdot \gamma \circ \mathsf{J}] \\
\iff & \quad \{ \text{ universal property (10) } \} \\
& \alpha \cdot \gamma \circ \mathsf{J} = run \cdot ([\alpha] \cdot \gamma) \circ \mathsf{J} \\
\iff & \quad \{ -\circ \mathsf{J} \text{ functor (54d) } \} \\
& \alpha \cdot \gamma \circ \mathsf{J} = run \cdot [\alpha] \circ \mathsf{J} \cdot \gamma \circ \mathsf{J} \\
\iff & \quad \{ \text{ computation (11) } \} \\
& \alpha \cdot \gamma \circ \mathsf{J} = \alpha \cdot \gamma \circ \mathsf{J} \quad .
\end{aligned}$$

As all universal concepts, right Kan extensions are unique up to isomorphism. This is a consequence of naturality: let $\mathsf{G}/_1\mathsf{J}$ and $\mathsf{G}/_2\mathsf{J}$ be two Kan extensions. Since the string of isomorphisms

$$\mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/_1\mathsf{J}) \cong \mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/_2\mathsf{J})$$

is natural in F, the principle of indirect proof [15] implies that $\mathsf{G}/_1\mathsf{J} \cong \mathsf{G}/_2\mathsf{J}$. There is also a simple calculational proof, which nicely serves to illustrate the laws above. The isomorphism is given by

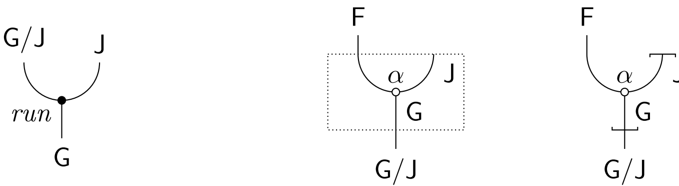
$$[run_1]_2 : \mathsf{G}/_1\mathsf{J} \cong \mathsf{G}/_2\mathsf{J} : [run_2]_1 \quad . \tag{14}$$

We show $[run_1]_2 \cdot [run_2]_1 = id$. The proof of the other half proceeds completely analogously.

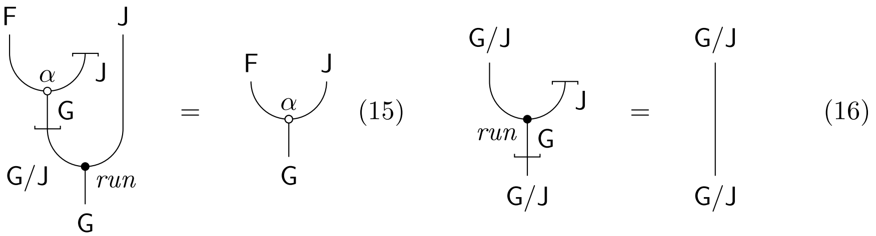
$$\begin{aligned}
& [run_1]_2 \cdot [run_2]_1 \\
= & \quad \{ \text{ fusion (13) } \} \\
& [run_1 \cdot [run_2]_1 \circ \mathsf{J}]_2 \\
= & \quad \{ \text{ computation (11) } \} \\
& [run_2]_2 \\
= & \quad \{ \text{ reflection (12) } \} \\
& id
\end{aligned}$$

Remark 4. If the Kan extension along J exists for every G, then $-/\mathsf{J}$ itself can be turned into a functor, so that the bijection $\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J},\mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F},\mathsf{G}/\mathsf{J})$ is also natural in G. In other words, we have an adjunction $-\circ\mathsf{J} \dashv -/\mathsf{J}$. If furthermore the adjunction $-\circ\mathsf{J} \dashv -/\mathsf{J}$ exists for every J—we have an adjunction with a parameter—then there is a unique way to turn $=/-$ into a higher-order bifunctor of type $(\mathscr{D}^{\mathscr{C}})^{\mathsf{op}} \times \mathscr{E}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{D}}$, so that the bijection is also natural in J [27, Th. IV.7.3, p102]. $\square$

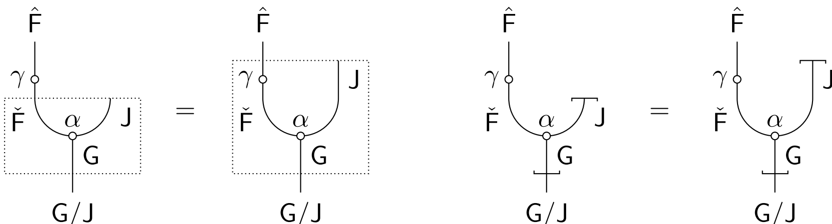
Turning to the two-dimensional notation, the unit *run* is drawn as a *solid* circle $\bullet$ (diagram on the left below). This convention allows us to omit the label *run* to avoid clutter. More interesting is the diagrammatic rendering of $[\alpha]$. My first impulse was to draw a dotted box around α, pruning J and relabelling G to G/J (diagram in the middle).



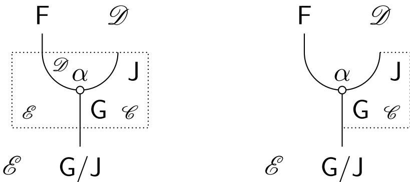
However, as we shall see in a moment, the diagram on the right is a better choice. The F branch is left untouched; the J and G branches are enclosed in square brackets (⌐ and ⊥). Computation (11) and reflection (12) are then rendered as follows.



Seen as a graph transformation, the computation law (15) allows us to replace the node labelled *run* by the sub-graph α. The reflection law (16) means that we can cut off a 'dead branch', a *run* node enclosed in brackets. The fusion law is the most interesting one—it shows the advantage of the bracket notation.



If we used the box notation, then fusion would allow us to shift γ in and out of the box. The bracket notation on the other hand incorporates the fusion law—recall that diagrams that differ only in the vertical position of natural transformations are identified—and is thus our preferred choice. (As an aside, the box notation is advantageous if we choose to label the regions: the category inside the box to the right of J and G is $\mathscr{C}$, whereas the category outside the box to the right of F and G/J is $\mathscr{D}$. A compromise is to draw only the lower right compartment of the box as shown on the right below.)



It is important to note that the computation law (15) contains universally quantified variables: it holds for all functors F and for all natural transformations α, the latter denoted by a *hollow* circle $\circ$ for emphasis. This is in contrast to all of the other two-dimensional laws we have seen before: the triangle identities and the monad laws involve only constants. When the computation law (15) is invoked, we have to substitute a subgraph for α and a bundle of strings for F. In particular, if F is replaced by Id, then the bundle is empty. The two-dimensional matching process is actually not too difficult: essentially one has to watch out for a solid circle ($\bullet$) to the right below of a closing bracket ($\llcorner\!\!\lrcorner$).

Finally, let us record that the diagrammatic reasoning is complete since computation, reflection and fusion imply the universal property (10). '$\Longleftarrow$': This implication amounts to the computation law (11). '$\Longrightarrow$': We reason

$$
\begin{aligned}
& [run \cdot \beta \circ \mathsf{J}] \\
= & \quad \{ \text{ fusion (13) } \} \\
& [run] \cdot \beta \\
= & \quad \{ \text{ reflection (12) } \} \\
& \beta \ .
\end{aligned}
$$

Remark 5. We can specialise Kan extensions to the preorder setting, if we equip a preorder with a monoidal structure: an associative operation that is monotone and that has a neutral element. Consider as an example the integers equipped with multiplication $*$. The bijection (9) then corresponds to the equivalence

$$m * k \leqslant n \iff m \leqslant n \div k \ ,$$

which specifies integer division $\div$ for $k > 0$. The equivalence uniquely defines division since the ordering relation is antisymmetric. The notation for Kan extensions is, in fact, inspired by this instance.

We obtain more interesting examples if we generalise monoids to categories. For instance, Kan extensions correspond to so-called *factors* in relation algebra, which are also known as residuals or weakest postspecifications [16].

$$F \cdot J \subseteq G \iff F \subseteq G \,/\, J \tag{17}$$

Informally, $G \,/\, J$ is the weakest (most general) postspecification that approximates G after specification J has been met. Again, the universal property uniquely defines $G \,/\, J$ since the subset relation is antisymmetric. The type of *run* corresponds to the computation law

$$(G \,/\, J) \cdot J \subseteq G \ . \tag{18}$$

(Kan extensions, integer quotients and factors are, in fact, instances of a more general 2-categorical concept. Actually, the development in this and in the following two sections can be readily generalised to 2-categories. Relation algebra is a simple instance of a 2-category where the vertical categories are preorders. A 'monoidal preorder' such as the integers with multiplication is an even simpler instance where the horizontal category is a monoid and *the* vertical category is a preorder.) □

4 Codensity Monad

The right Kan extension of J along J is a monad, $\mathsf{M} = \mathsf{J}/\mathsf{J}$, the so-called *codensity monad of* J. To motivate the definition of the monad operations, let us instantiate the Kan bijection (9) to $\mathsf{G} := \mathsf{J}$:

$$\mathscr{D}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J}, \mathsf{J}) \cong \mathscr{D}^{\mathscr{D}}(\mathsf{F}, \mathsf{M}) \ . \tag{19}$$

Recall that the bijection is natural in the functor F. For the return of the monad we set F to the identity functor, which suggests that return is just the transpose of the identity. The unit *run* of the Kan extension has type $\mathsf{M}\circ\mathsf{J} \dot{\to} \mathsf{J}$. To define the multiplication of the monad, we instantiate F to $\mathsf{M}\circ\mathsf{M}$, which leaves us with the task of providing a natural transformation of type $\mathsf{M}\circ\mathsf{M}\circ\mathsf{J} \dot{\to} \mathsf{J}$: the composition $run \cdot \mathsf{M}\circ run$ will do nicely. To summarise, the codensity monad of J is given by

$$\mathsf{M} = \mathsf{J}/\mathsf{J} \ , \tag{20a}$$

$$\eta = [id] \ , \tag{20b}$$

$$\mu = [run \cdot \mathsf{M}\circ run] \ . \tag{20c}$$

Of course, we have to show that the data satisfies the monad laws. As in the previous section, we provide two proofs, one using traditional notation and one using two-dimensional notation.

For the unit laws (6a)–(6b) we reason

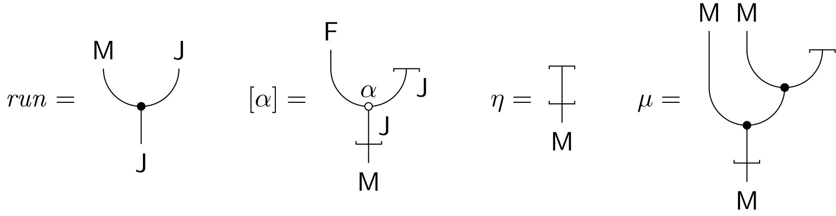
$$
\begin{aligned}
& \mu \cdot \eta \circ \mathsf{M} \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \cdot \eta \circ \mathsf{M} \\
= \;& \{ \text{ fusion (13) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \eta \circ \mathsf{M} \circ \mathsf{J}] \\
= \;& \{ \text{ interchange law (56) } \} \\
& [\mathit{run} \cdot \eta \circ \mathsf{J} \cdot \mathit{run}] \\
= \;& \{ \text{ definition of } \eta \text{ (20b) } \} \\
& [\mathit{run} \cdot [\mathit{id}] \circ \mathsf{J} \cdot \mathit{run}] \\
= \;& \{ \text{ computation (11) } \} \\
& [\mathit{run}] \\
= \;& \{ \text{ reflection (12) } \} \\
& \mathit{id} \;,
\end{aligned}
\qquad
\begin{aligned}
& \mu \cdot \mathsf{M} \circ \eta \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \cdot \mathsf{M} \circ \eta \\
= \;& \{ \text{ fusion (13) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mathsf{M} \circ \eta \circ \mathsf{J}] \\
= \;& \{ \; \mathsf{M} \circ - \text{ functor (54b) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ (\mathit{run} \cdot \eta \circ \mathsf{J})] \\
= \;& \{ \text{ definition of } \eta \text{ (20b) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ (\mathit{run} \cdot [\mathit{id}] \circ \mathsf{J})] \\
= \;& \{ \text{ computation (11) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{id}] \\
= \;& \{ \; \mathsf{M} \circ - \text{ functor (54a) } \} \\
& [\mathit{run}] \\
= \;& \{ \text{ reflection (12) } \} \\
& \mathit{id} \;.
\end{aligned}
$$

All of the basic identities are used: reflection (12), computation (11) and fusion (13).

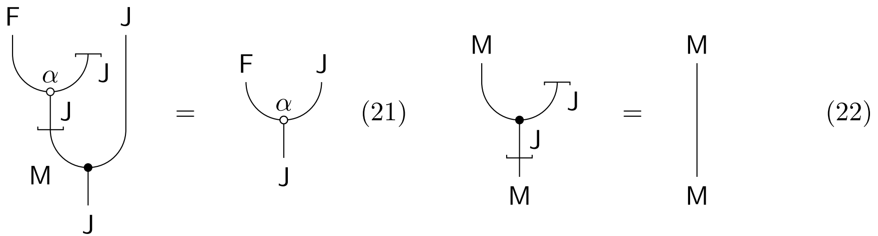
For the associative law (6c) we show that both sides of the equation simplify to $[\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mathsf{M} \circ \mathsf{M} \circ \mathit{run}]$, which merges three layers of effects:

$$
\begin{aligned}
& \mu \cdot \mu \circ \mathsf{M} \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \cdot \mu \circ \mathsf{M} \\
= \;& \{ \text{ fusion (13) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mu \circ \mathsf{M} \circ \mathsf{J}] \\
= \;& \{ \text{ interchange law (56) } \} \\
& [\mathit{run} \cdot \mu \circ \mathsf{J} \cdot \mathsf{M} \circ \mathsf{M} \circ \mathit{run}] \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \circ \mathsf{J} \cdot \mathsf{M} \circ \mathsf{M} \circ \mathit{run}] \\
= \;& \{ \text{ computation (11) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mathsf{M} \circ \mathsf{M} \circ \mathit{run}]
\end{aligned}
\qquad
\begin{aligned}
& \mu \cdot \mathsf{M} \circ \mu \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \cdot \mathsf{M} \circ \mu \\
= \;& \{ \text{ fusion (13) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mathsf{M} \circ \mu \circ \mathsf{J}] \\
= \;& \{ \; \mathsf{M} \circ - \text{ functor (54b) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ (\mathit{run} \cdot \mu \circ \mathsf{J})] \\
= \;& \{ \text{ definition of } \mu \text{ (20c) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ (\mathit{run} \cdot [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \circ \mathsf{J})] \\
= \;& \{ \text{ computation (11) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ (\mathit{run} \cdot \mathsf{M} \circ \mathit{run})] \\
= \;& \{ \; \mathsf{M} \circ - \text{ functor (54b) } \} \\
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run} \cdot \mathsf{M} \circ \mathsf{M} \circ \mathit{run}] \;.
\end{aligned}
$$

Turning to the second set of proofs, here are the two-dimensional counterparts of the natural transformations involved.

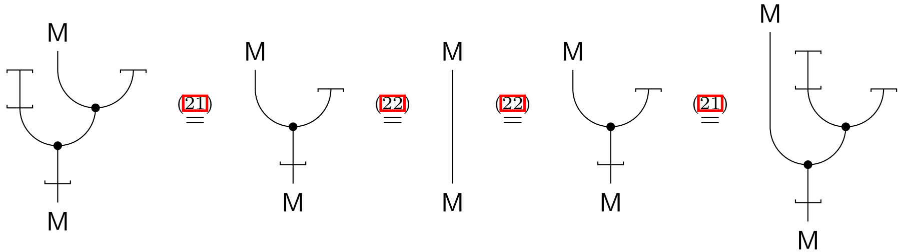


Diagrammatically, η indeed resembles a leaf, whereas μ is a nested fork with one branch cut off. For the calculations it is useful to specialise the computation law (15) and the reflection law (16) to $\mathsf{G} := \mathsf{J}$.



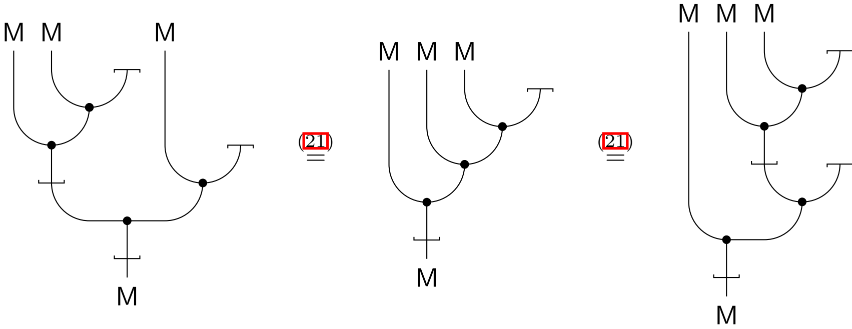
Recall that (21) holds for all functors F and for all natural transformations α.

Now, to show the unit laws we simply combine computation and reflection.



To follow the graph transformations involving (21), first identify the occurrence of a closing bracket (⌙) which leads to a solid circle (•) below. Then replace the solid circle (•) by the graph enclosed in ⌐ and ⌙, additionally removing the brackets. The instances of (21) above are in a sense extreme: F is in both cases instantiated to Id and α to id.

For the proof of associativity, we invoke the computation law twice.



Now, F is instantiated to $\mathsf{M}\circ\mathsf{M}$ and α to $run \cdot \mathsf{M}\circ run$. Again, the two-dimensional proofs carve out the essential steps.

Remark 6. Continuing Remark 5, let us specialise the above to relational algebra. The factor $J\,/\,J$ is a closure operator. The proofs correspond to the typing derivations of $\eta = [id]$ and $\mu = [run \cdot \mathsf{M}\circ run]$. Specifically, to prove $Id \subseteq J\,/\,J$ we appeal to the universal property (17) which leaves us with $Id \cdot J \subseteq J$. Likewise, to prove $(J\,/\,J) \cdot (J\,/\,J) \subseteq (J\,/\,J)$ if suffices to show that $(J\,/\,J) \cdot (J\,/\,J) \cdot J \subseteq J$. The obligation can be discharged using the computation law (18), twice. □

5 Absolute Kan Extension—Implementation

So far we have been concerned with general properties of right Kan extensions. Let us now turn our attention to the existence of extensions, which in computer-science terms is the implementation. A general result is this: if R is a right adjoint, then the right Kan extension along R exists for any functor G. To prove the result we take a short detour.

Adjunctions can be lifted to functor categories: if $\mathsf{L} \dashv \mathsf{R}$ is an adjunction then both $\mathsf{L}\circ- \dashv \mathsf{R}\circ-$ and $-\circ\mathsf{R} \dashv -\circ\mathsf{L}$ are adjunctions. (Recall that both $\mathsf{K}\circ-$ and $-\circ\mathsf{E}$ are functors, see Appendix A.) Since pre-composition is post-composition in the opposite category, the two statements are actually dual—note that L and R are flipped in the adjunction $-\circ\mathsf{R} \dashv -\circ\mathsf{L}$. For reasons to become clear in a moment, let us focus on pre-composition:

$$\text{if} \qquad \mathscr{L} \underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\perp}}}} \mathscr{R} \qquad \text{then} \qquad \mathscr{X}^{\mathscr{R}} \underset{-\circ\mathsf{L}}{\overset{-\circ\mathsf{R}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\perp}}}} \mathscr{X}^{\mathscr{L}} \;.$$

For the proof of this fact we establish the equivalence

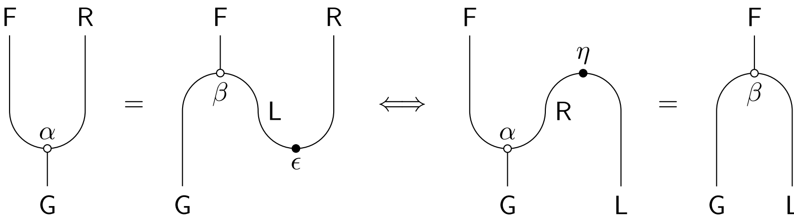
$$\alpha = \mathsf{G}\circ\epsilon \cdot \beta\circ\mathsf{R} \quad\iff\quad \alpha\circ\mathsf{L} \cdot \mathsf{F}\circ\eta = \beta \;, \tag{23}$$

for all functors F and G and for all natural transformations $\alpha : \mathsf{F}\circ\mathsf{R} \to \mathsf{G}$ and $\beta : \mathsf{F} \to \mathsf{G}\circ\mathsf{L}$. This equivalence amounts to (2) phrased in terms of the units.

We show the implication from left to right, the proof for the opposite direction proceeds completely analogously.

$$
\begin{aligned}
& (\mathsf{G}{\circ}\epsilon \cdot \beta{\circ}\mathsf{R}){\circ}\mathsf{L} \cdot \mathsf{F}{\circ}\eta \\
= & \quad \{ \ -{\circ}\mathsf{L} \text{ functor } (54d) \ \} \\
& \mathsf{G}{\circ}\epsilon{\circ}\mathsf{L} \cdot \beta{\circ}\mathsf{R}{\circ}\mathsf{L} \cdot \mathsf{F}{\circ}\eta \\
= & \quad \{ \text{ interchange law } (56) \ \} \\
& \mathsf{G}{\circ}\epsilon{\circ}\mathsf{L} \cdot \mathsf{G}{\circ}\mathsf{L}{\circ}\eta \cdot \beta \\
= & \quad \{ \ \mathsf{G}{\circ}- \text{ functor } (54b) \ \} \\
& \mathsf{G}{\circ}(\epsilon{\circ}\mathsf{L} \cdot \mathsf{L}{\circ}\eta) \cdot \beta \\
= & \quad \{ \text{ assumption: triangle identity } (3a) \ \} \\
& \mathsf{G}{\circ}id_{\mathsf{L}} \cdot \beta \\
= & \quad \{ \text{ identity } \} \\
& \beta
\end{aligned}
$$

If we write the equivalence (23) using two-dimensional notation,



then the proof becomes more perspicuous. For the left-to-right direction we focus on the left-hand side of the equivalence and put a cap on the R branches (on both sides of the equation) and then pull the L string straight down (on the right-hand side of the equation). Conversely, for the right-to-left direction we place a cup below the L branches and then pull the R string straight up.

Returning to the original question of existence of right Kan extensions, we have established

$$\mathscr{X}^{\mathscr{R}}(\mathsf{F}{\circ}\mathsf{R}, \mathsf{G}) \cong \mathscr{X}^{\mathscr{L}}(\mathsf{F}, \mathsf{G}{\circ}\mathsf{L}) \ , \tag{24}$$

which is an instance of the Kan bijection (9). In other words, $\mathsf{G}{\circ}\mathsf{L}$ is the right Kan extension of G along R. To bring the definition of unit and shift to light, we align the equivalence (23) with the universal property of right Kan extensions (10). We obtain

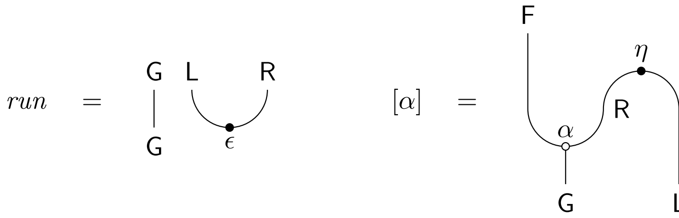
$$\mathsf{G}/\mathsf{R} = \mathsf{G}{\circ}\mathsf{L} \ , \tag{25a}$$

$$run = \mathsf{G}{\circ}\epsilon \ , \tag{25b}$$

$$[\alpha] = \alpha{\circ}\mathsf{L} \cdot \mathsf{F}{\circ}\eta \ . \tag{25c}$$

Since the bijection (24) is also natural in G, the right Kan extension along R exists for every G.

The unit *run* and $[\alpha]$ are rendered as follows.



To shift $\alpha : \mathsf{F}\circ\mathsf{R} \dot{\to} \mathsf{G}$ we simply put a cap on the rightmost branch labelled R.

Two special cases of (25a) are worth singling out: $\mathsf{L} = \mathsf{Id}/\mathsf{R}$ and $\mathsf{R}\circ\mathsf{L} = \mathsf{R}/\mathsf{R}$. Thus, the *left* adjoint can be expressed as a *right* Kan extension—$\mathsf{L} = \mathsf{Id}/\mathsf{R}$ is a so-called *absolute Kan extension* [27, p.249]. Very briefly, G/J is an absolute Kan extension if and only if it is preserved by any functor: $\mathsf{F}\circ(\mathsf{G}/\mathsf{J})$ and $\mathsf{F}\circ run$ is the Kan extension of $\mathsf{F}\circ\mathsf{G}$ along J. The associativity of horizontal composition implies that the Kan extension $\mathsf{G}\circ\mathsf{R}$ is indeed absolute. Moreover, the monad induced by the adjunction $\mathsf{L} \dashv \mathsf{R}$ coincides with the codensity monad of R:

$$(\mathsf{R}\circ\mathsf{L}, \eta, \mathsf{R}\circ\epsilon\circ\mathsf{L}) = (\mathsf{R}/\mathsf{R}, [id], [run \cdot \mathsf{M}\circ run]) \ . \tag{26}$$

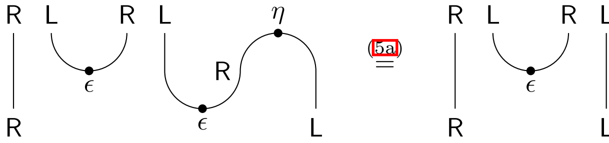
It remains to show that the two alternative definitions of unit and multiplication actually coincide. For the unit, the proof is straightforward.

$$\begin{aligned}
& [id] \\
= \ & \{ \text{ definition of } [-] \text{ (25c) } \} \\
& id\circ\mathsf{L} \cdot \mathsf{Id}\circ\eta \\
= \ & \{ \text{ identity (54c) and (55d) } \} \\
& \eta
\end{aligned}$$

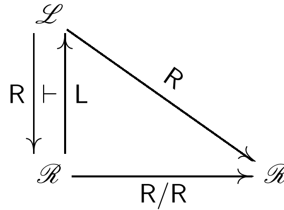
The proof for the multiplication rests on the triangle identity (3b).

$$\begin{aligned}
& [run \cdot \mathsf{M}\circ run] \\
= \ & \{ \text{ definition of } [-] \text{ (25c) } \} \\
& (run \cdot \mathsf{M}\circ run)\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{M}\circ\eta \\
= \ & \{ -\circ\mathsf{L} \text{ functor (54d) } \} \\
& run\circ\mathsf{L} \cdot \mathsf{M}\circ run\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{M}\circ\eta \\
= \ & \{ \text{ definition of } run \text{ (25b) } \} \\
& \mathsf{R}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{R}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{M}\circ\eta \\
= \ & \{ \text{ definition of } \mathsf{M} \text{ (25a) } \} \\
& \mathsf{R}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{R}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{R}\circ\mathsf{L}\circ\eta \\
= \ & \{ \mathsf{M}\circ\mathsf{R}\circ- \text{ functor (54b) } \} \\
& \mathsf{R}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{M}\circ\mathsf{R}\circ(\epsilon\circ\mathsf{L} \cdot \mathsf{L}\circ\eta) \\
= \ & \{ \text{ triangle identity (3b) } \} \\
& \mathsf{R}\circ\epsilon\circ\mathsf{L}
\end{aligned}$$

As before, the two-dimensional proofs are much shorter. For the unit, $\mathsf{F} := \mathsf{Id}$, there is nothing to do—putting a cap on id_{R} is just the cap. For the multiplication, $\mathsf{F} := \mathsf{R} \circ \mathsf{L} \circ \mathsf{R} \circ \mathsf{L}$, the proof collapses to a single application of the triangle identity—note that $run \cdot \mathsf{M} \circ run = \mathsf{R} \circ \epsilon \circ \epsilon$.



As an intermediate summary, we have shown that every monad is isomorphic to a codensity monad! This perhaps surprising result follows from the fact that every monad is induced by an adjoint pair of functors—this was shown independently by Kleisli [24] and Eilenberg and Moore [10].



Remark 7. To connect the development above to relation algebra, we first have to adopt the notion of an adjunction. The relations L and R are adjoint if and only if $L \cdot R \subseteq Id$ and $Id \subseteq R \cdot L$. In relation algebra this implies $R = L^{\circ}$, where $(-)^{\circ}$ is the converse operator. Thus, left adjoints are exactly the functions, simple and entire arrows (denoted by a lower-case letter below). The lifting of adjunctions to functor categories corresponds to the so-called *shunting rules* for functions [3].

$$l \cdot F \subseteq G \iff F \subseteq l^{\circ} \cdot G$$
$$F \cdot l^{\circ} \subseteq G \iff F \subseteq G \cdot l$$

Specifically, bijection (24) corresponds to the latter equivalence. Using the principle of indirect proof, it is then straightforward to show that $G \cdot l = G \mathbin{/} l^{\circ}$. In particular, $l = Id \mathbin{/} l^{\circ}$ and $l^{\circ} \cdot l = l^{\circ} \mathbin{/} l^{\circ}$. □

6 Kan Extension as an End—Implementation

Let us now turn to the heart of the matter. There is an elegant formula, the end formula, which describes right Kan extensions in terms of powers and ends [27, p.242].

$$(\mathsf{G}/\mathsf{J})\,(A : \mathscr{D}) = \forall Z : \mathscr{C} \,.\, \Pi\, \mathscr{D}(A, \mathsf{J}\,Z) \,.\, \mathsf{G}\,Z \tag{27}$$

The object on the right, which lives in $\mathscr{E}$, can be interpreted as a generalised continuation type. This can be seen more clearly if we write both the hom-set

$\mathscr{D}(A, \mathsf{J}\, Z)$ and the power $\Pi\, \mathscr{D}(A, \mathsf{J}\, Z) \mathrel{.} \mathsf{G}\, Z$ as function spaces: $(A \to \mathsf{J}\, Z) \to \mathsf{G}\, Z$. Informally, an element of $(\mathsf{G}/\mathsf{J})\, A$ is a polymorphic function that given a continuation of type $A \to \mathsf{J}\, Z$ yields an element of type $\mathsf{G}\, Z$ for all Z.

The purpose of this section is to prove the end formula and to derive the associated implementations of *run* and shift. To keep the paper sufficiently self-contained, we first introduce powers in Section 6.1 and ends in Section 6.2. (Some reviewers wondered why string diagrams do not appear beyond this point. The reason is simple: to be able to use string diagrams we have to know that the entities involved are functors and natural transformations. Here we set out to establish these properties. More pointedly, we use string diagrams if we wish to prove something against the *specification* of Kan extensions. Here, we aim to to prove an *implementation* correct.) The reader who is not interested in the details may wish to skip to Section 7, which investigates applications of the framework.

Remark 8. The end formula can be *derived* using a calculus of ends [27]—the calculus is introduced in [7]. The details are beyond the scope of this paper. □

6.1 Background: Power

Let $\mathscr{C}$ be a category. The *power* [27, p.70] of a set A : **Set** and an object $X : \mathscr{C}$ consists of an object written $\Pi\, A \mathrel{.} X : \mathscr{C}$ and a function $\pi : A \to \mathscr{C}(\Pi\, A \mathrel{.} X, X)$. These two things have to satisfy the following *universal property*: for each object $B : \mathscr{C}$ and for each function $g : A \to \mathscr{C}(B, X)$, there exists an arrow $(\bigtriangleup\, a \in A \mathrel{.} g(a)) : \mathscr{C}(B, \Pi\, A \mathrel{.} X)$ (pronounce "split g") such that

$$f = (\bigtriangleup\, \hat{a} \in A \mathrel{.} g(\hat{a})) \iff (\lambda\, \check{a} \in A \mathrel{.} \pi(\check{a}) \cdot f) = g \ , \tag{28}$$

for all $f : \mathscr{C}(B, \Pi\, A \mathrel{.} X)$.

The power $\Pi\, A \mathrel{.} X$ is an iterated product of the *object* X indexed by elements of the *set* A. The projection $\pi(a)$ is an arrow in $\mathscr{C}$ that selects the component whose index is a; the *arrow* $\bigtriangleup\, a \in A \mathrel{.} g(a)$ creates an iterated product, whose components are determined by the *function* g. A note on notation: the mediating arrow $\bigtriangleup\, a{\in}A \mathrel{.} g(a)$ is a binding construct as this allows us to leave the definition of g implicit. The notation also makes explicit that a ranges over a *set*. (The power $\Pi\, A \mathrel{.} X$ is sometimes written X^A, a notation we do not use.) Furthermore, we use λ for function abstraction and $-\,(=)$ for function application in **Set**.

As an example, for a two-element set, say, $A := \{0, 1\}$, the power $\Pi\, A \mathrel{.} X$ specialises to $X \times X$ with $\pi(0) = \mathit{outl}$, $\pi(1) = \mathit{outr}$ and $\bigtriangleup\, a \in A \mathrel{.} g(a) = g(0) \vartriangle g(1)$.

In **Set**, the power $\Pi\, A \mathrel{.} X$ is the set of all functions from A to X, that is, $\Pi\, A \mathrel{.} X = A \to X$. The projection π is just reverse function application: $\pi(a) = \lambda\, g : A \to X \mathrel{.} g(a)$; split is given by $\bigtriangleup\, a \in A \mathrel{.} g(a) = \lambda\, b \in B \mathrel{.} \lambda\, a \in A \mathrel{.} g(a)(b)$, that is, it simply swaps the two arguments of the curried function g.

The universal property (28) has three immediate consequences that are used repeatedly in the forthcoming calculations. If we substitute the left-hand side into the right-hand side, we obtain the *computation law*

$$\pi(\check{a}) \cdot (\bigtriangleup\, \hat{a} \in A \mathrel{.} g(\hat{a})) = g(\check{a}) \ , \tag{29}$$

for all $\check{a} \in A$. Instantiating f in (28) to the identity $id_{\Pi A \,.\, X}$ and substituting the right- into the left-hand side, yields the *reflection law*

$$id = (\triangle\, a \in A \,.\, \pi(a)) \ . \tag{30}$$

Finally, the *fusion law* allows us to fuse a split with an arrow to form another split (the proof is left as an exercise to the reader):

$$(\triangle\, a \in A \,.\, g(a)) \cdot k = (\triangle\, a \in A \,.\, g(a) \cdot k) \ . \tag{31}$$

The fusion law states that $\triangle : (A \to \mathscr{C}(B, X)) \to \mathscr{C}(B, \Pi\, A \,.\, X)$ is natural in B.

If the power $\Pi\, A \,.\, X$ exists for every set $A : \mathbf{Set}$, then there is a unique way to turn $\Pi - \,.\, X$ into a functor of type $\mathbf{Set} \to \mathscr{C}^{\mathsf{op}}$ so that $\pi : A \to \mathscr{C}(\Pi\, A \,.\, X, X)$ is natural in A. We calculate

$$\begin{aligned}
& \mathscr{C}(\Pi\, h \,.\, X, X) \cdot \pi = \pi \cdot h \\
\iff & \quad \{ \text{ definition of hom-functor } \mathscr{C}(-, X) \ \} \\
& (\lambda\, a \in A \,.\, \pi(a) \cdot (\Pi\, h \,.\, X)) = \pi \cdot h \\
\iff & \quad \{ \text{ universal property (28) } \} \\
& \Pi\, h \,.\, X = (\triangle\, a \in A \,.\, \pi(h(a))) \ ,
\end{aligned}$$

which suggests that the arrow part of $\Pi - \,.\, X$ is defined

$$\Pi\, h \,.\, X = (\triangle\, a \in A \,.\, \pi(h(a))) \ . \tag{32}$$

In other words, we have an adjoint situation: $\Pi - \,.\, X \dashv \mathscr{C}(-, X)$.

$$\mathscr{C}^{\mathsf{op}} \underset{\mathscr{C}(-,X)}{\overset{\Pi - \,.\, X}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{Set}$$

Since the hom-functor $\mathscr{C}(-, X)$ is contravariant, $\Pi - \,.\, X$ is contravariant, as well. Moreover, $\Pi - \,.\, X$ is a left adjoint, targeting the opposite category $\mathscr{C}^{\mathsf{op}}$.

$$\mathscr{C}^{\mathsf{op}}(\Pi\, A \,.\, X, B) \cong \mathbf{Set}(A, \mathscr{C}(B, X)) \tag{33}$$

The units of the adjunction are given by $\epsilon\, B = \triangle\, a \in \mathscr{C}(B, Y) \,.\, a$ and $\eta\, A = \lambda\, a \in A \,.\, \pi(a)$, that is, $\eta = \pi$. A contravariant adjoint functor such as $\Pi - \,.\, X$ gives rise to two monads: $\mathscr{C}(\Pi - \,.\, X, X)$ is a monad in $\mathbf{Set}$ and $\Pi\, \mathscr{C}(-, X) \,.\, X$ is a comonad in $\mathscr{C}^{\mathsf{op}}$ and consequently a monad in $\mathscr{C}$. Both monads can be seen as continuation monads. Let us spell out the details for the second monad: its unit is the counit (!) of the adjunction, for the multiplication we calculate

$$
\begin{aligned}
& \mu A \\
= \; & \{ \text{ definition of } \mu \text{ (8c) } \} \\
& ((\Pi - . X) \circ \eta \circ (\mathscr{C}(-, X)))\, A \\
= \; & \{ \text{ definition of horizontal composition } \circ \} \\
& \Pi\, \eta\, (\mathscr{C}(A, X)) . X \\
= \; & \{ \text{ definition of } \Pi\, h . X \text{ (32) } \} \\
& \textstyle\bigtriangleup\ a \in \mathscr{C}(A, X) . \pi(\eta\, (\mathscr{C}(A, X))\, (a)) \\
= \; & \{ \text{ definition of } \eta, \text{ see above } \} \\
& \textstyle\bigtriangleup\ a \in \mathscr{C}(A, X) . \pi(\pi(a)) \ .
\end{aligned}
$$

To summarise, the continuation monad $\mathsf{K} = (\Pi - . X) \circ (\mathscr{C}(-, X))$ is defined

$$
\begin{aligned}
\mathsf{K}\, A &= \Pi\, \mathscr{C}(A, X) . X \ , \\
\mathsf{K}\, f &= \bigtriangleup\ k . \pi(k \cdot f) \ , \\
\eta &= \bigtriangleup\ k . k \ , \\
\mu &= \bigtriangleup\ k . \pi(\pi(k)) \ .
\end{aligned}
$$

This is the abstract rendering of the monad C from the introduction with $\mathsf{M}\, A = X$ a constant functor. The correspondence can be seen more clearly, if we specialise the ambient category $\mathscr{C}$ to **Set**. We obtain

$$
\begin{aligned}
\mathsf{K}\, f &= \bigtriangleup\ k . \pi(k \cdot f) = \lambda\, m . \lambda\, k . \pi(k \cdot f)\, m = \lambda\, m . \lambda\, k . m\, (k \cdot f) \\
\eta &= \bigtriangleup\ k . k = \lambda\, a . \lambda\, k . k\, a \ , \\
\mu &= \bigtriangleup\ k . \pi(\pi(k)) = \lambda\, m . \lambda\, k . \pi(\pi(k))\, m = \lambda\, m . \lambda\, k . m\, (\lambda\, a . a\, k) \ ,
\end{aligned}
$$

which is exactly the Haskell code given in the introduction—recall that join and bind are related by $join\ m = m \gg\!= id$. For the full story—M an arbitrary functor, not necessarily constant—we need to model the universal quantifier in the type of C, which is what we do next in Section 6.2.

If the adjunction $\Pi - . X \dashv \mathscr{C}(-, X)$ exists for every $X : \mathscr{C}$, then there is a unique way to turn $\Pi - . = : \mathbf{Set}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{C}$ into a bifunctor so that (33) is also natural in X [27, Th. IV.7.3, p102]. The arrow part of the bifunctor is defined

$$
\Pi\, h . p = (\bigtriangleup\ a \in A . p \cdot \pi(h(a))) \ , \tag{34}
$$

for all $h : A \to B$ and for all $p : \mathscr{C}(X, Y)$.

6.2 Background: End

Ends capture polymorphic functions as objects. Before we can define the notion formally, we first need to introduce the concept of a dinatural transformation [27, p.218].

Let $\mathsf{S}, \mathsf{T} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ be two parallel functors. A *dinatural transformation* $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ is a collection of arrows: for each object $A : \mathscr{C}$ there is an arrow $\delta\, A : \mathsf{S}\,(A, A) \to \mathsf{T}\,(A, A)$, such that

$$\mathsf{T}\,(id, h) \cdot \delta\, \hat{A} \cdot \mathsf{S}\,(h, id) = \mathsf{T}\,(h, id) \cdot \delta\, \check{A} \cdot \mathsf{S}\,(id, h) \ , \tag{35}$$

for all $h : \mathscr{C}(\hat{A}, \check{A})$. A component of a dinatural transformation instantiates both arguments of the bifunctors to the same object, which explains the term dinaturality, a contraction of the more unwieldy *diagonal naturality*.

A natural transformation $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ can be turned into a dinatural transformation $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ by setting $\delta\, A = \alpha\,(A, A)$. There is an identity dinatural transformation, but, unfortunately, dinatural transformations do not compose in general. However, they are closed under composition with a natural transformation: $(\alpha \cdot \delta)\, A = \alpha\,(A, A) \cdot \delta\, A$ where $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ and $\alpha : \mathsf{T} \dot{\to} \mathsf{U}$, and $(\delta \cdot \alpha)\, A = \delta\, A \cdot \alpha\,(A, A)$ where $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ and $\delta : \mathsf{T} \ddot{\to} \mathsf{U}$.

A dinatural transformation $\omega : \Delta A \ddot{\to} \mathsf{T}$ from a constant functor, $\Delta A\, X = A$, is called a *wedge*. For wedges, the dinaturality condition (35) simplifies to

$$\mathsf{T}\,(id, h) \cdot \omega\, \hat{A} = \mathsf{T}\,(h, id) \cdot \omega\, \check{A} \ , \tag{36}$$

for all $h : \mathscr{C}(\hat{A}, \check{A})$.

Now, the *end* [27, p.222] of a functor $\mathsf{T} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ consists of an object written $\mathsf{End}\,\mathsf{T} : \mathscr{D}$ and a wedge $App : \Delta(\mathsf{End}\,\mathsf{T}) \ddot{\to} \mathsf{T}$. These two things have to satisfy the following universal property: for each object A and for each wedge $\omega : \Delta A \ddot{\to} \mathsf{T}$, there exists an arrow $\Lambda\, \omega : \mathscr{D}(A, \mathsf{End}\,\mathsf{T})$ such that

$$\omega = App \cdot \Delta g \iff \Lambda\, \omega = g \ , \tag{37}$$

for all $g : \mathscr{D}(A, \mathsf{End}\,\mathsf{T})$. Note that on the left a dinatural transformation, App, is composed with a natural transformation, Δg defined $\Delta g\, X = g$.

The end $\mathsf{End}\,\mathsf{T}$ is also written $\forall X : \mathscr{C}\ .\ \mathsf{T}\,(X, X)$, which supports the intuition that an end is a polymorphic type. The wedge App models type application: the component $App\, A : \mathsf{End}\,\mathsf{T} \to \mathsf{T}\,(A, A)$ instantiates a given end to the object A. Accordingly, Λ is type abstraction—to emphasise this point we also write Λ using a binder: $\Lambda\, Z\ .\ \omega\, Z$ serves as alternative notation for $\Lambda\, \omega$.

The universal property (37) has the usual three consequences. If we substitute the right-hand side into the left-hand side, we obtain the *computation law*:

$$\omega = App \cdot \Delta(\Lambda\, \omega) \ . \tag{38}$$

Instantiating g in (37) to the identity $id_{\mathsf{End}\,\mathsf{T}}$ and substituting the left- into the right-hand side, yields the *reflection law*:

$$\Lambda\, App = id \ . \tag{39}$$

Finally, the *fusion law* allows us to fuse a 'type abstraction' with an arrow to form another 'type abstraction' (the proof is left as an exercise to the reader):

$$\Lambda\, \omega \cdot h = \Lambda\,(\omega \cdot \Delta h) \ . \tag{40}$$

If all the necessary ends exist, we can turn End into a higher-order functor of type $\mathscr{D}^{\mathscr{C}^{\mathsf{op}} \times \mathscr{C}} \to \mathscr{D}$. The object part maps a bifunctor to its end; the arrow part maps a *natural transformation* $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ to an arrow $\mathsf{End}\,\alpha : \mathscr{D}(\mathsf{End}\,\mathsf{S}, \mathsf{End}\,\mathsf{T})$. There is a unique way to define this arrow so that type application $App : \Delta(\mathsf{End}\,\mathsf{T}) \ddot{\to} \mathsf{T}$ is natural in T:

$$\alpha \cdot App = App \cdot \Delta(\mathsf{End}\,\alpha) \ . \tag{41}$$

We simply appeal to the universal property (37)

$$\alpha \cdot App = App \cdot \Delta(\mathsf{End}\,\alpha) \iff \mathsf{End}\,\alpha = \Lambda\,(\alpha \cdot App) \ ,$$

which suggests that the arrow part of End is defined

$$\mathsf{End}\,\alpha = \Lambda\,(\alpha \cdot App) \ . \tag{42}$$

The proof that End indeed preserves identity and composition is again left as an exercise to the reader.

6.3 End Formula

Equipped with the new vocabulary we can now scrutinise the end formula (27), $(\mathsf{G}/\mathsf{J})\,(A : \mathscr{D}) = \forall Z : \mathscr{C} \ . \ \Pi\ \mathscr{D}\,(A, \mathsf{J}\,Z) \ . \ \mathsf{G}\,Z$, more closely. This definition is shorthand for $\mathsf{G}/\mathsf{J} = \mathsf{End} \circ \mathsf{T}$ where

$$\mathsf{T}\,A\,(Z^-, Z^+) = \Pi\ \mathscr{D}\,(A, \mathsf{J}\,Z^-) \ . \ \mathsf{G}\,Z^+ \ , \tag{43}$$

is a higher-order functor of type $\mathscr{D} \to \mathscr{D}^{\mathscr{C}^{\mathsf{op}} \times \mathscr{C}}$. (As an aside, Z^- and Z^+ are identifiers ranging over objects. The superscripts indicate variance: Z^- is an object of $\mathscr{C}^{\mathsf{op}}$ and Z^+ is an object of $\mathscr{C}$.) Clearly, G/J thus defined is a functor. It is useful to explicate its action on arrows.

$$\begin{aligned}
& \forall Z : \mathscr{C} \ . \ \textstyle\prod \mathscr{D}(f, \mathsf{J}\,Z) \ . \ \mathsf{G}\,Z \\
= \ & \quad \{ \text{ definition of } \mathsf{End} \ (42) \ \} \\
& \Lambda\,Z : \mathscr{C} \ . \ (\textstyle\prod \mathscr{D}(f, \mathsf{J}\,Z) \ . \ \mathsf{G}\,Z) \cdot App\,Z \\
= \ & \quad \{ \text{ definition of } \textstyle\prod - \ . \ Y \quad (32) \ \} \\
& \Lambda\,Z : \mathscr{C} \ . \ (\triangle\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ . \ \pi(c \cdot f)) \cdot App\,Z \\
= \ & \quad \{ \text{ fusion } (31) \ \} \\
& \Lambda\,Z : \mathscr{C} \ . \ \triangle\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ . \ \pi(c \cdot f) \cdot App\,Z
\end{aligned}$$

Let us record the definition.

$$(\mathsf{G}/\mathsf{J})\,f = \Lambda\,Z : \mathscr{C} \ . \ \triangle\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ . \ \pi(c \cdot f) \cdot App\,Z \tag{44}$$

The unit $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G}/\mathsf{J}) \circ \mathsf{J}, \mathsf{G})$ of the right Kan extension is defined

$$run\,A = \pi(id_{\mathsf{J}\,A}) \cdot App\,A \ . \tag{45}$$

The end is instantiated to A and then the component whose index is the identity $id_{\mathsf{J}\,A}$ is selected. A number of proof obligations arise. We have to show that run is a natural transformation and that it satisfies the universal property (10) of right Kan extensions. For the calculations, the following property of run, a simple program optimisation, proves to be useful. Let $f : \mathscr{D}(A, \mathsf{J}\,B)$, then

$$run\,B \cdot (\mathsf{G}/\mathsf{J})\,f = \pi(f) \cdot App\,B \quad . \tag{46}$$

We reason

$$\begin{aligned}
& run\,B \cdot (\mathsf{G}/\mathsf{J})\,f \\
= \; & \{ \text{ definition of } run \text{ (45) } \} \\
& \pi(id_{\mathsf{J}\,B}) \cdot App\,B \cdot (\mathsf{G}/\mathsf{J})\,f \\
= \; & \{ \text{ definition of } \mathsf{G}/\mathsf{J} \text{ (44) } \} \\
& \pi(id_{\mathsf{J}\,B}) \cdot App\,B \cdot (\Lambda\, Z : \mathscr{C} \;.\; \textstyle\bigtriangleup\; c \in \mathscr{D}(\mathsf{J}\,B, \mathsf{J}\,Z) \;.\; \pi(c \cdot f) \cdot App\,Z) \\
= \; & \{ \text{ computation (38) } \} \\
& \pi(id_{\mathsf{J}\,B}) \cdot (\textstyle\bigtriangleup\; c \in \mathscr{D}(\mathsf{J}\,B, \mathsf{J}\,B) \;.\; \pi(c \cdot f) \cdot App\,B) \\
= \; & \{ \text{ computation (29) } \} \\
& \pi(f) \cdot App\,B \quad .
\end{aligned}$$

The naturality of run follows from the dinaturality of App.

$$\begin{aligned}
& run\,\check{A} \cdot (\mathsf{G}/\mathsf{J})\,(\mathsf{J}\,h) \\
= \; & \{ \text{ property of } run \text{ (46) } \} \\
& \pi(\mathsf{J}\,h) \cdot App\,\check{A} \\
= \; & \{ \; App \text{ is dinatural, see below } \} \\
& \mathsf{G}\,h \cdot \pi(id_{\mathsf{J}\,\hat{A}}) \cdot App\,\hat{A} \\
= \; & \{ \text{ definition of } run \text{ (45) } \} \\
& \mathsf{G}\,h \cdot run\,\hat{A}
\end{aligned}$$

To comprehend the second step let us instantiate the dinaturality condition (36) to $App : \Delta(\mathsf{End}\,(\mathsf{T}\,A)) \ddot{\to} \mathsf{T}\,A$. Let $h : \mathscr{C}(\hat{Z}, \check{Z})$, then

$$\begin{aligned}
& \mathsf{T}\,A\,(id, h) \cdot App\,\hat{Z} = \mathsf{T}\,A\,(h, id) \cdot App\,\check{Z} \\
\iff \; & \{ \text{ definition of } \mathsf{T} \text{ (43) } \} \\
& (\textstyle\bigtriangleup\; a \in A \;.\; \mathsf{G}\,h \cdot \pi(a)) \cdot App\,\hat{Z} = (\textstyle\bigtriangleup\; a \in A \;.\; \pi(\mathsf{J}\,h \cdot a)) \cdot App\,\check{Z} \\
\iff \; & \{ \text{ fusion (31) } \} \\
& (\textstyle\bigtriangleup\; a \in A \;.\; \mathsf{G}\,h \cdot \pi(a) \cdot App\,\hat{Z}) = (\textstyle\bigtriangleup\; a \in A \;.\; \pi(\mathsf{J}\,h \cdot a) \cdot App\,\check{Z}) \\
\implies \; & \{ \text{ left-compose with } \pi(id_{\mathsf{J}\,A}) \text{ and computation (29) } \} \\
& \mathsf{G}\,h \cdot \pi(id_{\mathsf{J}\,A}) \cdot App\,\hat{Z} = \pi(\mathsf{J}\,h) \cdot App\,\check{Z} \quad .
\end{aligned}$$

Next we show that *run* satisfies the universal property (10) of right Kan extensions. Along the way we derive the definition of shift. Let $\alpha : \mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G})$ and $\beta : \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/\mathsf{J})$, then

$$
\begin{aligned}
& \alpha = \mathit{run} \cdot \beta \circ \mathsf{J} \\
\iff & \{ \text{ equality of natural transformations } \} \\
& \forall A : \mathscr{C} \,.\, \alpha\, A = \mathit{run}\, A \cdot \beta\,(\mathsf{J}\, A) \\
\iff & \{ \text{ Yoneda Lemma: } \mathscr{E}(\mathsf{F}\,(\mathsf{J}\, A), \mathsf{G}\, A) \cong \mathscr{D}(-, \mathsf{J}\, A) \dot{\to} \mathscr{E}(\mathsf{F}\,-, \mathsf{G}\, A) \ \} \\
& \forall A : \mathscr{C}, B : \mathscr{D} \,.\, \forall c : \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c = \mathit{run}\, A \cdot \beta\,(\mathsf{J}\, A) \cdot \mathsf{F}\, c \\
\iff & \{ \ \beta \text{ is natural } \} \\
& \forall A : \mathscr{C}, B : \mathscr{D} \,.\, \forall c : \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c = \mathit{run}\, A \cdot (\mathsf{G}/\mathsf{J})\, c \cdot \beta\, B \\
\iff & \{ \text{ property of } \mathit{run} \text{ (46) } \} \\
& \forall A : \mathscr{C}, B : \mathscr{D} \,.\, \forall c : \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c = \pi(c) \cdot \mathit{App}\, A \cdot \beta\, B \\
\iff & \{ \text{ universal property of powers (28) } \} \\
& \forall A : \mathscr{C}, B : \mathscr{D} \,.\, (\textstyle\bigtriangleup\, c \in \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c) = \mathit{App}\, A \cdot \beta\, B \\
\iff & \{ \text{ universal property of ends (37) } \} \\
& \forall B : \mathscr{D} \,.\, (\Lambda\, A : \mathscr{C} \,.\, \textstyle\bigtriangleup\, c \in \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c) = \beta\, B \\
\iff & \{ \text{ define } [\alpha]\, B = \Lambda\, A : \mathscr{C} \,.\, \textstyle\bigtriangleup\, c \in \mathscr{D}(B, \mathsf{J}\, A) \,.\, \alpha\, A \cdot \mathsf{F}\, c \ \} \\
& \forall B : \mathscr{D} \,.\, [\alpha]\, B = \beta\, B \\
\iff & \{ \text{ equality of natural transformations } \} \\
& [\alpha] = \beta \ .
\end{aligned}
$$

Each of the steps is fairly compelling, except perhaps the second one, which rests on the Yoneda Lemma [27, p.61]. Its purpose is to introduce the functor application $\mathsf{F}\, c$ so that the naturality of β can be utilised. Thus, shift is defined

$$[\alpha]\, A = \Lambda\, Z : \mathscr{C} \,.\, \textstyle\bigtriangleup\, c \in \mathscr{D}(A, \mathsf{J}\, Z) \,.\, \alpha\, Z \cdot \mathsf{F}\, c \ . \tag{47}$$

Two remarks are in order. First, the body of the type abstraction, that is $\bigtriangleup\, c \in \mathscr{D}(A, \mathsf{J}\, Z) \,.\, \alpha\, Z \cdot \mathsf{F}\, c$ is a *dinatural* transformation because it equals $\mathit{App}\, Z \cdot \beta\, A = (\mathit{App} \cdot \Delta(\beta\, A))\, Z$—see derivation above—which is dinatural in Z. Second, $[\alpha]$ itself is a *natural* transformation because β is one by assumption.

Let us now turn our attention to the implementation of the codensity monad of a functor J. Combining (20a) with the end formula (27) gives

$$\mathsf{C}\, A = \forall Z : \mathscr{C} \,.\, \Pi\, \mathscr{D}\,(A, \mathsf{J}\, Z) \,.\, \mathsf{J}\, Z \ .$$

The instance of the end formula on the right is commonly regarded as *the* codensity monad. This view is partially justified since the end formula provides a general implementation of right Kan extensions, subject to the existence of the necessary powers and ends. It confuses, however, an implementation with an abstract concept. (This confusion is not uncommon in computer science.) The codensity monad is also regarded as the 'real' continuation monad. To see the

relation to continuation-passing style, let us unroll the definitions of return and join. For $\eta = [id]$ (20b), we obtain

$$
\begin{aligned}
& [id]\,A \\
= \; & \{ \text{ definition of } [-] \text{ (47) } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, id\; Z \cdot \mathsf{Id}\, k \\
= \; & \{ \text{ identity } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, k \;\;,
\end{aligned}
$$

and for $\mu = [run \cdot \mathsf{M} \circ run]$ (20c) we calculate

$$
\begin{aligned}
& [run \cdot \mathsf{M} \circ run]\, A \\
= \; & \{ \text{ definition of } [-] \text{ (47) } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, run\; Z \cdot \mathsf{M}\,(run\; Z) \cdot \mathsf{M}\,(\mathsf{M}\, k) \\
= \; & \{ \; \mathsf{M} \text{ functor } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, run\; Z \cdot \mathsf{M}\,(run\; Z \cdot \mathsf{M}\, k) \\
= \; & \{ \text{ property of } run \text{ (46) } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, run\; Z \cdot \mathsf{M}\,(\pi(k) \cdot App\; Z) \\
= \; & \{ \text{ property of } run \text{ (46) } \} \\
& \Lambda\, Z : \mathscr{C} \,.\, \triangle\, k \in \mathscr{D}(A, \mathsf{J}\,Z) \,.\, \pi(\pi(k) \cdot App\; Z) \cdot App\; Z \;\;.
\end{aligned}
$$

To summarise, the codensity monad implemented in terms of powers and ends is given by

$$
\begin{aligned}
\mathsf{C}\, A &= \forall Z \,.\, \Pi\; \mathscr{D}\,(A, \mathsf{J}\,Z) \,.\, \mathsf{J}\,Z \;\;, \\
\mathsf{C} f &= \Lambda\, Z \,.\, \triangle\, k \,.\, \pi(k \cdot f) \cdot App\; Z \;\;, \\
\eta &= \Lambda\, Z \,.\, \triangle\, k \,.\, k \;\;, \\
\mu &= \Lambda\, Z \,.\, \triangle\, k \,.\, \pi(\pi(k) \cdot App\; Z) \cdot App\; Z \;\;,
\end{aligned}
$$

which is similar to the continuation monad K of Section 6.1, except for occurrences of type abstraction and type application. This is the abstract rendering of the Haskell code for C from the introduction—note that in Haskell type abstraction and type application are implicit.

7 Examples

Let $\mathsf{L} \dashv \mathsf{R}$ be an adjunction.

$$
\mathscr{C} \underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{D}
$$

We have encountered two implementations of the codensity monad of R: the standard implementation $\mathsf{R} \circ \mathsf{L}$ and the implementation induced by the end formula (27). Since right Kan extensions are unique up to isomorphism (see Section 3), we have

$$\mathsf{R} \circ \mathsf{L} = \mathsf{R} /_1 \mathsf{R} \cong \mathsf{R} /_2 \mathsf{R} = \lambda\, A : \mathscr{D} \,.\, \forall Z : \mathscr{C} \,.\, \Pi\; \mathscr{D}(A, \mathsf{R}\, Z) \,.\, \mathsf{R}\, Z \;\;. \tag{48}$$

The isomorphisms are $[run_1]_2$ and $[run_2]_1$ (again see Section 3).

In the following sections we look at a few instances of this isomorphism. The list is by no means exhaustive, but it is indicative of the wide range of applications—the reader is invited to explore further adjunctions.

7.1 Identity Monad

The simplest example of an adjunction is $\mathsf{Id} \dashv \mathsf{Id}$, which induces the identity monad.

$$\mathscr{C} \underset{\mathsf{Id}}{\overset{\mathsf{Id}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C}$$

The units of the adjunction are identities: $\epsilon_1 = id$ and $\eta_1 = id$. Furthermore, run and shift are defined $run_1 = id$ and $[\alpha]_1 = \alpha$.

Instantiating (48) to $\mathsf{Id} \dashv \mathsf{Id}$ yields

$$\mathsf{Id} \cong \lambda\, A : \mathscr{D} \,.\, \forall Z : \mathscr{C} \,.\, \Pi\; \mathscr{D}(A, Z) \,.\, Z \;\;, \tag{49}$$

which generalises one of the main examples in Wadler's famous paper "Theorems for free!" [34]. Wadler shows $A \cong \forall Z : \mathscr{C} \,.\, \Pi\; \mathscr{D}(A, Z) \,.\, Z$. Equation (49) tells us that this isomorphism is also natural in A. The isomorphisms $[run_1]_2$ and $[run_2]_1$ specialise to

$$[run_1]_2 = [id]_2 = \eta_2 \;\;,$$
$$[run_2]_1 = run_2 \;\;.$$

One direction is given by the unit of the 'continuation monad'; for the other direction we simply run the continuation monad.

7.2 State Monad

The Haskell programmer's favourite adjunction is currying: $- \times X \dashv (-)^X$.

$$\mathscr{C} \underset{(-)^X}{\overset{- \times X}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C}$$

In **Set**, a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument. In general, we are seeking the right adjoint of pairing with a fixed object $X : \mathscr{C}$.

$$\mathscr{C}(A \times X, B) \cong \mathscr{C}(A, B^X) \ .$$

The object B^X is called the *exponential* of X and B. That this adjunction exists is one of the requirements for *cartesian closure* [25]. In **Set**, B^X is the set of total functions from X to B.

The curry adjunction induces the state monad, where $X : \mathscr{C}$ serves as the state space. This instance of (48) reads

$$(- \times X)^X \cong \lambda\, A : \mathscr{C}\ .\ \forall Z : \mathscr{C}\ .\ \Pi\ \mathscr{C}\,(A, Z^X)\ .\ Z^X \tag{50}$$

On the left we have the standard implementation of the state monad using state transformers. The end formula yields an implementation in continuation-passing style. The continuation of type $\mathscr{C}(A, Z^X) \cong \mathscr{C}(A \times X, Z)$ takes an element of the return type A and an element of the state type X, the final state. The initial state is passed to the exponential in the body of the power.

The Haskell rendering of the two implementations is fairly straightforward. Here is the standard implementation

$$\mathbf{newtype}\,\mathsf{State}_1\ a = \mathit{In}\,\{\,\mathit{out} : X \to (a, X)\,\}\ ,$$

and here is the CPS-based one

$$\mathbf{newtype}\,\mathsf{State}_2\ a = \mathit{CPS}\,\{\,\mathit{call} : \forall\, z\ .\ (a \to (X \to z)) \to (X \to z)\,\}\ .$$

7.3 Free Monad of a Functor

One of the most important adjunctions for the algebra of programming is $\mathsf{Free} \dashv \mathsf{U}$, which induces the so-called free monad of a functor. This adjunction makes a particularly interesting example as it involves two different categories. Here are the gory details:

Let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be an endofunctor. An F-*algebra* is a pair $\langle A,\ a\rangle$ consisting of an object $A : \mathscr{C}$ (the carrier of the algebra) and an arrow $a : \mathscr{C}(\mathsf{F}\,A, A)$ (the action of the algebra). An F-*algebra homomorphism* between algebras $\langle A,\ a\rangle$ and $\langle B,\ b\rangle$ is an arrow $h : \mathscr{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\,h$. Identity is an F-algebra homomorphism and homomorphisms compose. Thus, the data defines a category, called $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$.

The category $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$ has more structure than $\mathscr{C}$. The forgetful or underlying functor $\mathsf{U} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ forgets about the additional structure: $\mathsf{U}\,\langle A,\ a\rangle = A$ and $\mathsf{U}\,h = h$. While the definition of the forgetful functor is deceptively simple, it gives rise to an interesting concept via an adjunction.

$$\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \underset{\mathsf{U}}{\overset{\mathsf{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C}$$

The left adjoint Free maps an object A to the *free* F-*algebra* over A, written $\langle \mathsf{F}^* A, \mathit{com} \rangle$. In **Set**, the elements of $\mathsf{F}^* A$ are terms built from constructors determined by F and variables drawn from A. Think of the functor F as a grammar describing the syntax of a language. The action $\mathit{com} : \mathscr{C}(\mathsf{F}(\mathsf{F}^* A), \mathsf{F}^* A)$ constructs a *com*posite term from an F-structure of subterms. There is also an operation $\mathit{var} : \mathscr{C}(A, \mathsf{F}^* A)$ for embedding a *var*iable into a term. This operation is a further example of a universal arrow: for each F-algebra B and for each arrow $g : \mathscr{C}(A, \mathsf{U}\, B)$ there exists an F-algebra homomorphism $\mathit{eval}\; g : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\, A, B)$ (pronounce "*eval*uate with g") such that

$$f = \mathit{eval}\; g \iff \mathsf{U} f \cdot \mathit{var} = g \;, \tag{51}$$

for all $f : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\, A, B)$. In words, the meaning of a term is uniquely determined by the meaning of the variables. The fact that $\mathit{eval}\; g$ is a homomorphism entails that the meaning function is compositional: the meaning of a composite term is defined in terms of the meanings of its constituent parts.

The adjunction $\mathsf{Free} \dashv \mathsf{U}$ induces the free monad F^* of the functor F. The isomorphism (48) gives two implementations of the free monad.

$$\mathsf{F}^* \cong \lambda\, A : \mathscr{C} \,.\, \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \Pi\; \mathscr{C}\,(A, \mathsf{U}\, Z) \,.\, \mathsf{U}\, Z \tag{52}$$

The standard implementation represents terms as finite trees: the free algebra $\mathsf{F}^* A$ is isomorphic to $\mu \mathsf{F}_A$ where $\mathsf{F}_A\, X = A + \mathsf{F}\, X$ [1]. The implementation based on Kan extensions can be seen as the *Church representation* [26,5] of terms. Note that the variable Z ranges over F-algebras. The continuation of type $\mathscr{C}(A, \mathsf{U}\, Z)$ specifies the meaning of variables. Given such a meaning function a term can be evaluated to an element of type $\mathsf{U}\, Z$. (It is debatable whether the term 'continuation' makes sense here—U/U is certainly a *generalised* continuation type.)

It is instructive to consider how the definitions translate into Haskell. The implementation using trees is straightforward: the constructors *var* and *com* are turned into constructors of a datatype (we are building on the isomorphism $\mathsf{F}^* A \cong \mu \mathsf{F}_A$ here).

$$\mathbf{data}\; \mathsf{Term}_1\; a = \mathit{Var}\; a \mid \mathit{Com}\,(\mathsf{F}\,(\mathsf{Term}_1\; a))$$

The Church representation is more interesting as we have to deal with the question of how to model the variable Z, which ranges over F-algebras. One way to achieve this is to constrain Z by a class context.

$$\begin{array}{l}\mathbf{class}\; \mathit{Algebra}\; a\; \mathbf{where}\\ \quad \mathit{algebra} : \mathsf{F}\; a \to a\end{array}$$

The Church representation then reads

$$\mathbf{newtype}\; \mathsf{Term}_2\; a = \mathit{Abstr}\,\{\, \mathit{apply} : \forall\, z \,.\, (\mathit{Algebra}\; z) \Rightarrow (a \to z) \to z \,\} \;.$$

The two implementations represent two extremes. For terms as trees, constructing a term is easy, whereas evaluating a term is hard work. For the Church

representation, it is the other way round. Evaluating a term is a breeze as, in a sense, a term *is* an evaluator. Constructing a term is slightly harder, but not that much. The reader is invited to spell out the details.

Initial and free algebras are closely related (see above). The so-called extended initial algebra semantics [11] is a simple consequence of (52).

$$
\begin{aligned}
& \mu\mathsf{F} \\
\cong\ & \{ \text{ left adjoint preserve colimits: } \mu\mathsf{F} = \mathsf{U}\,0 \cong \mathsf{U}\,(\mathsf{Free}\,0) = \mathsf{F}^*\,0 \ \} \\
& \mathsf{F}^*\,0 \\
\cong\ & \{ \ (52) \ \} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \sqcap\, \mathscr{C}\,(0, \mathsf{U}\,Z) \,.\, \mathsf{U}\,Z \\
\cong\ & \{ \ 0 \text{ is initial } \} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \sqcap\, 1 \,.\, \mathsf{U}\,Z \\
\cong\ & \{ \ \sqcap\, 1 \,.\, X \cong X \ \} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \mathsf{U}\,Z \\
\cong\ & \{ \text{ relation between ends and limits [27, Prop. IX.5.3] } \} \\
& \mathsf{Lim}\,\mathsf{U}
\end{aligned}
$$

The calculation shows that a *colimit*, the initial algebra $\mu\mathsf{F}$, is isomorphic to a *limit*, the limit of the forgetful functor. (This is familiar from lattice theory: the least element of a lattice is both the supremum of the empty set and the infimum of the entire lattice.)

Remark 9. Arrows of type $A \to \mathsf{Lim}\,\mathsf{U}$ and natural transformations of type $\Delta A \dot{\to} \mathsf{U}$ are in one-to-one correspondence. (In other words, we have an adjunction $\Delta \dashv \mathsf{Lim}$.) Using the concept of a *strong* dinatural transformation, the naturality property can be captured solely in terms of the underlying category $\mathscr{C}$ [11]. Whether a similar construction is also possible for ends is left for future work. □

7.4 List Monad

The Haskell programmer's favourite data structure, the type of parametric lists, arises out of an adjunction between **Mon**, the category of monoids and monoid homomorphisms, and **Set**, the category of sets and total functions.

$$\mathbf{Mon} \underset{\mathsf{U}}{\overset{\mathsf{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{Set}$$

Now U is the underlying functor that forgets about the monoidal structure, mapping a monoid to its carrier set. Its left adjoint Free maps a set A to the *free monoid on* A, whose elements are finite sequences of elements of A.

The adjunction $\mathsf{Free} \dashv \mathsf{U}$ induces the list monad List. For this instance, the isomorphism (48) can be simplified to

$$\mathsf{List} \cong \lambda\, A : \mathbf{Set} \,.\, \forall Z : \mathbf{Mon} \,.\, (A \to \mathsf{U}\,Z) \to \mathsf{U}\,Z \quad . \tag{53}$$

The variable Z now ranges over monoids. An element of the end can be seen as an evaluator: given a function of type $A \to \mathsf{U}\, Z$, which determines the meaning of singleton lists, the list represented can be homomorphically evaluated to an element of type $\mathsf{U}\, Z$.

Turning to Haskell, the standard implementation corresponds to the familiar datatype of lists.

$$\mathbf{data}\,\mathsf{List}_1\; a = \mathit{Nil} \mid \mathit{Cons}\,(a, \mathsf{List}_1\; a)$$

For the Church representation we take a similar approach as in the previous section: we introduce a class *Monoid* and constrain the universally quantified variable by a class context.

$$\begin{array}{l}
\mathbf{class}\;\mathit{Monoid}\;a\;\mathbf{where} \\
\quad \epsilon \quad : a \\
\quad (\bullet) : a \to a \to a \\
\mathbf{newtype}\,\mathsf{List}_2\; a = \mathit{Abstr}\,\{\,\mathit{apply} : \forall\, z\;.\;(\mathit{Monoid}\;z) \Rightarrow (a \to z) \to z\,\}
\end{array}$$

Of course, in Haskell there is no guarantee that an instance of *Monoid* is actually a monoid—this is a proof obligation for the programmer. We can turn the free constructions into monoids as follows:

$$\begin{array}{l}
\mathbf{instance}\;\mathit{Monoid}\;(\mathsf{List}_1\; a)\;\mathbf{where} \\
\quad \epsilon \qquad\qquad\quad = \mathit{Nil} \\
\quad \mathit{Nil} \bullet y \qquad\;\; = y \\
\quad \mathit{Cons}\,(a, x) \bullet y = \mathit{Cons}\,(a, x \bullet y) \\
\mathbf{instance}\;\mathit{Monoid}\;(\mathsf{List}_2\; a)\;\mathbf{where} \\
\quad \epsilon \qquad = \mathit{Abstr}\,(\lambda k \to \epsilon) \\
\quad x \bullet y = \mathit{Abstr}\,(\lambda k \to \mathit{apply}\; x\; k \bullet \mathit{apply}\; y\; k)\;\;.
\end{array}$$

The second instance is closely related to the Haskell code from the introduction: the implementation of backtracking using a success and a failure continuation simply specialises z to the monoid of endofunctions.

$$\begin{array}{l}
\mathbf{instance}\;\mathit{Monoid}\;(a \to a)\;\mathbf{where} \\
\quad \epsilon \qquad = \mathit{id} \\
\quad x \bullet y = x \cdot y
\end{array}$$

We can instantiate z to this monoid without loss of generality as every monoid is isomorphic to a monoid of endofunctions, the so-called Cayley representation, named after Arthur Cayley:

$$(A, \epsilon, \bullet) \cong (\{\,(a \bullet -) : A \to A \mid a \in A\,\}, \mathit{id}, \cdot)\;\;.$$

This isomorphism is also the gist of Hughes' efficient representation of lists [19].

To summarise, the CPS variant of the list monad combines Kan extensions and Cayley representations—Dan explains the success and Art the failure continuation.

8 Related Work

As mentioned in the introduction, all of the core results appear either explicitly or implicitly in Mac Lane's textbook [27]. Specifically, Section X.7 of the textbook introduces the notion of absolute Kan extensions—that $\mathsf{L} \dashv \mathsf{R}$ induces $-\circ\mathsf{R} \dashv -\circ\mathsf{L}$ is implicit in the proof of Theorem X.7.2. Overall, our paper solves Exercise X.7.3, which asks the reader to show that J/J is a monad and that $\mathsf{R}\circ\mathsf{L}$ is isomorphic to R/R.

Kan Extension. Kan extensions are named after Daniel Kan, who constructed **Set**-valued extensions using limits and colimits in 1960. Kan extensions have found a few applications in computer science: right Kan extensions have been used to give a semantics to generalised folds for nested datatypes [4]; left Kan extensions have been used to provide an initial algebra semantics for certain "generalised algebraic datatypes" [22].

Codensity Monad. The origins of the 'trick', wrapping a CPS transformation around a monad, seem to be unknown. The trick captured as a monad transformer was introduced by the author in 1996 [12]. Much later, Jaskelioff noted that the transformer corresponds to a construction in category theory, the codensity monad [21]. None of the papers that utilise the trick [13,8,33,20,28], however, employ the isomorphism $\mathsf{R}\circ\mathsf{L} \cong \mathsf{R}/\mathsf{R}$—all of them work with M/M instead. In more detail:

Building on the work of Hughes [18], the author showed how to derive backtracking monad transformers that support computational effects such as the Prolog cut and an operation for delimiting the effect of a cut [13]. Wand et al [35] later identified a problem with our derivations, which built on fold-unfold transformations [6]. Roughly speaking, the culprit is the lack of a sound induction principle for local universal quantification. Wand et al proposed an alternative approach based on logical relations. Their proof, however, uses a different CPS monad with a fixed type O of observations, $\mathsf{B}\,a = (a \to (O \to O)) \to (O \to O)$, which is somewhat unsatisfactory.

Claessen [8] applied the trick to speed up his parallel parsing combinators. Voigtländer [33] showed that the trick gives an asymptotic improvement for free algebras and the operation of substitution. He sketched a proof of correctness and conjectured that a formal proof might require sophisticated techniques involving free theorems. This gap was later filled by Hutton et al [20] who proved correctness by framing it as an instance of the so-called worker/wrapper transformation. Their proof, however, is an indirect one as it only establishes the equivalence of two functions from a common source into the two monads. Finally, a more advanced application involving indexed monads was recently given by McBride [28].

Kan extensions and the codensity monad are also popular topics for blog posts. Piponi (`blog.sigfpe.com`) expands on the codensity monad as "the mother of all monads", a catchy phrase due to Peter Hancock. Kmett (`comonad.com/reader`) has a series of posts on both topics, including a wealth of Haskell code.

String Diagram. String diagrams were introduced by Penrose [31] as an alternative notation for "abstract tensor systems". These diagrams are widely used for monoidal categories—Selinger [32] surveys graphical languages for different types of monoidal structures. (A monoidal category is a special case of a 2-category, namely, one that has only a single object. Consequently, there is no need to label or colour regions.)

9 Conclusion

Monads can be seen as abstract datatypes for computational effects. (This view is admittedly a bit limited—consider the free monad of a functor, would you want to regard substitution as a computational effect?) The take-home message of this paper is that the right Kan extension of the functor R along R is a drop-in replacement for the monad induced by the adjunction $\mathsf{L} \dashv \mathsf{R}$. The paper stresses the importance of adjunctions, a point already emphasised in a previous paper by the author [14]. The bad news is that the construction, the implementation of the codensity monad using ends and powers, does not lend itself easily to a library implementation, at least not in today's programming languages. A generic implementation would require support for abstraction over categories.

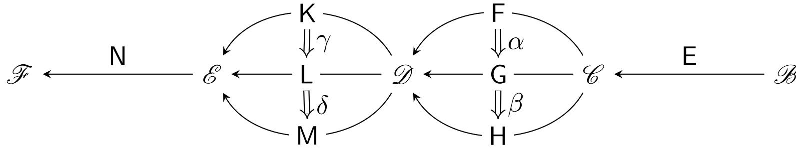
For quite a while I have been experimenting with two-dimensional notation for calculational proofs. I first tried traditional notation, the Poincaré dual of string diagrams, but I never used it in anger as I found the diagrams difficult to compose and to manipulate. The reason is that natural transformations, the main focus of interest, are represented by regions, which are often difficult to lay out in an aesthetically pleasing way. By contrast, string diagrams are fairly easy to draw. Furthermore, the least interesting piece of information—which are the categories involved?— can be easily suppressed. I hope to see string diagrams more widely used for program calculation in the future.

Everything we have said nicely dualises: right Kan extensions dualise to left Kan extensions and the codensity monad dualises to the density comonad. A left Kan extension can be seen as a generalised existential type and the density comonad corresponds to an abstract datatype or a simple object type—the type of parametric streams is one of the prime examples. Quite clearly, the dual story is interesting in its own right but this is a story to be told elsewhere.

Acknowledgements. I owe a particular debt of gratitude to Steven Vickers for pointing me to wire diagrams, which inspired me to explore two-dimensional notation. A big thank you is furthermore due to Daniel James and Nicolas Wu for artistically typesetting the diagrams. Nicolas also suggested various presentational improvements. Thanks are furthermore due to Roland Backhouse, Jeremy Gibbons and the other (anonymous) referees of MPC 2012 for finding several typographical errors, glitches of language, and for forcing me to be precise about the contributions of the paper. In particular, I would like to thank Roland for proposing the notation G/J instead of the more traditional $\mathsf{Ran}_{\mathsf{J}}\ \mathsf{G}$ and for pointing me to regular algebras. I have added Remarks 1, 3, 5, 6 and 7 in response to his review.

A Composition of Functors and Natural Transformations

This appendix contains supplementary material. It is intended primarily as a reference, so that the reader can re-familiarise themselves with the category theory that is utilised in this paper. Specifically, we introduce composition of functors and natural transformations. We shall use the following entities to frame the discussion ($\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ are parallel functors, $\alpha : \mathsf{F} \to \mathsf{G}$ is a natural transformation between them etc).



Here we use traditional two-dimensional notation. The reader is invited to turn the diagram into its Poincaré dual, a string diagram.

Functors can be composed, written $\mathsf{K}\circ\mathsf{F}$. Rather intriguingly, the operation $\mathsf{K}\circ-$, post-composing a functor K, is itself functorial: the higher-order functor $\mathsf{K}\circ- : \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$ maps the functor F to the functor $\mathsf{K}\circ\mathsf{F}$ and the natural transformation α to the natural transformation $\mathsf{K}\circ\alpha$ defined $(\mathsf{K}\circ\alpha)\,A = \mathsf{K}\,(\alpha\,A)$. Post-composition dualises to pre-composition: the higher-order functor $-\circ\mathsf{E} : \mathscr{D}^{\mathscr{C}} \to \mathscr{D}^{\mathscr{B}}$ maps the functor F to the functor $\mathsf{F}\circ\mathsf{E}$ and the natural transformation α to the natural transformation $\alpha\circ\mathsf{E}$ defined $(\alpha\circ\mathsf{E})\,A = \alpha\,(\mathsf{E}\,A)$. (The reader should convince themselves that $\mathsf{K}\circ\alpha : \mathsf{K}\circ\mathsf{F} \dot{\to} \mathsf{K}\circ\mathsf{G}$ and $\alpha\circ\mathsf{E} : \mathsf{F}\circ\mathsf{E} \dot{\to} \mathsf{G}\circ\mathsf{E}$ are again natural transformations.) Here are the functor laws spelled out.

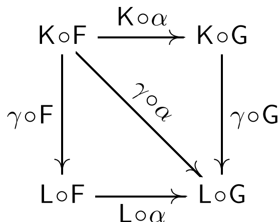
$$\mathsf{K}\circ id_{\mathsf{F}} = id_{\mathsf{K}\circ\mathsf{F}} \tag{54a}$$
$$\mathsf{K}\circ(\beta\cdot\alpha) = (\mathsf{K}\circ\beta)\cdot(\mathsf{K}\circ\alpha) \tag{54b}$$
$$id_{\mathsf{F}}\circ\mathsf{E} = id_{\mathsf{F}\circ\mathsf{E}} \tag{54c}$$
$$(\beta\cdot\alpha)\circ\mathsf{E} = (\beta\circ\mathsf{E})\cdot(\alpha\circ\mathsf{E}) \tag{54d}$$

Altogether, we have three different forms of composition: $\mathsf{K}\circ\mathsf{F}$, $\gamma\circ\mathsf{F}$ and $\mathsf{K}\circ\alpha$. They are 'pseudo-associative' and have the functor Id as their neutral element.

$$\gamma\circ(\mathsf{F}\circ\mathsf{E}) = (\gamma\circ\mathsf{F})\circ\mathsf{E} \tag{55a}$$
$$\mathsf{K}\circ(\beta\circ\mathsf{E}) = (\mathsf{K}\circ\beta)\circ\mathsf{E} \tag{55b}$$
$$\mathsf{N}\circ(\mathsf{M}\circ\alpha) = (\mathsf{N}\circ\mathsf{M})\circ\alpha \tag{55c}$$
$$\mathsf{Id}\circ\alpha = \alpha \tag{55d}$$
$$\alpha\circ\mathsf{Id} = \alpha \tag{55e}$$

This means that we can freely drop parentheses when composing compositions.

Given two natural transformations $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ and $\gamma : \mathsf{K} \dot{\to} \mathsf{L}$, there are two ways to turn a $\mathsf{K}\circ\mathsf{F}$ into an $\mathsf{L}\circ\mathsf{G}$ structure.



The diagram commutes since γ is natural:

$$
\begin{aligned}
& ((\gamma \circ \mathsf{G}) \cdot (\mathsf{K} \circ \alpha))\, X \\
= \; & \{ \text{ definition of compositions } \} \\
& \gamma\, (\mathsf{G}\, X) \cdot \mathsf{K}\, (\alpha\, X) \\
= \; & \{ \; \gamma \text{ is natural: } \mathsf{L}\, h \cdot \gamma\, A = \gamma\, B \cdot \mathsf{K}\, h \; \} \\
& \mathsf{L}\, (\alpha\, X) \cdot \gamma\, (\mathsf{F}\, X) \\
= \; & \{ \text{ definition of compositions } \} \\
& ((\mathsf{L} \circ \alpha) \cdot (\gamma \circ \mathsf{F}))\, X \quad .
\end{aligned}
$$

The diagonal is called the *horizontal composition* of natural transformations, denoted $\gamma \circ \alpha$.

$$(\gamma \circ \mathsf{G}) \cdot (\mathsf{K} \circ \alpha) = \gamma \circ \alpha = (\mathsf{L} \circ \alpha) \cdot (\gamma \circ \mathsf{F}) \tag{56}$$

The definition witnesses the fact that functor composition $\mathscr{E}^{\mathscr{D}} \times \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$ is a bi-functor: (56) defines its action on arrows.

References

1. Awodey, S.: Category Theory, 2nd edn. Oxford University Press (2010)
2. Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Category theory as coherently constructive lattice theory (1994), http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
4. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing 11(2), 200–222 (1999)
5. Böhm, C., Berarducci, A.: Automatic synthesis of typed λ-programs on term algebras. Theoretical Computer Science 39(2-3), 135–154 (1985)
6. Burstall, R., Darlington, J.: A tranformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
7. Cáccamo, M., Winskel, G.: A Higher-Order Calculus for Categories. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 136–153. Springer, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-44755-5_11
8. Claessen, K.: Functional pearl: Parallel parsing processes. Journal of Functional Programming 14(6), 741–757 (2004), http://dx.doi.org/10.1017/S0956796804005192
9. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 151–160. ACM Press (June 1990)
10. Eilenberg, S., Moore, J.C.: Adjoint functors and triples. Illinois J. Math 9(3), 381–398 (1965)
11. Ghani, N., Uustalu, T., Vene, V.: Build, Augment and Destroy, Universally. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 327–347. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-30477-7_22

12. Hinze, R.: Efficient monadic-style backtracking. Tech. Rep. IAI-TR-96-9, Institut für Informatik III, Universität Bonn (October 1996)
13. Hinze, R.: Deriving backtracking monad transformers. In: Wadler, P. (ed.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 186–197. ACM, New York (2000)
14. Hinze, R.: Adjoint folds and unfolds—an extended study. Science of Computer Programming (2011) (to appear)
15. Hinze, R., James, D.W.H.: Reason isomorphically! In: Oliveira, B.C., Zalewski, M. (eds.) Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP 2010), pp. 85–96. ACM, New York (2010)
16. Hoare, C.A.R., He, J.: The weakest prespecification. Inf. Process. Lett. 24(2), 127–132 (1987)
17. Huber, P.J.: Homotopy theory in general categories. Mathematische Annalen 144, 361–385 (1961), http://dx.doi.org/10.1007/BF01396534, doi:10.1007/BF01396534
18. Hughes, J.: The Design of a Pretty-Printing Library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)
19. Hughes, R.J.M.: A novel representation of lists and its application to the function reverse. Information Processing Letters 22(3), 141–144 (1986)
20. Hutton, G., Jaskelioff, M., Gill, A.: Factorising folds for faster functions. Journal of Functional Programming 20(Special Issue 3-4), 353–373 (2010), http://dx.doi.org/10.1017/S0956796810000122
21. Jaskelioff, M.: Modular Monad Transformers. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 64–79. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00590-9_6
22. Johann, P., Ghani, N.: Foundations for structured programming with GADTs. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 297–308. ACM, New York (2008), http://doi.acm.org/10.1145/1328438.1328475
23. Kan, D.M.: Adjoint functors. Transactions of the American Mathematical Society 87(2), 294–329 (1958)
24. Kleisli, H.: Every standard construction is induced by a pair of adjoint functors. Proceedings of the American Mathematical Society 16(3), 544–546 (1965), http://www.jstor.org/stable/2034693
25. Lambek, J.: From lambda-calculus to cartesian closed categories. In: Seldin, J., Hindley, J. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 376–402. Academic Press (1980)
26. Leivant, D.: Reasoning about functional programs and complexity classes associated with type disciplines. In: Proceedings 24th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1983, Tucson, AZ, USA, pp. 460–469. IEEE Computer Society Press, Los Alamitos (1983)
27. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics. Springer, Berlin (1998)
28. McBride, C.: Functional pearl: Kleisli arrows of outrageous fortune. Journal of Functional Programming (to appear)
29. Mellish, C., Hardy, S.: Integrating Prolog into the Poplog environment. In: Campbell, J. (ed.) Implementations of Prolog, pp. 533–535. Ellis Horwood Limited (1984)
30. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)

31. Penrose, R.: Applications of negative dimensional tensors. In: Welsh, D. (ed.) Combinatorial Mathematics and its Applications: Proceedings of a Conference held at the Mathematical Institute, Oxford, July 7-10, 1969, pp. 221–244. Academic Press (1971)
32. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) New Structures for Physics. Springer Lecture Notes in Physics, vol. 813, pp. 289–355. Springer, Heidelberg (2011)
33. Voigtländer, J.: Asymptotic Improvement of Computations over Free Monads. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 388–403. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70594-9_20
34. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989), London, UK, pp. 347–359. Addison-Wesley Publishing Company (September 1989)
35. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, pp. 54–65. ACM, New York (2004), http://doi.acm.org/10.1145/1016850.1016861

Author Index

Armstrong, Alasdair 220

Backhouse, Roland 157
Bahr, Patrick 263
Barthe, Gilles 1

Clarke, Dave 132

Dang, Han-Hing 177
Dongol, Brijesh 102

Endres, Markus 241

Foster, Simon 220

Ghica, Dan R. 23
Grégoire, Benjamin 1
Guttmann, Walter 198

Hayes, Ian J. 102
Hinze, Ralf 324
Hoare, Tony 7

Lux, Alexander 25

Mandel, Louis 74
Mantel, Heiko 25
Midtgaard, Jan 132
Möller, Bernhard 177, 241
Morgan, Carroll C. 48

Paterson, Ross 300
Perner, Matthias 25
Plateau, Florence 74

Roocks, Patrick 241

Sergey, Ilya 132
Struth, Georg 220

van Staden, Stephan 7

Zanella Béguelin, Santiago 1