Robert Glück
Michael Lowry (Eds.)

LNCS 3676

Generative Programming and Component Engineering

4th International Conference, GPCE 2005
Tallinn, Estonia, September/October 2005
Proceedings

Springer

Lecture Notes in Computer Science 3676

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

Editorial Board

Robert Glück Michael Lowry (Eds.)

Generative Programming and Component Engineering

4th International Conference, GPCE 2005
Tallinn, Estonia, September 29 – October 1, 2005
Proceedings

Springer

Volume Editors

Robert Glück
University of Copenhagen, DIKU, Department of Computer Science
Universitetsparken 1, 2100 Copenhagen, Denmark
E-mail: glueck@acm.org

Michael Lowry
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035-1000, USA
E-mail: lowry@email.arc.nasa.gov

Library of Congress Control Number: 2005932566

CR Subject Classification (1998): D.2, D.1, D.3, K.6

ISSN 0302-9743
ISBN-10 3-540-29138-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29138-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11561347 06/3142 5 4 3 2 1 0

Preface

Generative Programming and Component Engineering (GPCE) is a leading research conference on automatic programming and component engineering. These approaches to software engineering have the potential to revolutionize software development as automation and components revolutionized manufacturing. The conference brings together researchers and practitioners interested in advancing automation for software development. It is also a premier forum for cross-fertilization between the programming language and software engineering research communities.

GPCE arose as a joint conference, merging the prior conference on Generative and Component-Based Software Engineering (GCSE) and the Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG). The proceedings of the previous GPCE conferences were published in the LNCS series of Springer as volumes 2487, 2830, and 3286. In 2005 GPCE was co-located with the International Conference on Functional Programming (ICFP) and the symposium on Trends in Functional Programming (TFP), reflecting the vigorous interaction between the functional programming and generative programming research communities. GPCE and ICFP are both sponsored by the Association for Computing Machinery.

The quality and breadth of the papers submitted to GPCE 2005 was impressive. All 86 papers, including 5 papers for tool demonstrations, were rigorously reviewed by 17 highly qualified Program Committee members. The members of the Program Committee first provided in-depth individual reviews of the submitted papers, and then debated the merits of the papers through an extended electronic Program Committee meeting. After much (friendly) argument, 25 regular papers and 2 tool demonstration papers were selected for publication. The Program Committee provided extensive technical feedback to the authors of the submitted papers. The conference program was complemented with three invited talks, three extended tutorials, and three all-day workshops.

The accepted papers are grouped into eight topic areas: aspect-oriented programming, component engineering and templates, demonstrations, domain-specific languages, generative techniques, generic programming, meta-programming and transformation, and multi-stage programming. The invited talks were from leading innovators in the field: Oscar Nierstrasz on object-oriented reengineering patterns, Oege de Moor on the AspectBench compiler for AspectJ, and Bernd Fischer on certifiable program generation.

The program chairs would like to thank foremost the authors of the submitted papers: their research is the justification for this conference. Both program chairs were impressed by the expertise and diligence of the Program Committee members and their co-reviewers. Their technical dedication, as reflected in the quality of their reviews, was the foundation of the strength of these proceedings.

The general chair, Eugenio Moggi, was tireless in steering the program chairs towards a technically superb program. The publicity chair, Eelco Visser, went beyond the call of duty in raising awareness of the conference in the software engineering and programming languages research communities. Andrew Malton and Jeff Gray solicited and organized a workshop and tutorial program of interest to researchers and practitioners alike. Tarmo Uustalu graciously served as local arrangements chair, providing a hospitable atmosphere in the beautiful venue of Tallin, Estonia. The paper submissions and the reviewing process were ably supported by the Web-based EasyChair system (http://www.easychair.org/). The program chairs would like to extend our appreciation to Andrei Voronkov, who developed EasyChair and is the leading force behind its continued development. His personal attention to our conference greatly facilitated managing the volume of reviews and discussions amongst the Program Committee. Finally, we would like to recognize the importance of the gentle guidance of the GPCE Steering Committee. Their long-term dedication is the core that binds together this research community.

July 2005

Robert Glück
Michael Lowry

Organization

General Chair

Eugenio Moggi (Genoa University, Italy)

Program Committee Co-chairs

Robert Glück (University of Copenhagen, Denmark)
Michael Lowry (NASA Ames Research Center, USA)

Program Committee

Don Batory (University of Texas, USA)
Ira Baxter (Semantic Designs, USA)
Cristiano Calcagno (Imperial College London, UK)
Prem Devanbu (University of California at Davis, USA)
Ulrich Eisenecker (University of Leipzig, Germany)
Tom Ellman (Vassar College, USA)
Robert Filman (NASA Ames Research Center, USA)
Zhenjiang Hu (University of Tokyo, Japan)
Patricia Johann (Rutgers University, USA)
John Launchbury (Galois Connections Inc., USA)
Anne-Françoise Le Meur (University of Science and Technology Lille, France)
Hong Mei (Peking University, China)
Nicolas Rouquette (NASA Jet Propulsion Lab, USA)
William Scherlis (Carnegie Mellon University, USA)
Yannis Smaragdakis (Georgia Institute of Technology, USA)
Walid Taha (Rice University, USA)
Todd Veldhuizen (Chalmers University of Technology, Sweden)

Publicity Chair

Eelco Visser (Utrecht University, The Netherlands)

Workshop and Tutorial Chairs

Andrew Malton (University of Waterloo, Canada)
Jeff Gray (University of Alabama at Birmingham, USA)

Local Arrangements Chair

Tarmo Uustalu (Institute of Cybernetics, Estonia)

Additional Referees

Alexander Ahern
Robert L. Akers
Olivier Barais
Josh Berdine
Christie Bolton
John Tang Boyland
Dolores Diaz
Dan Dougherty
Laurence Duchien
Bernd Fischer
Aaron Greenhouse
Timothy J. Halloran
William Harrison
Jim Hook
Liwen Huang
Shan Shan Huang
Samuel Kamin
Paul Kelly
Oleg Kiselyov
Julia Lawall
Christopher League
Daan Leijen
Dongxi Liu
Jan-Willem Maessen
Michael Mehlich
Eugenio Moggi
Shin-Cheng Mu
Keisuke Nakano
Matthias Neubauer
Emir Pasalic
Renaud Pawlak
Nicolas Pessemier
Amr Sabry
Isao Sasano
Lionel Seinturier
Douglas Smith
Andreas Speck
Franklyn Turbak
Geoffrey Washburn
Tetsuo Yokoyama
Nobuko Yoshida
Hongjun Zheng
David Zook

Sponsors

GPCE 2005 was sponsored by ACM SIGPLAN, in cooperation with ACM SIGSOFT. We gratefully acknowledge the support of Utrecht University in hosting the conference Web site, and the Institute of Cybernetics, Tallinn, for handling the local arrangements and registration.

Table of Contents

Invited Talks

Domain-Specific Language

Aspect-Oriented Programming

Meta-programming and Transformation

Generative Techniques I

Multi-stage Programming

Generative Techniques II

Components and Templates

Generic Programming

Demonstrations

Object-Oriented Reengineering Patterns An Overview

Oscar Nierstrasz[1], Stéphane Ducasse[2], and Serge Demeyer[3]

[1] Software Composition Group, University of Bern, Switzerland
[2] Laboratoire d'Informatique, Systèmes, Traitement de l'Information, et de la Connaissance, Université de Savoie, France
[3] Lab On REengineering, University of Antwerp, Belgium

Abstract. Successful software systems must be prepared to evolve or they will die. Although object-oriented software systems are built to last, over time they degrade as much as any legacy software system. As a consequence, one must invest in reengineering efforts to keep further development costs down. Even though software systems and their business contexts may differ in countless ways, the techniques one uses to understand, analyze and transform these systems tend to be very similar. As a consequence, one may identify various *reengineering patterns* that capture best practice in reverse- and re-engineering object-oriented legacy systems. We present a brief outline of a large collection of these patterns that have been mined over several years of experience with object-oriented legacy systems, and we indicate how some of these patterns can be supported by appropriate tools.

1 Introduction

A *legacy software system* is a system that you have *inherited* and is *valuable* to you. Successful (*i.e.*, valuable) software systems typically evolve over a number of years as requirements evolve and business needs change. This leads to the well-documented phenomenon that such systems become more *complex* over time, and become progressively harder to maintain, unless special measures are taken to simplify their architecture and design [13].

Numerous problems manifest themselves as a legacy system begins to turn into a burden. First of all, *knowledge* about the system deteriorates. Documentation is often missing or obsolete. The original developers or users may have left the project. As a consequence, inside knowledge about the system may be missing. Automated tests that document how the system functions are rarely available.

Second, the *process* for implementing changes ceases to be effective. Simple changes take too long. A continuous stream of bug fixes is common. Maintenance dependencies make it difficult to implement changes or to separate products.

Finally, the *code* itself will exhibit various disagreeable symptoms. Large amounts of duplicated code are common, as are other "code smells" such as violations of encapsulation, large, procedural classes, and explicit type checks.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 1–9, 2005.

Concretely, the code will manifest *architectural problems* such as improper layering and lack of modularity, as well as *design problems* such as misuse of inheritance, missing inheritance and misplaced operations. Excessive build times are also a common sign of architectural decay.

Since the bulk of a (successful) software system's life cycle is known to reside in maintenance, and "maintenance" is known to consist largely in the introduction of new functionality [14], identifying and resolving these problems becomes critical for the survival of legacy systems.

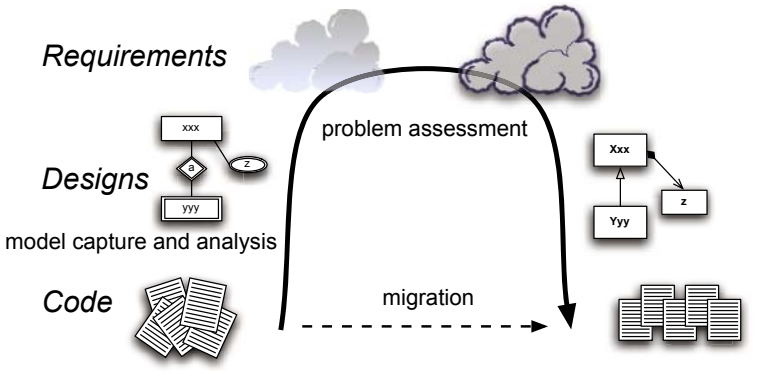


Fig. 1. The Reengineering life cycle

To this end, it is useful to distinguish *reverse engineering* from *reengineering* of software systems [2]. By "reverse engineering", we mean the process of analyzing a software system in order to expose its structure and design at a higher level of abstraction, *i.e.*, the process of extracting various *models* from the concrete software system. By "reengineering" we refer to the process of transforming the system to a new one that implements essentially the same functional requirements, but also enables further development.

The process of reverse- and re-engineering consists of numerous activities, including architecture and design recovery, test generation, problem detection, and various high and low-level refactorings. In Figure 1 we see an ideal depiction of the reverse- and re-engineering life cycle [3,10].

Although the motivations for reengineering a legacy system may vary considerably according to the business needs of the organization, the actual technical steps taken tend to be very similar. As a consequence, it is possible to identify a number of generally useful *process patterns* that one may apply while reverse- and re-engineering a legacy system. We provide a brief overview of these patterns in Section 2. By the same token, there exist various tools that can help support the reengineering process. In Section 3 we present a brief outline of some of the tools we have developed and applied to various legacy systems.

2 Reengineering Patterns

The term "pattern" used in the context of software usually evokes the notion of "design patterns" — recurring solutions to design problems. *Reengineering patterns* are not design patterns, but rather *process patterns* — recurring solutions to problems that arise during the process of reverse- and re-engineering.

We distinguish patterns from "rules" or "guidelines" because each pattern must be interpreted in a given context. Patterns are not applied blindly, but entail tradeoffs. Just as one would never deliberately implement a software system applying all of the GOF patterns [7], one should not blindly apply reengineering patterns without considering all the consequences.

We were able to mine a large number of reengineering patterns during the course of FAMOOS, a European project[1] whose goal was to support the evolution of first-generation object-oriented software towards object-oriented frameworks. FAMOOS focussed on methods and tools to analyse and detect design problems in object-oriented legacy systems, and to migrate these systems towards more flexible architectures. The main results of FAMOOS are summarized in the FAMOOS Handbook [4] and in the book "Object-Oriented Reengineering Patterns" [3].

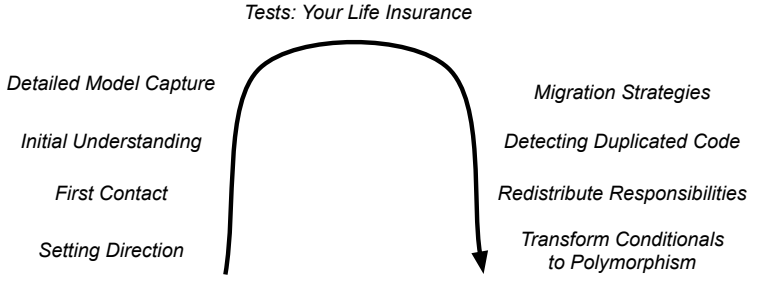


Fig. 2. Reengineering pattern clusters

In Figure 2 we see how various clusters of reengineering patterns can be mapped to our ideal reengineering life cycle. Each name represents a collection of process patterns that can be applied at a particular stage during the reengineering of a legacy system.

Setting Direction contains several patterns to help you determine where to focus your re- engineering efforts, and make sure you stay on track. *First Contact* consists of a set of patterns that may be useful when you encounter a legacy system for the first time. *Initial Understanding* helps you to develop a first simple model of a legacy system, mainly in the form of class diagrams.

[1] ESPRIT Project 21975: "Framework-based Approach for Mastering Object-Oriented Software Evolution". `www.iam.unibe.ch/~scg/Archive/famoos`

Detailed Model Capture helps you to develop a more detailed model of a particular component of the system. *Tests: Your Life Insurance* focusses on the use of testing not only to help you understand a legacy system, but also to prepare it for a reengineering effort. *Migration Strategies* help you keep a system running while it is being reengineered, and increase the chances that the new system will be accepted by its users. *Detecting Duplicated Code* can help you identify locations where code may have been copied and pasted, or merged from different versions of the software. *Redistribute Responsibilities* helps you discover and reengineer classes with too many responsibilities. *Transform Conditionals to Polymorphism* will help you to redistribute responsibilities when an object-oriented design has been compromised over time.

Since a detailed description of the patterns is clearly out of the scope of a short paper, let us just briefly consider a single pattern cluster. *First Contact* consists of patterns that can be useful when first encountering a legacy system. There are various *forces* at play, which one must be conscious of. In particular, legacy systems tend to be *large* and complex, so it will be difficult to get an overview of the system. *Time is short*, so it is important to gather quality information quickly. Furthermore, *first impressions are dangerous*, so it is important not to rely on a single source of information.

One has various resources at hand: the source code, the running system, the users, the maintainers, documentation, the source code repository, the changes log, the list of bug requests, the test cases, and so on. Even if some of these are missing or unreliable, one must take care to not reject anything out of hand.

In Figure 3 we see a map of the patterns in this cluster, and how they relate to each other. As with each pattern cluster, patterns support each other to resolve the forces at play. The *First Contact* cluster resolves the forces by balancing what you learn from the users and maintainers with what you learn from the source code.

In Figure 4 we see a capsule summary of one of the better-known patterns of this cluster. The *name* is typically an action to be performed, that expresses the key idea of the pattern. Not every pattern is always relevant in every context, so one must be clear about the *intent* of each pattern, the *problem* it solves, the key idea of the *solution*, and the *tradeoffs* entailed. In this particular pattern, the context of a demo is used as a device to help the user to focus on concrete rather than abstract qualities of the application, while communicating typical use cases and scenarios to the engineer. Each pattern may also include *hints*, *variants*, *examples*, *rationale*, *related patterns*, and an indication of *what to do next*. *Known uses* are very important, since only established best practices can truly be considered "patterns".

3 Reengineering Tools and Techniques

It is easy to put too much faith into tools. For this reason the reengineering patterns put more emphasis on process than tools. (As a popular saying puts it: "A fool with a tool is still a fool.")

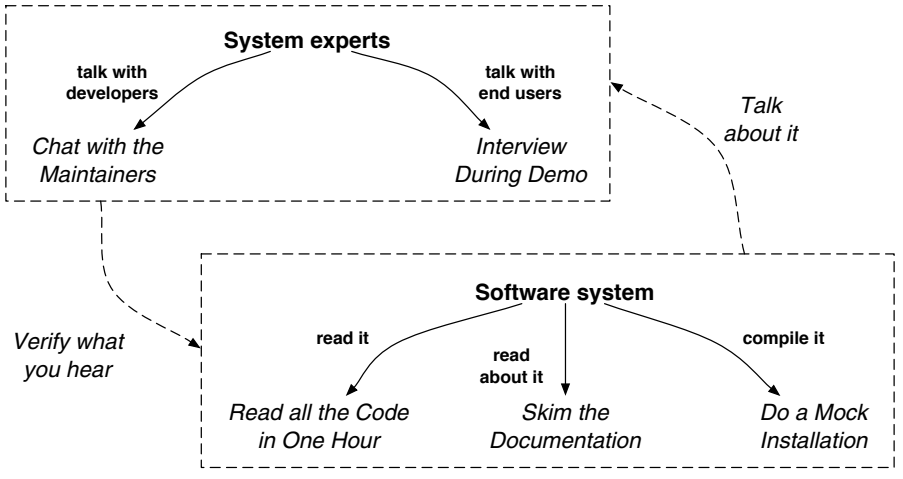


Fig. 3. First Contact

Nevertheless, certain activities can be streamlined with the help of carefully chosen tools. In particular, the process of reverse engineering can be aided by tools that build models from source code. Note that it is *not* a question of generating UML diagrams from source code. (10'000 class diagrams do not necessarily aid program comprehension more than 1'000'000 lines of source code.)

One the other hand, during *Initial Understanding*, a key pattern is *Study the Exceptional Entities*. Very often it is the software entities that are very large, very small, most tightly coupled, inherit the most, inherit the least, *etc.*, that tell one the most about how a software system works. It may be that these outliers are indicative of design problems, but this need not be the case.

CodeCrawler is a tool that presents simple visualizations of software entities based on direct metrics [12]. A polymetric view, is a two-dimensional visualization of nodes (as entities) and edges (as relationships) that maps various metric values to attributes of the nodes and edges. For example, different metrics can be mapped to the size, position and color of a node, or to the thickness and color of the edge.

Polymetric views can be generated for different purposes: coarse-grained views to assess global system properties, fine-grained views to assess properties of individual software artifacts, and evolutionary views to assess properties over time.

Figure 5 shows a System Complexity View which is coarse grained view [11]. The figure shows the hierarchies of CodeCrawler itself. Each node represents a class, and each edge represents an inheritance relationship. The height of a node represents the number of methods, the width represents the number of attributes and the (greyscale) color represents the number of lines of code. A System Complexity View can help one to quickly identify many kinds of outliers. For example, tall, isolated, dark nodes have many methods, many lines of

Name	*Interview During Demo*
Intent	Obtain an initial feeling for the appreciated functionality of a software system by seeing a demo and interviewing the person giving the demo.
Problem	How can you get an idea of the typical usage scenarios and the main features of a software system?
Solution	Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Note that the interviewing part is at least as enlightening as the demo.
Hints	The user who is giving the demo is crucial to the outcome of this pattern so take care when selecting the person. Therefore, do the demonstration several times with different persons giving the demo.
Tradeoffs	*Pro:* Focuses on valued features. *Con:* Provides anecdotal evidence only. *Difficulties:* Requires interviewing experience.
Example	*(Description of a typical interview ...)*
Rationale	Because users must start from a working system, they will adopt a positive attitude in explaining what works. The interviewer can ask precise questions, get precise answers, thus digging out the expert knowledge about the system's usage.
Known Uses	Commonly used for evaluating user-interfaces.
Related Patterns	See *Customer Interaction Patterns* [17]
What Next	Carry out several attempts of *Interview During Demo* with different kinds of stakeholders. Perform these attempts before, after or interwoven with *Read all the Code in One Hour* and *Skim the Documentation*. Afterwards, consider to *Chat with the Maintainers* to verify some of your findings.

Fig. 4. A pattern in a nutshell

code, and few attributes, and they may be signs of procedural classes with long, algorithmic methods.

CodeCrawler is built on top of Moose, a reengineering environment that offers a common infrastructure for various reverse- and re-engineering tools [5,15]. At the core of Moose is a common meta-model for representing software systems in a language-independent way. Around this core are provided various services that are available to the different tools. These services include metrics evaluation and visualization, a repository for storing multiple models, a meta-meta model for tailoring the Moose meta-model, and a generic GUI for browsing, querying and grouping.

Some other tools that have been developed either in the context of Famoos, or subsequently as clients of Moose, include:

- Duploc— detects duplicated code in large software systems in a language-independent way [6,16].
- ConAn— applies formal concept analysis to detect implicit contracts in object-oriented software [1].

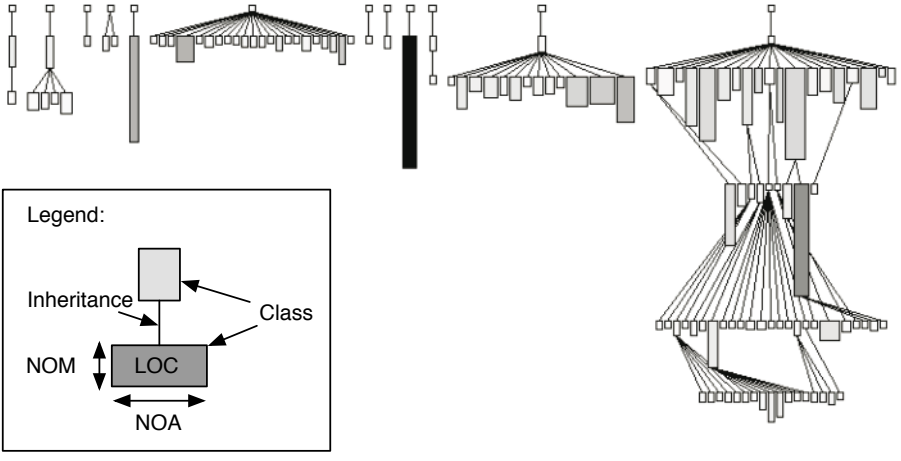


Fig. 5. A System Complexity view of CodeCrawler

- Van— analyzes version histories of software systems to uncover trends [8].
- TraceScraper— analyzes run-time traces of instrumented software to correlate features with software artifacts [9].

4 Conclusions

Given the premise that "the only constant is change", any interesting software system must evolve to stay interesting. As a consequence, however, we must invest in reengineering if the architecture and design of the system is to stay abreast of the changing requirements. Even though every system is different, we can identify various useful *reengineering patterns* that ease the process of understanding a complex legacy system, identifying its problems, and transforming it to a more flexible design.

The patterns we have documented include only those for which we have personally witnessed success. The Famoos reengineering patterns therefore represent only a starting point, and not a definitive work. What is important is that each pattern document best practice as experienced by experts in the field, as opposed to new research ideas that have not yet been proven in industrial contexts. There is clearly much research that can be done to investigate, for example, the synergy between tools and reengineering patterns, but one must not confuse the two.

We hope that the value of reengineering patterns, and more generally process patterns, will increasingly be recognized and encouraged as an effective means to improve the state of the art and disseminate best practice.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applica-

tions" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006). Thanks are due to Laura Ponisio for suggesting several improvements in the text.

References

1. Gabriela Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, January 2005.
2. Elliot J. Chikofsky and James H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.
3. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
4. Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, October 1999.
5. Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.
6. Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance: Research and Practice*, 2005. To appear.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
8. Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.
9. Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.
10. R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.
11. Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.
12. Michele Lanza and Stéphane Ducasse. Codecrawler — an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005.
13. Manny M. Lehman and Les Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.
14. Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison Wesley, Boston, MA, 1980.

15. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Girba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*. LNCS, 2005. Invited paper. To appear.
16. Matthias Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Berne, June 2005.
17. Linda Rising. Customer interaction patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 585–609. Addison Wesley, 2000.

abc: The AspectBench Compiler for AspectJ*

Chris Allan[1], Pavel Avgustinov[1], Aske Simon Christensen[2], Laurie Hendren[3], Sascha Kuzins[1], Jennifer Lhoták[3], Ondřej Lhoták[3], Oege de Moor[1], Damien Sereni[1], Ganesh Sittampalam[1], and Julian Tibble[1]

[1] Programming Tools Group, Oxford University, United Kingdom
[2] BRICS, University of Aarhus, Denmark
[3] Sable Research Group, McGill University, Montreal, Canada

Abstract. *abc* is an extensible, optimising compiler for AspectJ. It has been designed as a workbench for experimental research in aspect-oriented programming languages and compilers. We outline a programme of research in these areas, and we review how *abc* can help in achieving those research goals.

1 Goals

Aspect-oriented programming has become hugely successful as a highly disciplined form of program generation. Unlike many other forms of program generation, it is based on the idea of transforming program execution rather than program syntax: an aspect observes a base program, and when certain programmer-specified events occur, it runs some extra code of its own. This basis of transforming executions rather than syntax make aspects less brittle than more traditional forms of meta-programming. The most popular implementation of these ideas is AspectJ, an extension of Java [19].

While there is general consensus about the importance of aspects, the design space for aspect-oriented programming languages is not yet well understood. Many questions need to be answered, for example:

Definition of events. What is the nature of events observed by an aspect? Are they individual actions (like a method call)? Or perhaps the call stack (as in AspectJ's **cflow** construct) [33]? Or perhaps the complete history of the computation to date, as some kind of trace [12, 32]? Some authors have even suggested constructs that inspect events that *may* happen in the future [18, 21].

Definition of patterns. How do we describe the events that can be intercepted? In AspectJ, the language of patterns is fairly simple and restricted. Many authors have suggested a notation based on logic programming would be appropriate *e.g.* [15, 22], but would that render compile times unacceptably high? Other proposals include the use of temporal logic [30], regular expressions [3], and even context-free grammars to describe traces [32].

Modular reasoning. Is it desirable to hide some events, to allow for modular reasoning, where a module can be freely replaced by another implementation, without

* This work was supported, in part, by IBM, and by EPSRC in the United Kingdom, and NSERC in Canada.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 10–16, 2005.

affecting system behaviour [2]? Indeed, what is the correct notion of 'modular reasoning' in the context of aspects [20]? Such modular reasoning may also be facilitated by identifying aspects that are *harmless* in the sense that they do not interfere with the operation of the base program [10, 28].

Adding state to existing classes. Aspects often need to add state to existing classes, in the form of new fields and methods to manipulate those new fields. What are appropriate language mechanisms for that, and how do they interact with inheritance? Recent work on improved virtual types [8], and nested inheritance [26] address similar issues in a principled way — can they succesfully be combined with aspects?

Aspect composition. How are aspects composed? In AspectJ, the interaction between aspects is controlled via *precedence declarations*, but in comparison to other work on *feature composition* such declarations are not very expressive [7]. Recently the connection between feature-oriented programming and aspects has been made explicit, and this has suggested improvement to AspectJ [23]. Perhaps earlier works on object and module calculi may be adapted to give a rigorous account of aspect composition [9]?

Overheads and optimisation. What are the overheads of using aspects at runtime? Until recently, it was believed such overheads are negligible. Indeed, in [25] a very nice explanation is given (in terms of partial evaluation) of how one can do much of the event-pattern matching at compile-time instead of runtime. Even with those optimsations, however, precise measurements have shown that the use of **cflow** and **around** can still lead to substantial loss of performance [13]. What are the optimisations for eliminating such overheads? A first step towards addressing these issues is presented in [6].

The goal of *abc*, the AspectBench Compiler for AspectJ, is to provide a workbench for experimental research in all these areas. *abc* is a full implementation of the AspectJ language, with an extensible frontend and a framework for implementing sophisticated analyses and optimisations. *abc* is freely available under the LGPL [1].

2 Architecture

The frontend of *abc* is based on Polyglot, a framework for experimenting with extensions of Java [27]. In Polyglot, all compiler passes are implemented as non-destructive rewriting of the AST; the rewrite operations themselves are encoded direct in Java via the visitor pattern. Furthermore, Polyglot features the use of *delegates* to enable extensions to add further members in the middle of the inheritance hierarchy. This functionality is similar to the intertype declarations of AspectJ, but it is encoded in pure Java.

The backend of *abc* is based on Soot, a framework for analysing and transforming Java bytecode [31]. It most notably provides a typed, stackless intermediate representation named *Jimple* — this is much more convenient for analysis and transformation than normal bytecode. The backend includes the *advice weaver* that inserts extra code (named *advice* in AspectJ) at points specified by the patterns (called *pointcuts* in AspectJ).

Details of the architecture are depicted in Figure 1. The frontend reads Java and class files, and based on these it produces an abstract syntax tree (AST) for the AspectJ program. The *separator* then strips out all aspect-specific features, into a pure Java AST and a datastructure that we call the *AspectInfo*. The *AspectInfo* contains all information necessary to *weave* the aspects into bytecode: it could thus be thought of as a meta-program that is executed by later phases of the compiler. In the next step, we alter the type hierarchy as required for the aspects (by adding new members introduced via intertype declarations, as well as new superclasses and interfaces as stipulated by **declare parents** declarations). At this point we can generate Jimple code for the pure Java parts of the program. We then weave in the advice, and apply any optimisations necessary, before finally producing bytecode.

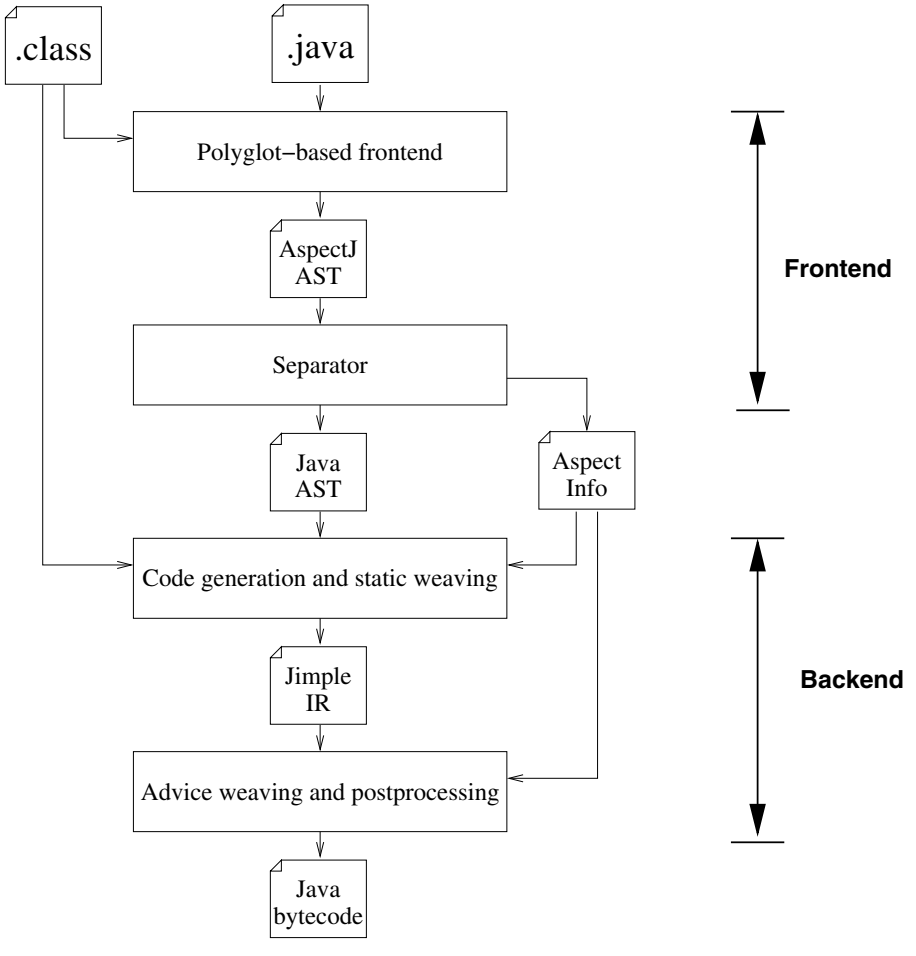


Fig. 1. Architecture of *abc*

3 Example Extensions

The design of *abc* was guided by a number of small extensions of our own, which are described in [5]. The largest example that we have investigated thus far is the addition of trace matches with free variables [3]. This allows one to write patterns that range not over individual events like method calls, but instead over program traces. Such proposals were made earlier by [12, 32]: our contribution is the introduction of variable binding in the patterns. This feature allows one to track the behaviour of individual objects.

To illustrate, consider the safe usage of iterators. We wish to check at runtime that no iterator is used after its underlying collection has been modified. In our extension of AspectJ, this would be written as shown in Figure 2. In the header of the trace match, we declare the free variables bound in the pattern, here an iterator *i* and a *datasource*. Next we declare three *symbols*, which make up the alphabet of interest: the creation of an iterator, the *next* operation, and the modification of the datasource. We then specify the pattern that should trigger an exception: create an iterator, do zero or more *next* operations, modify the datasource, and then do one more *next*. Of course one might also argue that calling *Iterator.hasNext()* would be an error after modifying the datasource: the pattern is easily modified to accommodate that if desired.

This example does in fact highlight another important direction for future work on AspectJ, namely the ability to check this type of property at compile time. AspectJ

```
// track iterator i on data source ds
tracematch (Iterator i, DataSource ds) {
  // declare alphabet of interest
  sym create_iter after returning(i):
    call(Iterator DataSource.iterator())
    && target(ds);
  sym call_next before :
    call(Object Iterator.next())
    && target(i);
  sym update_source after:
    call(* DataSource.update(..))
    && target(ds);
  // regular pattern that will trigger extra code
  create_iter call_next*
    update_source call_next
  {
    throw new
      ConcurrentModificationException();
  }
}
```

Fig. 2. Example trace match

already has rudimentary features (**declare warning/error**) for this purpose, and it is natural to extend them so that the language provides a continuum of dynamic and static property checking. The type of static analysis to realise this vision is likely to be similar to that of [11, 16]; furthermore these analyses can serve to optimise the use of trace matches. Other related proposals are [14, 24]. In fact, these works point towards the possibility of a generalised language of queries instead of AspectJ's existing pointcut language, where each query has a dynamic interpretation as well as a static approximation for compile-time checking.

4 Evaluation

The first version of *abc* was released on October 22, 2004. Now is therefore a good time to take stock, and evaluate its success to date. Perhaps the best indicator is the extent to which *abc* has been adopted by other research groups. At the time of writing, we are aware of over 10 teams worldwide who use *abc* for their own research projects. To mention just a few examples of such extensions by others: *LoopsAJ*, an extension for loop join points for use in scientific computing [17]; *SCoPE*, an extension that supports static conditional pointcut evaluation [4]; *J-LO*, an extension for supporting the dynamic checking of temporal properties [30]; and *Cona*, an extension which supports contracts for AspectJ [29].

We believe that this provides good evidence that *abc* is an adequate framework to attack the research problems listed in Section 1. Indeed, all of these topics are currently under investigation in the *abc* project. Given the success of *abc* in gaining adoption by others, we are also urgently pursuing the problem of combining independent language extensions.

References

1. abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.
2. Jonathan Aldrich. Open modules: modular reasoning about advice. In *European Conference on Object-Oriented Programming (ECOOP)*, to appear, 2005.
3. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, to appear, 2005.
4. Tomoyuki Aotani and Hidehiko Masuhara. Compiling conditional pointcuts for user-level semantic pointcuts. In *Proceedings of the SPLAT workshop at AOSD 2005*, 2005. Workshop proceedings available from: `http://www.daimi.au.dk/ eernst/ splat05/papers/`.
5. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

6. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005.
7. Don Batory. A tutorial on feature-oriented programming and the AHEAD tool suite. In *Summer school on Generative and Transformation Techniques in Software Engineering*, Lecture Notes in Computer Science, to appear, 2005.
8. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-oriented Programming (ECOOP)*, pages 523–549, 1998.
9. Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Department of Computer Science, Iowa State University, Ames, Iowa, 2003. Available from: `http://www.cs.iastate.edu/~cclifton/papers/TR03-13.pdf`.
10. Daniel S. Dantas and David Walker. Harmless advice. In *Foundations of Object-Oriented Languages (FOOL '05)*, 2005. Workshop proceedings available from: `http://homepages.inf.ed.ac.uk/wadler/fool/program/`.
11. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI)*, pages 56–68, 2002.
12. Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In *Aspect-oriented Software Development*, pages 141–150. Addison-Wesley, 2004.
13. Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 150–169, 2004.
14. Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, to appear, 2005.
15. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Aspect-Oriented Software Development (AOSD)*, pages 60–69. ACM Press, 2003.
16. Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Programming Language Design and Implementation (PLDI)*, pages 69–82. ACM Press, 2002.
17. Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *Aspect-Oriented Software Development (AOSD)*, pages 122–131. ACM Press, 2004.
18. Gregor Kiczales. The fun has just begun. Keynote address at AOSD. Available at `aosd.net/archive/2003/kiczales-aosd-2003.ppt`, 2003.
19. Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
20. Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering (ICSE '05)*, pages 49–58, 2005.
21. Karl Klose and Klaus Ostermann. Back to the future: pointcuts as predicates over traces. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL '05)*, pages 33–38, 2005. Workshop proceedings available from: `http://archives.cs.iastate.edu/documents/disk0/00/00/03/61/`.

22. Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 111–126. June 2004. Workshop proceedings available from: `http://www.disi.unige.it/person/CazzolaW/RAM-SE04.html`.
23. Roberto Lopez-Herrejon and Don Batory. Improving incremental development in AspectJ by bounding quantification. In *Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2005. Workshop proceedings available from: `http://www.daimi.au.dk/~eernst/splat05/papers/`.
24. Michael Martian, V. Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: A program query language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, to appear, 2005.
25. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.
26. Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Object-Oriented Programming, Systems, and Languages (OOPSLA)*, pages 99–115, 2004.
27. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.
28. Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the Twelfth International Symposium on the Foundations of Software Engineering*, pages 147–158, 2004.
29. Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 196–197, New York, NY, USA, 2004. ACM Press.
30. Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*, Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, UK, to appear, 2005. Elsevier Science Publishers.
31. Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction (CC)*, pages 18–34, 2000.
32. Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Foundations of Software Engineering (FSE)*, pages 159–169, 2004.
33. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.

Certifiable Program Generation

Ewen Denney and Bernd Fischer

USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
{edenney, fisch}@email.arc.nasa.gov

Abstract. Code generators based on template expansion techniques are easier to build than purely deductive systems but do not guarantee the same level of assurance: instead of providing "correctness-by-construction", the correctness of the generated code depends on the correctness of the generator itself. We present an alternative assurance approach, in which the generator is extended to enable Hoare-style safety proofs for each individual generated program. The proofs ensure that the generated code does not "go wrong", i.e., does not violate certain conditions during its execution.

The crucial step in this approach is to extend the generator in such way that it produces all required annotations (i.e., pre-/postconditions and loop invariants) without compromising the assurance provided by the subsequent verification phase. This is achieved by embedding annotation templates into the code templates, which are then instantiated in parallel by the generator. This is feasible because the structure of the generated code and the possible safety properties are known when the generator is developed. It does not compromise the provided assurance because the annotations only serve as auxiliary lemmas and errors in the annotation templates ultimately lead to unprovable safety obligations.

We have implemented this approach and integrated it into the AUTOBAYES and AUTOFILTER program generators. We have then used it to fully automatically prove that code generated by the two systems satisfies both language-specific properties such as array-bounds safety or proper variable initialization-before-use and domain-specific properties such as vector normalization, matrix symmetry, or correct sensor input usage.

1 Introduction

Program generation has a significant potential to improve the software development process and promises many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. However, the key to realizing these benefits is of course generator correctness—nothing is gained from replacing manual coding errors with automatic coding errors. Moreover, following the motto "trust but verify" there should be some more explicit evidence for the correctness of the generated code than just trust in the correctness of the generator itself, or as has been argued, "rigorous arguments must be provided to demonstrate the correctness of the translator and/or the generated code" [WH99].

Several approaches have been explored to ensure and demonstrate correctness. In deductive program synthesis [SW+94, Kre98], the program is generated as byproduct of an existence proof for a theorem derived from the specification; other approaches based

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 17–28, 2005.

on refinement [Smi90, BG+98] or translation verification [WB96] can offer similar "correct-by-construction" guarantees. However, code generators based on these ideas are difficult to build and to scale up, and have not found widespread application. Code generators that are based on template expansion techniques are easier to build but can currently not guarantee the same level of assurance. Traditionally, they are only validated by testing, which requires significant effort that can quickly become excessive, in particular for applications in high-assurance domains like aerospace. For example, the aerospace software development standard DO-178B [RTC92] mandates that the implementation of the generator is tested to the same level of criticality the generated code requires.

We are developing and implementing an alternative approach that is not based on the verification or validation of the generator but instead focuses on the safety of each individual generated program. Our core idea is to extend the generator itself such that it produces all logical annotations (i.e., pre-/postconditions and loop invariants) that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the generated code does not "go wrong", i.e., does not violate certain conditions during its execution. The crucial aspect of the approach is to ensure that errors in the original code generator or in the certification extension do not compromise the assurance provided by the subsequent verification phase, or, in other words, that the proofs are correct and actually prove the safety properties claimed.

We have integrated this approach into the program generators AUTOBAYES [FS03] and AUTOFILTER [WS04]. We have then used it to fully automatically prove that code generated by the two systems satisfies both language-specific properties such as array-bounds safety or proper variable initialization-before-use and domain-specific properties such as vector normalization, matrix symmetry, or correct sensor input usage.

This paper summarizes our previous work on certifiable program generation. More details can be found in the cited references.

2 AUTOBAYES and AUTOFILTER

AUTOBAYES and AUTOFILTER are two domain-specific program generators that follow a schema-based approach to code generation. This extends the "plain" template expansion techniques by adding semantic constraints to the templates. AUTOBAYES [FS03] works in the scientific data analysis domain and generates parameter learning programs, while AUTOFILTER [WS04] generates state estimation code based on variants of the Kalman filter algorithm. Both systems share a large common core (e.g., symbolic subsystem, certification subsystem, and target code generators) but have their individual schema libraries. They are implemented in SWI-Prolog and together comprise approximately 100 kLoC. Both systems work fully automatically and can generate code of considerable size and complexity (approximately 1500 LoC with deeply nested loops) within a few seconds.

Schemas. A *schema* comprises a parameterized code fragment (i.e., template) together with a set of constraints that determine whether the schema is applicable and how the parameters can be instantiated. The constraints are formulated as conditions on a problem model, which allows the problem structure to directly guide the application of the

schemas and thus constrains the search space. The parameters are instantiated by the code generator, either directly on schema application or by recursive calls with a modified problem. The schemas are organized hierarchically into a *schema library* which further constrains the search space. Schemas represent both fundamental building blocks (i.e., algorithms) and solution methods (i.e., transformations) of the domain; they are thus similar to the lemmas used in purely deductive systems but they can contain explicit calls to a meta-programming kernel in order to construct code.

Symbolic Computations. Symbolic computations are used in AUTOBAYES and AUTOFILTER to support schema instantiation and code optimization. The core of the symbolic subsystem is a small rewrite engine which supports associative-commutative operators and explicit contexts. It thus allows rules as for example $x/x \rightarrow_{C \vdash x \neq 0} 1$ where $\rightarrow_{C \vdash x \neq 0}$ means "rewrites to, provided $x \neq 0$ can be proven from the current context C." Expression simplification and symbolic differentiation are implemented on top of the rewrite engine. The basic rules are straightforward; however, vectors and matrices require careful formalizations, and some rules also require explicit meta-programming, e.g., when bound variables are involved.

Intermediate Code. The code fragments in the schemas are formulated in an imperative intermediate language. This is essentially a "sanitized" variant of C (i.e., no pointers, no side effects in expressions etc.); however, it also contains a number of domain-specific constructs like vector/matrix operations, finite sums, and convergence-loops.

Optimization. Straightforward schema instantiation and composition produces suboptimal code; worse, many of the suboptimalities cannot be removed completely using a separate, after-the-fact optimization phase. Schemas can thus explicitly trigger large-scale optimizations which take into account information from the code generation process. For example, all numeric routines restructure the goal expression using code motion, common sub-expression elimination, and memoization; since the schemas know the goal variables, no dataflow analysis is required to identify invariant sub-expressions, and code can be moved around aggressively, even across procedure borders.

Target Code Generation. In a final step, the optimized intermediate code is translated into code tailored for a specific run-time environment. We currently have target code generators for the Octave and Matlab environments, and can also produce standalone Ada, C, and Modula-2 code. Each target code generator employs one rewrite system to eliminate the constructs of the intermediate language which are not supported by the target environment ("desugaring") and a second rewrite system to clean up the desugared code; most rules are shared between the different code generators.

Problem Specifications. Schema-based program generation does not necessarily require a logical conjecture as starting point for a proof. The Code derivation can therefore begin with a specification in a more application-oriented *domain-specific language*. Our specification languages combine some target language constructs (e.g., declarations) with established scientific and engineering notations (e.g., differential equations). This allows a concise and fully declarative formulation of the problem together with

some details of the desired configuration and architecture of the code to be generated. AUTOBAYES uses a specification language that is very close to the generative statistical models used in Bayesian statistics, while AUTOFILTER uses a more control engineering-oriented notation to formulate process models.

3 Certification Architecture

Our certification approach generally follows similar lines as proof carrying code (PCC) [Nec97]; in particular, the role of the extended code generator as producer of annotated target code is very similar to that of a certifying compiler [NL98, CL+00] in the PCC approach. However, there are also some key differences. First, since we target code generation instead of compilation, we work on the source code level instead of the object code level. On the positive side, since some safety properties can be formulated more naturally (e.g., initialization-before-use) or only (e.g., loop variable restrictions) on the source code level, this allows us to formulate and support more safety properties relevant to application domains. In particular, high-level domain-specific properties such as matrix symmetry or frame safety [LPR01] are inherently defined on the source code level. On the negative side, these domain-specific properties make the annotation generation (see Section 5) more difficult. Fortunately, unlike a general purpose compiler, a domain-specific code generator embodies enough domain knowledge to provide the information required. Second, the proofs are not tightly integrated into the code and are currently not even distributed together with the code; hence, our approach provides *certifiable* rather than *certifying* program generation. However, this is not a fundamental deficit and could be changed relatively straightforwardly, if necessary. Third, we apply a different prover technology and our architecture allows choosing from different off-the-shelf fully automated theorem provers (ATP) for first-order logic instead of relying on a customized higher-order system. However, the ATP can essentially be considered as a black box.

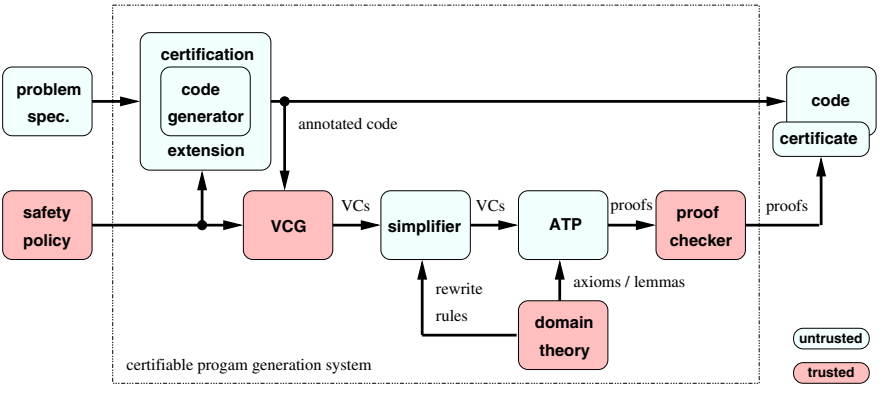


Fig. 1. Certifiable program generation: System architecture

Figure 1 shows the overall architecture of a certifiable program generation system. At its core is the original code generator which is extended for certification purposes and complemented by a verification condition generator (VCG), a simplifier, an ATP, a proof checker, and a domain theory. These components and their interactions are described in the rest of this paper and in more detail in [WSF02, DF03, DFS05]. As in the PCC approach, the architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. *Trusted* components *must be correct* because any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the assurance provided by a certifiable program generation system does not depend on the correctness of its two largest components: the original code generator (including the certification extensions), and the ATP; instead, we need only trust the safety policy, the VCG, the domain theory, and the proof checker.

4 Source-Level Safety Certification

The purpose of safety certification is to demonstrate that the code does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. It is formally defined as an entailment relation that formalizes when the evaluation of an expression and the execution of a statement are safe in a given environment. A *safety policy* is a set of proof rules and auxiliary definitions which are designed to show that safe programs satisfy the safety property of interest. The intention is that a safety policy enforces a particular safety property and strictly speaking an off-line proof is required to show the policy correct with respect to the property [DF03]. Since the calculus is sound and complete (modulo the completeness of the logic underlying the formulation of the annotations), it does in particular prevent us from proving unsafe programs safe. The proof rules can be formalized concisely using the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition P holds before and the command c terminates, then Q holds afterwards (see [Mit96] for more information about Hoare-style program proofs).

For each notion of safety which is of interest a safety property and the corresponding safety policy need be formulated. The formulation of the safety property is usually straightforward, and the proof rules for any given safety policy can fortunately be constructed systematically, by instantiating a generic rule set that is derived from the standard rules of the Hoare-calculus [DF03]. The basic idea is to extend the standard environment of program variables with a "safety environment" of "safety" or "shadow" variables which record safety information related to the corresponding program variable. The rules are then responsible for maintaining this environment and producing the appropriate safety obligations.

Figure 2 shows the rules instantiated for the relatively simple case of memory safety. Here the safety environment consists of shadow variables x_{hi} that are used to record the dimension of the corresponding arrays x. The only statement that affects the value of a shadow variable is thus the declaration of an array (cf. the *adecl*-rule). However, all rules also need to produce the appropriate safety formulas $safe_{\text{mem}}(e)$ for all immediate

subexpressions e of the statements. Since the safety property defines that an expression is safe if all access to array variables are within the bounds given by the corresponding shadow variables, $safe_{\text{mem}}(x\,[\,e\,])$ for example simply translates to $1 \leq e \leq x_{\text{hi}}$.

(*decl*) $$\frac{}{Q\ \{\textbf{var}\ x\}\ Q}$$

(*adecl*) $$\frac{}{Q[n/x_{\text{hi}}]\ \{\textbf{var}\ x\,[\,n\,]\}\ Q}$$

(*skip*) $$\frac{}{Q\ \{\textbf{skip}\}\ Q}$$

(*assign*) $$\frac{}{Q[e/x] \wedge safe_{\text{mem}}(e)\ \{x := e\}\ Q}$$

(*update*) $$\frac{}{Q[upd(x, e_1, e_2)/x] \wedge safe_{\text{mem}}(x\,[\,e_1\,]) \wedge safe_{\text{mem}}(e_2)\ \{x\,[\,e_1\,] := e_2\}\ Q}$$

(*if*) $$\frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge safe_{\text{mem}}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

(*while*) $$\frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge safe_{\text{mem}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

(*for*) $$\frac{P\ \{c\}\ I[i+1/i] \quad I \wedge e_1 \leq i \leq e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i] \wedge safe_{\text{mem}}(e_1) \wedge safe_{\text{mem}}(e_2)\ \{\textbf{for}\ i := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

(*comp*) $$\frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ ;\ c_2\}\ Q}$$

(*assert*) $$\frac{P \Rightarrow P' \quad P\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ Q'\}\ Q}$$

(*cons*) $$\frac{P \Rightarrow P' \quad P'\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P\ \{c\}\ Q}$$

Fig. 2. Proof rules for memory safety

We have defined five different safety properties and implemented the corresponding safety policies. Array-bounds safety (*array*) requires each access to an array element to be within the specified upper and lower bounds of the array. Variable initialization-before-use (*init*) ensures that each variable or individual array element has been explicitly assigned a value before it is used. Both are typical examples of language-specific properties. Matrix symmetry (*symm*) requires certain two-dimensional arrays to be symmetric. Sensor input usage (*inuse*) is a variation of the general *init*-property which guarantees that each sensor reading passed as an input to the Kalman filter algorithm is actually used during the computation of the output estimate. These two examples are specific to the Kalman filter domain. The final example (*norm*) ensures that certain one-

dimensional arrays represent normalized vectors, i.e., that their contents add up to one; it is specific to the data analysis domain.

The VCG directly implements the generic rules and, starting with the initial postcondition *true*, applies them to the statements in the usual backwards style, emitting the constructed safety obligations along the way. Since it is part of the trusted component base, it has been designed to be "correct-by-inspection", i.e., deliberately simple. Hence, it does not implement any optimizations or even apply any simplifications. Consequently, the generated obligations tend to be large and must be simplified separately before they can be tackled by the ATP. The resulting proofs can be sent to a proof checker to ensure that the ATP produced a valid proof.

5 Annotation Generation

As Figure 2 shows, the proof rules require logical annotations, in particular loop invariants. The construction of these annotations is usually the limiting factor for the practical application of Hoare-style verification tools, e.g., ESC [FL$^+$02]. Fortunately, the annotations are untrusted. They are never directly used as safety obligations themselves but only serve as lemmas for use by the trusted VCG. Due to the soundness of the calculus, an error in an annotation can, at worst, lead to the construction of an invalid safety obligation. As an example, consider the *while*-rule. If the constructed invariant I is too strong (e.g., $I \equiv \mathit{false}$), then it is easy to show that the postcondition Q follows but impossible to show that the precondition P constructed from the loop body c holds. If the invariant is too weak (e.g., $I \equiv \mathit{true}$), then it will generally be impossible to show that either the precondition or the postcondition follows, unless of course the program is trivially safe. Similar arguments hold for the other rules as well. Even compatible "parallel" errors in the generation of the code and the annotations will not compromise the assurance. In the worst case, the generated code will be functionally wrong but if the proof succeeds, it will still execute safely. This role of the annotations thus allows us to extend the untrusted code generator to produce both code *and* annotations, without compromising the assurance provided by the safety proofs.

The central question though is how this can be achieved. Obviously, there is no free lunch, and ultimately the annotations have to be provided by the developer. This can be done in the form of annotation templates that are integrated into the schemas and instantiated in parallel with the code templates by the generator. The basic process to extend the generator comprises the following four steps:

1. Analyze the generated code and identify the location and structure of the required annotations.
2. For each location, identify the schemas that produced the respective code fragment.
3. For each affected schema, generalize the respective annotations to appropriate meta-annotations (e.g., replace program variables by meta variables).
4. For each meta-annotation, formulate an appropriate annotation template or meta-program that generates the annotation at the time of schema application, and integrate it into the schema.

Like building the generator in the first place, this an expensive manual process. It has to be repeated until all schemas are covered, and started again for each new safety

property. Moreover, the annotations are cross-cutting concerns, not only on the level of the generated programs, but also on the level of the program generator. This can make the extension of the generator quite hard work. However, it remains feasible because the overall structure and the purpose of the generated code as well as the possible safety properties are already known when the generator is extended.

```
                ...
                var a[n] ; var b[m]
                ...
/* A : */       for i := 1 to n
                inv ∀j · 1 ≤ j < i ⇒ 1 ≤ a[j] ≤ m
                 do a[i] := f(i) ;
                post ∀i · 1 ≤ i ≤ n ⇒ 1 ≤ a[i] ≤ m
                ...
/* B : */       for i := 1 to n
                inv 1 ≤ a[i] ≤ m
                 do b[a[i]] := g(i) ;
                ...
```

Fig. 3. Code fragment with annotations

The code fragment shown in Figure 3, which is taken in simplified form from code generated by AUTOBAYES, illustrates how the process works. It uses the loop at A to initialize the array a with an unspecified expression $f(i)$, and then uses a in the second loop at B to write the also unspecified expression $g(i)$ indirectly into b. In order to prove these array accesses safe, the invariant needs to restrict the contents of the a-elements to the valid index range of b, i.e., $a[i]$ has to be between 1 and m.

The question is now how and where to construct this invariant. However, when we write the schema that generates the first loop, we also already know that the a-elements will be used to index into b. This is part of the domain knowledge that is required to build the original code generator. In a first step, we thus extend this schema by an annotation template or meta-program that constructs the (local) invariant and postcondition given at A. The annotation template can focus on the locally relevant information, without needing to describe all the global information that may later be necessary for the proofs because the schemas are not combined arbitrarily but only along the hierarchy given in the schema library.

Unfortunately, these local annotations are in general still insufficient to prove the postcondition at the end of larger code fragments. In our example, we still need to get the information about the values in a into the loop invariant at B. Since this limited information transport is a recurrent problem, we do not pass around the constructed annotations during generation, but rely on a separate *annotation propagation* phase after the code has been constructed. The propagation algorithm can be seen as a very crude approximation of a strongest postcondition predicate transformer. It pushes the generated local annotations forward along the edges of the syntax tree as long as the information can be guaranteed to remain unchanged. Because the generator produces

code with restricted aliasing only, the test for which statements influence which annotations can easily be accomplished without a full static analysis by maintaining a set of modified variables during propagation.

The propagation phase also adds a few default annotations as it traverses the code, for example bounds on the loop variables. These could in principle also be reconstructed by the VCG, but that would complicated the implementation of the trusted VCG. The fully annotated and propagated code is then used by the VCG.

6 Experimental Results

We have used the approach described here to certify different safety properties for code generated by AUTOBAYES and AUTOFILTER. Table 1 summarizes the relevant numbers for four representative examples. The first two examples are AUTOFILTER specifications. `ds1` is taken from the attitude control system of NASA's Deep Space One mission [WS04]. `iss` specifies a component in a simulation environment for the Space Shuttle docking procedure at the International Space Station. In both cases, the generated code is based on Kalman filter algorithms, which make extensive use of matrix operations. The other two examples are AUTOBAYES specifications which are part of a more comprehensive analysis [FH+03] of planetary nebula images taken by the Hubble Space Telescope. `segm` describes an image segmentation problem for which an iterative numerical clustering algorithm is synthesized. Finally, `gauss` fits an image against a two-dimensional Gaussian curve. This requires a multivariate optimization which is implemented by the Nelder-Mead simplex method. The code generated for these two examples has a substantially different structure from the state estimation examples. First, it contains many deeply nested loops, and some of them do not have a fixed (i.e., known at generation time) number of iterations but are executed until a dynamically calculated error value becomes small enough. In contrast, in the Kalman filter code, all loops are executed a fixed number of times. Second, all array accesses are element by element and there are no operations on entire matrices (e.g., matrix multiplication).

For each of the examples, Table 1 lists the size $|S|$ of the specification, the size $|P|$ of the generated program (including comments but without annotations), the applicable safety policies, the sizes $|A|$ and $|A^*|$ of the generated and propagated annotations, and finally the numbers N and N_{fail} of generated and invalid safety obligations as well as the generation and proof times T_{gen} and T_{proof}. All times are wall-clock times rounded to the next second and were obtained on a 2.4GHz standard Linux PC with 4GB memory. The generation times also include generation, simplification, and file output of the safety obligations; code generation alone accounts for approximately 90% of the times listed under the *array* safety policy. The proof times are based on using the E-Setheo [MI+97] prover which was able to discharge all valid obligations; they do not include the time spent on the invalid obligations. A more detailed analysis of the results achieved with different ATPs is available in [DFS05].

The table shows that the generated annotations can amount to a significant fraction of the generated code and, after propagation, can even dominate it. It also shows substantial differences in the size of the annotations required for the different safety properties; in particular, it also shows that array-bounds safety (which is the core prop-

Table 1. Results of safety certification

Example	$\|S\|$	$\|P\|$	Policy	$\|A\|$	$\|A^*\|$	N	N_{fail}	T_{gen}	T_{proof}
`ds1`	48	431	*array*	0	19	1	-	6	1
			init	87	444	74	-	11	84
			inuse	61	413	21	1	8	202
			symm	75	261	865	-	71	794
`iss`	97	755	*array*	0	19	4	-	25	3
			init	88	458	71	-	40	88
			inuse	60	361	1	1	32	-
			symm	87	274	480	-	66	510
`segm`	17	517	*array*	0	53	1	-	3	1
			init	171	1090	121	-	8	109
			norm	195	247	14	-	4	12
`gauss`	18	1039	*array*	20	505	20	-	21	16
			init	118	1615	316	-	54	259

erty guaranteed by PCC) requires almost no local annotations and can often be certified with only the default annotations added by the propagator. The number of generated safety obligations also varies substantially for the different safety properties. However, the proof effort remains tractable, and in most of the cases the ATP was able to successfully discharge all obligations in less than 15 minutes wall clock time.

In general, of course, an obligation can fail to be proven for a number of reasons. First, there may of course be an actual safety violation in the code. This is the case for the two invalid obligations that are produced for the sensor input usage property. The deeper reason for this, however, is not a flaw in the code generator but a sloppy specification that declares a vector that is not completely used. Second, the (generated) annotations may be insufficient or wrong. Annotation errors can come from any part of the schema, or from the propagation phase: an annotation might not be propagated far enough, or it might be propagated out of scope. Third, the theorem prover may time-out, either due to the size and complexity of the obligation, or due to an incomplete domain theory. For certification purposes, however, it is important to distinguish between unsafe programs and any other reasons for failure, and in the case of genuine safety violations, to locate the unsafe parts of the program.

7 Conclusions

We have described an extension to the AUTOBAYES and AUTOFILTER program generators which can automatically ensure important safety properties for the generated code. The core idea of our approach is to extend the generator itself in such way that it produces all logical annotations (i.e., pre-/postconditions and loop invariants) required for Hoare-style safety proofs without compromising the assurance provided by the proofs. In principle, the prover can fail to prove some valid proof obligations and thus raise false alarms but in practice we were able to design the system such that all valid obligations could be discharged fully automatically.

Our approach can be seen as "PCC for code generators" because it enables safety proofs for the generated code. We believe that it can directly be applied to code generators based on template expansion techniques in general, not only to our own systems. However, we believe also that our techniques could as well be used in a response to the recently announced Grand Challenge of developing a verifying compiler. We further believe that in principle any verification technique that can be guided by an appropriate form of annotations can be combined successfully with a certifiable code generator, not just the Hoare-style certification using the VCG/ATP combination described here. Another interesting research direction would thus be to combine annotation generation with other techniques, for example static analysis.

For future work, we plan to extend the system in two main areas, in addition to continually increasing the systems' generative power with more algorithmic schemas, more specification features, and more control over the derivation.

First, we are developing a more declarative and explicit modeling style. Much of the domain knowledge used by the system in deriving code is currently implicit; by making it explicit this can be used to (among other things) facilitate traceability between the code and it derivation in the generated documentation.

Second, we continue to extend the certification power of the system with more policies, more automation, and an integrated approach to documentation generation. In particular, we are now developing an "annotation inference" technique which addresses many of the difficulties of annotation generation. This will also enable us to easily apply our certification techniques to code generators other than our own.

Acknowledgements. Mike Whalen and Johann Schumann contributed substantially to the development and implementation of the certifiable program generation approach.

References

[BG+98] L. Blaine, L.-M. Gilham, J. Liu, D. R. Smith, and S. Westfold. "Planware – Domain-Specific Synthesis of High-Performance Schedulers". In D. F. Redmiles and B. Nuseibeh, (eds.), *Proc. 13th Intl. Conf. Automated Software Engineering*, pp. 270–280. IEEE Comp. Soc. Press, 1998.

[CL+00] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. "A certifying compiler for Java". In *Proc. ACM Conf. Programming Language Design and Implementation 2000*, pp. 95–107. ACM Press, 2000. Published as SIGPLAN Notices 35(5).

[DF03] E. Denney and B. Fischer. "Correctness of Source-Level Safety Policies". In K. Araki, S. Gnesi, and D. Mandrioli, (eds.), *Proc. FM 2003: Formal Methods*, *Lect. Notes Comp. Sci.* **2805**, pp. 894–913. Springer, 2003.

[DFS05] E. Denney, B. Fischer, and J. Schumann. "An Empirical Evaluation of Automated Theorem Provers in Software Certification". *International Journal of AI Tools*, 2005. To appear.

[FH+03] B. Fischer, A. Hajian, K. Knuth, and J. Schumann. "Automatic Derivation of Statistical Data Analysis Algorithms: Planetary Nebulae and Beyond". In G. Erickson and Y. Zhai, (eds.), *Proc. 23rd Intl. Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering*, pp. 276–291. American Institute of Physics, 2003.

[FL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. "Extended static checking for Java". In L. J. Hendren, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 2002*, pp. 234–245. ACM Press, 2002. Published as SIGPLAN Notices 37(5).

[FS03] B. Fischer and J. Schumann. "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models". *J. Functional Programming*, **13**(3):483–508, 2003.

[Kre98] C. Kreitz. "Program Synthesis". In W. Bibel and P. H. Schmitt, (eds.), *Automated Deduction — A Basis for Applications*, pp. 105–134. Kluwer, 1998.

[LPR01] M. Lowry, T. Pressburger, and G. Rosu. "Certifying Domain-Specific Policies". In M. S. Feather and M. Goedicke, (eds.), *Proc. 16th Intl. Conf. Automated Software Engineering*, pp. 118–125. IEEE Comp. Soc. Press, 2001.

[MI+97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. "The Model Elimination Provers SETHEO and E-SETHEO". *J. Automated Reasoning*, **18**:237–246, 1997.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[Nec97] G. C. Necula. "Proof-Carrying Code". In *Proc. 24th ACM Symp. Principles of Programming Languages*, pp. 106–19. ACM Press, 1997.

[NL98] G. C. Necula and P. Lee. "The Design and Implementation of a Certifying Compiler". In K. D. Cooper, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pp. 333–344. ACM Press, 1998. Published as SIGPLAN Notices 33(5).

[RTC92] RTCA Special Committee 167. Software Considerations in Airborne Systems and Equipment Certification. Technical report, RTCA, Inc., December 1992.

[Smi90] D. R. Smith. "KIDS: A Semi-Automatic Program Development System". *IEEE Trans. Software Engineering*, **16**(9):1024–1043, 1990.

[SW+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. "Deductive Composition of Astronomical Software from Subroutine Libraries". In A. Bundy, (ed.), *Proc. 12th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* **814**, pp. 341–355. Springer, 1994.

[WB96] V. L. Winter and J. M. Boyle. "Proving Refinement Transformations for Deriving High-Assurance Software". In *Proc. High-Assurance Systems EngineeringWorkshop*, pp. 68–77. IEEE Comp. Soc. Press, 1996.

[WH99] M. Whalen and M. Heimdahl. "On the Requirements of High-Integrity Code Generation". In *Proc. 4th Intl. Symp. High-Assurance Systems Engineering*, pp. 216–226. IEEE Comp. Soc. Press, 1999.

[WS04] J. Whittle and J. Schumann. "Automating the Implementation of Kalman Filter Algorithms". *ACM Transactions on Mathematical Software*, **30**(4):434–453, 2004.

[WSF02] M. Whalen, J. Schumann, and B. Fischer. "Synthesizing Certified Code". In L.-H. Eriksson and P. A. Lindsay, (eds.), *Proc. Intl. Symp. Formal Methods Europe 2002: Formal Methods—Getting IT Right*, *Lect. Notes Comp. Sci.* **2391**, pp. 431–450. Springer, 2002.

A Generative Programming Approach to Developing DSL Compilers

Charles Consel[1], Fabien Latry[1], Laurent Réveillère[1], and Pierre Cointe[2]

[1] INRIA / LaBRI, F-33402 Talence, France
[2] École des Mines de Nantes, F-44070 Nantes, France

Abstract. Domain-Specific Languages (DSLs) represent a proven approach to raising the abstraction level of programming. They offer high-level constructs and notations dedicated to a domain, structuring program design, easing program writing, masking the intricacies of underlying software layers, and guaranteeing critical properties.

On the one hand, DSLs facilitate a straightforward mapping between a conceptual model and a solution expressed in a specific programming language. On the other hand, DSLs complicate the compilation process because of the gap in the abstraction level between the source and target language. The nature of DSLs make their compilation very different from the compilation of common General-Purpose Languages (GPLs). In fact, a DSL compiler generally produces code written in a GPL; low-level compilation is left to the compiler of the target GPL. In essence, a DSL compiler defines some mapping of the high-level information and features of a DSL into the target GPL and underlying layers (*e.g.,* middleware, protocols, objects, ...).

This paper presents a methodology to develop DSL compilers, centered around the use of generative programming tools. Our approach enables the development of a DSL compiler to be structured on facets that represent dimensions of compilation. Each facet can then be implemented in a modular way, using aspects, annotations and specialization. Because these tools are high level, they match the needs of a DSL, facilitating the development of the DSL compiler, and making it modular and re-targetable.

We illustrate our approach with a DSL for telephony services. The structure of the DSL compiler is presented, as well as practical uses of generative tools for some compilation facets.

1 Introduction

"Generative software development aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages [7]".

At the design level, generative software development emphasizes the role of Domain-Specific Languages (DSLs) as a way to bridge the gap between high-level modeling and General-Purpose Languages (GPLs). Nevertheless, from the

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 29–46, 2005.

programming language viewpoint, there is a lack of methodology and a lack of tools to support the development of DSL compilers. The purpose of this paper is to apply some of the mainstream techniques promoted by the generative programming community to develop DSL compilers.

What is a DSL? From a programmer viewpoint, a DSL is typically created to model a program family [4]. The commonalities and the variabilities found in the target program family suggest abstractions and notations that are domain specific [6]. In contrast with GPLs such as Java or C++, a DSL has a narrow application scope and must be readable for domain experts.

In general, a program in a textual DSL is a concise set of high-level declarations, focusing on what to compute, as opposed to how to compute it. As an illustration, consider the telephony domain, and more specifically service creation. Conceptually, telephony services represent variations in creating, modifying and terminating a communication between parties. While this program family in a GPL covers the full implementation of services, its counterpart in a DSL corresponds to variations of service logic, abstracting over implementation details.

From a language designer viewpoint, a DSL makes domain-specific information an integral part of the programming paradigm. As such the programmer can be viewed as being prompted by the language to provide domain-specific information. This information may take the form of domain-specific types, syntactic constructs and notations. It serves domain-specific concerns, such as library interfacing, optimization, instrumentation, profiling and verification of the generated code. As of now, there are neither specific methodology, nor dedicated support tools, well suited to handle the compilation of both the high-level nature of a DSL and the richness of the built-in domain-specific information.

Challenges in DSL compilation. When mapping a DSL to a GPL, the higher level the DSL is, the more program generation is needed to bridge the gap with the target execution environment. Concretely, one program line written in a DSL commonly compiles into many lines in GPL. For instance, we have developed a DSL for telephony services, named SPL [2], that compiles into Java. An SPL program is on average 4 times more concise than its Java counterpart.

Not surprisingly, GPL-translated programs include rather large program templates. The process of generating these templates can be quite complex, relying on various conditions, and requiring a number of instantiations by computing and inserting constants. Without any dedicated tool support, this process can be quite laborious and error-prone. The resulting generator is often cumbersome and hard to debug. Additionally, the lack of tool support makes it difficult to have a modular treatment of the domain-specific concerns exposed by a program.

This paper. We propose a methodology to develop DSL compilers. The key idea of this methodology is to rely on generative programming tools [8]. These tools enable modeling the high-level nature of DSLs and the richness of the built-in domain-specific information in terms of program generation. This modeling can be done in the context of various generative programming approaches. Our

methodology is composed of two steps: compiling program logic and performing generative programming.

First, the DSL program logic is translated into an abstract GPL representation. This representation is abstract because it includes operations whose interpretation is not necessarily defined yet; it may thus not be executable. Although abstract, this translation generates a representation that encodes domain-specific information and that is localized in code regions of interest for further compilation treatment. In doing so, the representation is amenable to generative tools that represent the second step of our methodology.

Generative programming tools are used to define the compilation of domain-specific facets of a DSL in terms of program generation processes. One category of facets addresses the mapping of a DSL program into a target execution environment. A second category of facets is devoted to the compilation of language abstractions. A third category of facets defines code generation processes that are specific to the subject DSL program. This approach modularizes the process of generating a DSL compiler. Furthermore, each generative programming approach provides a paradigm, associated abstractions and tools dedicated to a specific family of program generation. The compiler developer can thus choose the most appropriate generative programming approach for a given facet.

Our approach is illustrated by three generative programming approaches, namely AOP [13], annotations [3], and program specialization [5,10]; conceptually other approaches can be considered. AOP is well-suited to introduce crosscutting behaviors in GPL-translated programs (*e.g.,* prologue and epilogue code of API invocations). Annotations enable non-functional concerns to be introduced in the compilation process (*e.g.,* resource management). Program specialization can address optimization-oriented program generation (*e.g.,* customization of software components).

Refining facets further leads us to distinguish between functional and non-functional facets. On the one hand, the functional facets define program generation processes that make the GPL-translated program executable. On the other hand, non-functional facets enrich the language execution in terms of requirements like performance, reliability and security.

Contributions.

- **A methodology to develop DSL compilers.** We define a methodology to develop a DSL compiler. We introduce a GPL-translated abstract representation of the program logic. This representation is structured so that further compilation can produce a spectrum of implementation variants.
- **A novel use of generative programming tools.** Our methodology relies on a novel use of generative programming. This methodology facilitates the generative programming process required by DSL compilation: high-level generative programming paradigms can be used to modularly process domain-specific information and abstractions.
- **A case study.** We use our methodology to develop a compiler for a DSL dedicated to the creation of telephony services. We present aspects, annotations

and specialization opportunities that model various compilation dimensions of this DSL.

Outline. The rest of this paper is organized as follows. Section 2 introduces our case study, a DSL for telephony services. Section 3 presents our methodology to develop a DSL compiler. Section 4 focuses on compiling the program logic. Section 5 and 6 presents the compilation of functional and non-functional language units. Finally, Section 7 discusses our approach, and Section 8 concludes.

2 Case Study

Our approach to developing DSL compilers is illustrated by a DSL to create telephony services. We choose the Session Initiation Protocol (SIP) [14,15] as the underlying signaling protocol. This protocol is used for Voice over IP (VoIP) and third generation mobile phones. It is standardized by the IETF[1] and adopted by the ITU.[2]

SIP is a rapidly emerging protocol that places telephony into mainstream computer science. It is combined with a number of existing protocols to handle various aspects of communications (*e.g.,* transport and real-time streaming). It relies on the client-server model. SIP platforms provide rich programming interfaces in languages like C# and Java. The general-purpose nature of these programming interfaces make them very large and intricate to use. This situation is demonstrated by JAIN SIP, a standardized Java interface to SIP [9,16], which consists of 130 classes and more than 3000 methods.

The need for a DSL in this domain stems from three main reasons. First, modern telephony is a software intensive area because of the host of new functionalities it offers. Second, programming telephony services now requires extensive expertise in SIP and its companion protocols, distributed programming, networking, and SIP programming interfaces. Third, a study of existing telephony services shows that programming a service logic in a given platform with a GPL is quite a laborious and error-prone process [1]. It requires recurring code patterns as the prologue and epilogue of invocations of the SIP programming interface.

We have developed a DSL named Session Processing Language (SPL) that enables telephony services to be defined concisely, using high-level abstractions and notations [2]. SPL enables the programmer to concentrate on the service logic, abstracting over low-level intricacies such as the protocol details and the underlying platform programming interface.

An SPL program defines a telephony service to which users can subscribe. The *session* is a key notion in SPL; it structures the development of a telephony service. A session consists of a set of handlers and a state. A handler defines a treatment for a protocol request or a platform event. A handler may be omitted, if no service logic is associated with it. A state allows some data to be maintained across a set of handlers. SPL offers different kinds of sessions that form

[1] IETF: Internet Engineering Task Force.
[2] ITU: International Telecommunications Union.

```
1   service Counter {
2     processing {
3       local void log (int);
4
5       registration {
6         int count;
7
8         response outgoing REGISTER() {
9           count = 0;
10          return forward;
11        }
12
13        void unregister() {
14          log (count);
15        }
16
17        dialog {
18          response incoming INVITE() {
19            response resp = forward;
20            if (resp != /SUCCESS) {
21                return forward 'sip:secretary@company.com';
22            } else {
23              count++;
24              return resp;
25            }
26  }}}}}
```

Fig. 1. The counter service in SPL

a hierarchy. A simple counter service written in SPL, exhibiting this hierarchy, is displayed in Figure 1.

The innermost session is the *dialog*: it manages a communication between parties. A dialog is created by the `INVITE` request, confirmed by the `ACK` request, and terminated by the `BYE` request. A dialog session defines handlers for the SIP requests and platform events pertaining to the communication management. In the counter service, an `INVITE` handler is defined to process incoming calls. This handler systematically *forwards* (*i.e.,* routes) a call to the user who subscribed to this service. If this forward does not succeed (*e.g.,* the user is busy), the call is forwarded to a secretary. Otherwise, the call is accepted, a counter is incremented, and the response is returned to the caller. The session variable `count` used in this handler is defined in the session surrounding a dialog, namely the *registration* session. Such a session is created for each user of the counter service that registers on the SIP platform by sending the `REGISTER` request. This session defines a state that consists of a unique variable (`count`), initialized in the `REGISTER` handler. A user is unregistered either when his registration lease expires or when the `REGISTER` request contains a zero-lease. Both situations are viewed by SPL as a unique event named `unregister`, for which a handler can be defined in the SPL program. In our example, the `unregister` handler invokes a function that logs the counter, when the session is terminated. At the top of the session hierarchy is the *service* session. Such a session is created when the service is deployed on the platform and deleted when it is undeployed.

SPL abstracts over a number of protocol and platform issues among which is the statefulness of a transaction. Let us explain this issue because it illustrates the gap between SPL and a SIP platform. This issue will later be used in examples. A transaction consists of a request and the response it triggers. Exist-

ing SIP platforms provide separate interface entries for processing requests and responses. A telephony program may process a request and its response independently; such a transaction is said to be *stateless*. Alternatively, a telephony program may need to match a response with the original request to continue some treatment initiated when the request got processed. Such a transaction is said to be *stateful* because it requires the platform to retain enough information to link a response to the original request. SPL abstracts over these implementation details. As shown in the `INVITE` handler, when the `INVITE` request is forwarded, the response is treated within the same handler. Statefulness of the transaction is determined by analyzing the service. In fact, SPL offers statefulness throughout the session hierarchy. In our example, the `count` variable is used in both the registration and dialog sessions. When a SIP message is processed by an SPL program, its GPL-translated version executes some code to trigger the corresponding handler and to extract the appropriate state.

3 A Methodology to Develop DSL Compilers

Our starting point in developing a DSL compiler is to define a direct translation of the *program logic* into a GPL representation. This translation is direct in that it generates abstract GPL code, not necessarily executable, where DSL mechanisms are not yet expanded but rather left uninterpreted. Although abstract, this translation encodes domain-specific information into the GPL representation, in a context where it can later be interpreted.

In our telephony case study, for example, while the routing of a SIP message in SPL is not concerned with statefulness, this property is explicitly encoded in the GPL-translated version.

The GPL-translated program logic can then be the input to various interpretations guiding two compilation dimensions: *functional* and *non-functional language units*. On the one hand, the functional units of the language are intended to complement the GPL-translated program logic to make it executable. On the other hand, the non-functional units of the language address implementation refinements or enrichments. The compilation of functional units varies with respect to both the target GPL and the target execution environment. In addition to these dimensions, the compilation of non-functional units may vary with respect to requirements like performance, reliability and security.

Whether or not functional, the compilation of DSL units can be decomposed into treatments that are inherent to either the target execution environment, the language or the program. These three categories of treatment are named *facets*.

The goal of the *execution environment facet* is to bridge the gap between the DSL execution model, possibly implicit, and the target execution environment. The DSL execution model is supposed to be high level and portable, abstracting over the intricacies of the target execution environment. The execution environment facet is intended to generate the necessary code to interface the GPL-translated program logic with the underlying layers. Considering the SPL case, the execution environment facet bridges the gap between the explicit

event-handling architecture of JAIN SIP and the implicit SPL model based on request handlers. For example, default request handlers need to be introduced to create and delete session states, whenever an SPL program does not define handlers for the corresponding requests.

The *language facet* is concerned with the interpretation and expansion of language mechanisms, whether or not explicit in the GPL-translated program logic. Such a facet generates recurring code patterns intrinsic to the language. For example, routing operations of all SPL services need to be analyzed to determine their statefulness. This information is made explicit in the GPL-translated program logic by adding some information in each invocation of the routing operations.

Lastly, the *program facet* invokes compiler treatments that are specific to the subject program. For instance, in the case of SPL, the request handler necessitates specific state extraction operations, depending on what session variables occur. Such a treatment is specific to a given SPL service and thus part of the program facet.

4 Compiling the Program Logic

The goal of compiling the logic of a program is to produce a representation that abstracts over implementation details while being amenable to generative programming tools. To be applied successfully, generative programming tools generally introduce some constraints on the program structure and may require program generation parameters. Program structure is needed to make the GPL-translated program logic match the granularity of the program entities manipulated by the generative programming tools. The program generation parameters are typically needed to customize the program generation process or the execution generated program. These parameters may correspond to compilation data that are domain-specific information and are encoded in the translated program or/and in generative programming declarations (*e.g.,* an aspect declaration).

Our approach enables the program generation process to be modularized: each module is responsible for a "slice" of program generation needed to compile a given DSL. This modularization is particularly well-fitted to explore the implementation scope offered by the high-level nature of a DSL.

An example of a GPL-translated program logic is displayed in Figure 2. It is the SPL program (showed in Figure 1) compiled into Java for a JAIN SIP interface [16]. This interface is typical of a client-server model in that it requires a telephony service to implement a `SipListener` interface, providing `processRequest` and `processResponse` methods. Additionally, a method is declared to handle various platform timeouts. Also, private methods have been introduced to handle the registration and un-registration of the service owner, as well as the forwarding of call invitations (responses are treated in `processResponse`). As can be noticed in `handler_INVITE`, for example, the service logic has been straightforwardly translated into Java: expressions manipulating variables are reproduced verbatim; message routing is translated into invocations of `sendRequest` for the SPL `forward` and `sendResponse` for the returning responses.

```
1  public class Counter implements SipListener {
2    [...]
3    private void handler_REGISTER (Request rq, String method) {
4      count = 0;
5      sendRequest (false, rq_request);  /**  SPL, line 10  **/
6    }
7    private void handler_unregister (Request rq, String method) {
8      local.log (count);  /**  SPL, line 14  **/
9      sendRequest (false, rq_request);  /**  SPL: default behavior  **/
10   }
11   private void handler_INVITE (Request rq, String method) {
12     sendRequest (true, rq_request);  /**  SPL, line 19  **/
13   }
14
15   public void processRequest (RequestEvent requestEvent) {
16     String method = rq_request.getMethod();
17     if (method.equals (Request.REGISTER)) {  /**  SPL, line 5  **/
18       if (!registrar.hasExpiresZero (rq_request)) {
19         if (!registrar.hasRegistration (rq_request)) {  /**  SPL, line 8  **/
20           handler_REGISTER (rq_request, method);
21         } else {  /**  SPL: default behavior  **/
22           sendRequest (false, rq_request);
23         }
24       } else if (registrar.hasRegistration (rq_request)) {  /**  SPL, line 13  **/
25         handler_unregister (rq_request, method);
26       } else {  /**  SPL: default behavior  **/
27         sendRequest (false, rq_request);
28       }
29     } else if (method.equals (Request.INVITE)) {  /** SPL, line 18  **/
30       handler_INVITE (rq_request, method);
31     } else {  /**  SPL: default behavior  **/
32       sendRequest (false, rq_request);
33     }
34   }
35
36   public void processResponse (ResponseEvent responseEvent) {
37     String method = rs_request.getMethod();
38     rs_responseCode = rs_response.getStatusCode();
39     if (method.equals (Request.INVITE)) {  /**  SPL, line 18  **/
40       if (rs_responseCode >= 300) {  /**  SPL, line 20  **/
41         AddressFactory addressFactory = getAddressFactory();
42         SipURI sipURI = addressFactory.createSipURI ("secretary", "company.com");
43         rs_request.setRequestURI (sipURI);
44         sendRequest (false, rs_request);  /** SPL, line 21  **/
45       } else {
46         count++;
47         sendResponse (true, rs_response);  /**  SPL, line 24  **/
48       }
49     } else {  /**  SPL: default behavior  **/
50       sendResponse (false, rs_response);
51     }
52   }
53
54   public void processTimeout (TimeoutEvent timeoutEvent) {[...]}
55 }
```

Fig. 2. Java-translated program logic

Let us now examine how to compile the program logic using generative programming approaches.

4.1 Aspect-Oriented Programming

The AOP approach consists of a join point language, to identify locations in the program execution, and an advice language, to define additional behavior at these locations [11].

Our goal is to produce a GPL-translated program logic that matches the expressivity of the join point language. That is, to structure the translated program so that it matches the granularity of the join point identification and the semantics of the associated advice. Conceptually, to fit the AOP approach, DSL compilation must introduce specific code structuring and communicate compilation data.

The GPL-translated program logic needs some structuring when some code is to be inserted at some program point. Identifying the target program point requires language entities such as a method definition or invocation, variable occurrences, and declarations. As a result, a sequence of commands may need to be placed into a method in a translated program to enable code to be inserted before, after, or around its execution. The top of Figure 2 shows private methods that have been introduced by the DSL compiler to easily insert code as the prologue and epilogue of invocations of SPL request handlers.

Compiling the program logic also involves collecting or computing information about the DSL program that needs to be made explicit in the translated program. This strategy separates the production of the information and its exploitation, leaving its interpretation to other compilation passes. In the context of AOP, passing information can either be made by adding extra arguments to operations or by introducing specific instance variables. The latter case is further discussed in Section 5.2.

In principle, functional DSL units correspond to aspects that can performed statically since, intuitively, these aspects refine the static compilation process. In contrast, non-functional units may compile into both static and dynamic aspects. Static aspects may be used to expand the implementation of specific operations, whereas dynamic aspects may define conditional monitoring actions.

A key advantage of our approach is to permit a DSL compiler to be designed and structured in terms of modules, that is, aspects. Each aspect defines a specific DSL behavior whose cross-cutting nature makes it an ideal target for AOP.

4.2 Annotations

Annotations can be included in the translated program to make some information explicit as the DSL program gets mapped into a lower-level representation. Annotations are traditionally used as extra information describing non-functional language issues.

Like aspects, annotations modularize the compilation of a DSL in that they introduce information that are later interpreted with respect to a given annotation processor.

Different sets of annotations have different goals. There may be annotations to make resource management explicit, or annotations intended to trigger check pointing of some state. When compiling the program logic, the aim is to generate annotations that are as self-contained as possible to minimize the work of the annotation processor occurring at a later phase. To do so, the location of an annotation is a key parameter. Also, arguments to the annotation may be needed as additional input to the annotation process.

4.3 Program Specialization

Program specialization is another use of a generative programming tool for DSL compilation. It addresses the optimization of DSL implementation building blocks. The idea is that a DSL can often be seen as a language gluing software components together. Because these components can be glued in a variety of contexts they must be highly generic. While this approach has obvious software engineering advantages, in practice, it may entail a significant performance penalty. To alleviate this problem, program specialization enables generic software components to be customized with respect to the context in which they are used. Because software components are invoked by compiler-generated code, the customization contexts can be determined by definition of the DSL compiler. Furthermore, since components are fixed, or slowly evolving, their *specializability* can be determined precisely. As a result, this program transformation can be made fully predictable, which is not the case for arbitrary program transformations like most program optimizations.

5 Compiling Functional Units

Compilation of functional units fits particularly well with AOP: it incrementally refines the semantics of language mechanisms, whether or not explicit in the source program. We do various forms of code expansion triggered by method names and instance variables.

We use a wide-spread and well-tested AOP tool, AspectJ [12], to define the compilation of various functional units needed for SPL. Limitations discussed in the following examples are not intrinsic to the AOP approach but are rather specific to AspectJ tool. These limitations are further discussed in Section 7.

5.1 Execution Environment Facet

The goal of functional execution environment facets is to bridge the gap between the DSL execution model and the target execution environment. In our case study, these facets are intended to generate the necessary code to interface the program logic with the underlying JAIN SIP platform.

```
public aspect Environment {

  public Request Counter.rq_request;
  public SipProvider Counter.rq_sipProvider;

  pointcut processRequest(): execution(public void Counter.processRequest (RequestEvent));
  before(RequestEvent rq_Event, Counter obj): processRequest() && args(rq_Event) && target(obj) {
    obj.rq_request = rq_Event.getRequest();
    obj.rq_sipProvider = (SipProvider) rq_Event.getSource();
  }
  [...]
}
```

Fig. 3. An aspect for an execution environment facet

The aspect program presented in Figure 3 introduces code that implements the SPL model in terms of the JAIN SIP event-handling architecture. To do so, the inserted code extracts the current request message from the `event` object generated by JAIN SIP. Dispatch over the type of request can then be done to invoke the appropriate Java-translated SPL handler. In addition, code to extract the `sipProvider` object is generated to enable further processing of the SIP message (*e.g.,* message headers and transaction creation).

Thanks to the functional execution environment facet presented above, the Java-translated program logic does not have to deal with the intricacies of the underlying JAIN SIP platform. Such a strategy helps to isolate target-dependent program generation in compiler modules, corresponding to aspects.

5.2 Language Facet

Functional language facets are concerned with the interpretation and expansion of language mechanisms. In our case study for example, state management and statefulness of routing operations require generating recurring code patterns. The corresponding aspects are discussed below.

State management. As illustrated in Figure 2, the Java-translated program logic does not include code for attaching a state to a session, and managing this state across the session life-cycle. For example, the state associated with a registration session needs to be created when processing a `REGISTER` request and deleted either when the request failed or the session ends. We have developed a language facet in AOP dedicated to state management. An excerpt of this facet is depicted in Figure 4.

The first advice specifies the code to execute for creating a registration state when processing a `REGISTER` request. However, since the handler for the `REGISTER` request is not mandatory in SPL, the corresponding method may not be present in the Java-translated program. If so, some code must be inserted to catch a `REGISTER` request and create the registration state. However, the pattern matching capability of the pointcut language does not permit to test the non-existence of a method invocation. To remedy this problem, this information is encoded as flags introduced into the aspect program. These flags enable the appropriate advice to be selected, as illustrated by the use of the `handler_REGISTER` flag.

Statefulness. In the Java-translated program logic, the first argument of the `sendRequest` method determines the statefulness of the routing operation. This information has been made explicit in the Java-translated program logic thanks to an analysis of the SPL program. The aspect program shown in Figure 5 describes the recurrent code fragment that needs to be executed when processing such an operation. In this example, calls to `sendRequest` are compiled into either stateless or stateful routing operations depending on the value of its first argument.

The use of functional language facets enables the translated program to be closer to the program logic. For example, state management requires a precise

```
public aspect Language {

  private boolean handler_REGISTER   = true;
  private boolean handler_REREGISTER = false;
  private boolean handler_unregister = true;

  pointcut register(): execution(private void Counter.handler_REGISTER (Request, String));
  before(Request rq, String method, Counter obj): register() && args(rq, method) && target(obj) {
    State state = new State();
    int ident   = obj.env.getId (obj.rq_request);
    obj.env.setEnv (ident, state);
    obj.state = state;
  }

  pointcut processRequest(): execution(public void Counter.processRequest (RequestEvent));
  before(Counter obj): processRequest() && target(obj) {
    if (!handler_REGISTER) {
      String method = obj.rq_request.getMethod();
      if (method.equals (Request.REGISTER)) {
        if (!obj.registrar.hasExpiresZero (obj.rq_request)) {
          if (!obj.registrar.hasRegistration (obj.rq_request)) {
            State state = new State();
            int ident   = obj.lib.env.getId (obj.rq_request);
            obj.lib.env.setEnv (ident, state);
            obj.state = state;
  }}}}}
  [...]
}
```

Fig. 4. An aspect for a language facet (1)

```
public aspect Language {
  [...]
  pointcut rq_sendRequest(): call(public void Counter.sendRequest (boolean,Request)) &&
    (withincode(public void Counter.processRequest (..)) || withincode(* Counter.handler_* (..)));
  void around(boolean b, Request r, Counter obj): rq_sendRequest() && args(b,r) && target(obj) {
    try {
      if (b) {
        ClientTransaction ct = obj.rq_sipProvider.getNewClientTransaction (r);
        ct.sendRequest();
        return;
      } else {
        obj.rq_sipProvider.sendRequest (r);
        return;
      }
    } catch (Exception ex) {}
  }
  [...]
}
```

Fig. 5. An aspect for a language facet (2)

understanding of the SIP protocol to determine when sessions need to be created and deleted. Such intricacies are factorized into the language facet, preventing the compiler writer to address all the DSL implementation issues at once.

5.3 Program Facet

Functional program facets are concerned with the generation of code that is specific to a program. For example, creation and manipulation of the state attached to a session depends on the session variables and the handlers that use them.

Such a situation is illustrated by the SPL program shown in Figure 1, where the `count` variable is used in the `REGISTER` handler (line 9). This variable occur-

```
public aspect Program {

  public int Counter.count;
  public class State implements Lib.State {
    private int count;
    void setCount (int x) { count = x; };
    int getCount () { return count; };
  }

  pointcut set_count(): set (int Counter.count);
  void around(int count, Counter obj): set_count() && args(count) && target(obj) {
    obj.state.setCount(count);
  }

  pointcut get_count(): get (int Counter.count);
  int around (Counter obj): get_count() && target(obj){
    return obj.state.getCount();
  }
}
```

Fig. 6. An aspect for a program facet

rence leads to the declaration of an aspect, displayed in Figure 6, that inserts the class definition `State` and the methods to access the instance variable. Furthermore, this aspect specifies that each occurrence of the `count` variable in the Java-translated program logic must be replaced by an access to the state.

Functional program facets allow defining DSL compilation at the granularity of a program, using implicit or explicit information from the source program.

6 Compiling Non-functional Units

Non-functional DSL units are compiled by exploiting information that refines or extends the resulting program implementation. Just like functional units, compilation of non-functional units cover all of the DSL facets, that is, execution environment, language and program.

6.1 Execution Environment Facet

Program specialization [5,10] has been successfully used to customize an execution environment according to a specific usage context. In our case study, the JAIN SIP platform is responsible for invoking specific methods in the Java-translated program logic for processing requests and responses. When no method is defined in the Java program, the platform does not need to pass the SIP message to the SPL service. Instead, it can directly perform the default platform behavior. Such a filtering of messages is similar to packet filtering in networking where only packets of interest are channeled to the application layer. Program specialization has already been applied in this context [18] and has demonstrated its benefits. In our case study, a similar strategy would aim to specialize the message filtering of the JAIN platform with respect to request names of interest to a program logic.

This approach enables automatically and systematically specializing highly generic software components, such as JAIN SIP components, according to a

customization context derived from the program logic, such as an SPL service. Beyond performance, specialization opens opportunity to reduce the foot-print of a software layer.

6.2 Language Facet

Chander *et al.* [3] have recently proposed an approach to resource-bounds checking. Their approach limits the resource usage accordingly to a policy, specifying the resources that a program can use, along with the corresponding usage bounds. The key idea is to ensure that for each operation that consumes resources, an adequate amount is still available. They propose to annotate a program with a *consume* e command specifying that e units of resource are used, and with an *acquire* e command specifying that e units of resources must be reserved. One of the advantages of this approach is to make it possible to use a theorem-prover to prove that adequate checks are being performed to guarantee correct resource usage for a given program. The verifier is composed by two components: a safety condition generator that extracts logical predicates (safety conditions) whose validity implies resource-usage safety and a prover that proves the predicates.

Figure 7 illustrates the use of this approach. The left of this figure shows an SPL program where a list of *callers* is defined: three persons authorized to contact the service owner. If not authorized, a call is redirected to a list of four *operators* until one of them picks up the phone. In the telephony domain, the forwarding action represents a critical resource because it triggers a chain of operations that may be costly in the telephony platform.

The Java-translated program logic is displayed in the right of Figure 7. Annotations have been introduced for routing operations and the loop command.

```
service limit_forward {
  processing {
    uri<4> operators = <...>;
    uri<3> callers = <...>;

    registration {
      dialog {
        response incoming INVITE() {
          foreach (caller in callers) {
            if (FROM == caller)
              return forward;
          }
          return forward operators;
}}}}}
```

```
public void processRequest(RequestEvent requestEvent) {
 [...]
  if (method.equals (Request.INVITE)) {
   //@ acquire 4
   Header f_h = rq_request.getHeader(FromHeader.NAME);
   String FROM = f_h.toString();
   int i = 0;
   int size = callers.size();
   while (i < size) {
    String caller = (String)callers.get(i);
    if (FROM.compareTo (caller) == 0) {
      //@ consume 1
      rq_sipProvider.sendRequest(rq_request);
      return;
    }
    i++;
   }
   //@ inv(i <= 3, 3 - i)
   //@ consume 4
   rq_sipProvider.sendRequest(operators, rq_request);
   return;
  }
}
[...]
```

Fig. 7. Annotations for a language facet

Through annotation analysis, we can determine that this example requires, at worst, the reservation of four resource units (*acquire 4*), corresponding to the case where no operator picks up the phone. As a result, we can assure that enough resources have been reserved before the program execution. Note that our *consume* annotations are made explicit in the translated program but they could just as well be incorporated in a library by extending the existing JAIN API.

This example illustrates the use of an existing tool to perform some non-functional processing of DSL programs. That is, resource-bounds checking is introduced by re-using an existing annotation language and underlying tools. In doing so, the DSL compiler writer is guided by the annotation language to determine the required non-functional information to be provided in the translated program. This strategy prevents him from re-doing an analysis of this non-functional domain. Furthermore, annotations allow a modularization of the compiler in that their processing is left to a later phase, performed by dedicated existing tools.

6.3 Program Facet

Unlike functional *program facets* that define compiler treatments specific to a subject program, non-functional *program facets* correspond to information collected on a given program. If we consider the example shown in Figure 7, an extension of the annotation approach could collect the *acquire* annotations for each SPL handler, to compute the maximum number of routing operations performed by the SPL program. This number could then be used by a security policy of the execution environment. In doing so, some restrictions could be enforced on the number of routing operations performed by an SPL program, to preserve the platform performance. This enforcement can occur prior to the program execution with respect to currently available resources. This process could be seen as admission control. The approach could be extended to all of the resources consumed by a program. As a result, a telephony service would be accompanied by a list of the resources required for its execution.

Note that this program facet is different from the language facet, presented in Section 6.2. Indeed, the language facet did not address resource consumption globally to the service; it only ensured that a consumed resource had been previously allocated.

An important issue in the telephony domain is service billing. By examining the kind of compilation treatments on billing operations, one can observe that it amounts to defining non-functional aspects, analogous to monitoring activities (*e.g.,* logging). As an example, some timer could be enabled in the handler that starts a call session, and disabled in the handler terminating a call.

7 Discussion

Our methodology for DSL compiler development is to translate the logic of a program into a GPL representation that is amenable to generative program-

ming approaches. One of these approaches relies on AOP to introduce specific behaviors at some locations in the GPL-translated program.

Our methodology for DSL compiler development heavily relies on generative programming approaches and corresponding tools. In doing so, their features, or even limitations, need to be take into account when developing the compiler. Concretely, limitations regarding AOP were discussed earlier. In fact, they did not concern the approach but rather the AspectJ tool used for our experiments. For example, variable introduction is only possible at the level of a class, not inside a method. Moreover, AspectJ aspects are context-insensitive in that they cannot directly manipulate the variables that are in the scope of a cross-cutting point. This feature would permit defining finer grained advice and improve the quality of the generated code.

The generality of meta-programming [17] makes it an alternative approach to AOP, as well as most other generative programming approaches. In this context, a DSL compiler resembles an interpreter annotated so as to execute the static language actions and to produce code for the dynamic language actions. Meta-programming tools give a fine-grained control over program generation, which can occur at any program point. A key issue that needs to be explored is how to modularize DSL compilation with meta-programming, to mimic what we do with AOP, annotations and program specialization.

8 Conclusion and Future Work

In this paper, we present a new methodology to develop DSL compilers. Our methodology is composed of two steps: compiling program logic and performing generative programming. Compiling a program logic produces a GPL-translated representation that abstracts over implementation details while being amenable to generative programming tools. These tools allow to model the high-level nature of DSLs, and the richness of the built-in domain-specific information, in terms of program generation.

Our approach modularize the program generation process of a DSL compiler. Each generative programming approach provides a paradigm, associated abstractions and tools, dedicated to a specific family of program generation. The compiler developer can thus choose the most appropriate generative programming approach for a given compilation dimension.

We have used our methodology to develop a compiler for a DSL dedicated to the creation of telephony services. We present aspects, annotations, and specialization opportunities that model various compilation dimensions of this DSL. This case study demonstrates that a DSL can make use of generative programming approaches and techniques in very effective ways. In essence, a DSL exposes information about programs that can be mapped by our approach into the realm of generative programming.

Compared to traditional approaches for compiler development, our methodology enables to have a modular treatment of the domain-specific concerns exposed

by a program. The resulting compilation process of a DSL is made simpler and less error-prone.

References

1. L. Burgy, L. Caillot, C. Consel, F. Latry, and L. Réveillère. A comparative study of SIP programming interfaces. In *Proceedings of the ninth International Conference on Intelligence in service delivery Networks (ICIN 2004)*, Bordeaux, France, October 2004.
2. L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Réveillère. Telephony Software Engineering: A Domain-Specific Approach. Research Report RR-5548, INRIA, Bordeaux, France, April 2005.
3. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proceedings of the 14th European Symposium on Programming (ESOP)*, volume 3444, pages 311–325. Springer-Verlag, 2005.
4. C. Consel. *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter From A Program Family To A Domain-Specific Language, pages 19–29. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.
5. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.
6. C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, September 1998.
7. K. Czarnecki. Overview of generative software development. In *Proceeding of the Unconventional Programming Paradigm*. To appear as Springer LNCS (Fradet, P. editor), September 2005.
8. M. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
9. J. Deruelle, M. Ranganathan, and D. Montgomery. Programmable active services for JAIN SIP. Technical report, National Institute of Standards and Technology, June 2004.
10. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer-Verlag, 2001.
13. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, 1997.

14. A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265, IETF, June 2002.
15. J. Rosenberg, et al. SIP : Session Initiation Protocol. RFC 3261, IETF, June 2002.
16. Sun Microsystems. The JAIN SIP API specification v1.1. Technical report, Sun Microsystems, June 2003.
17. W. Taha. *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter A Gentle Introduction to Multi-stage Programming, pages 30 – 50. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.
18. S. Thibault, C. Consel, J. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, September 2000.

Efficient Code Generation for a Domain Specific Language*

Andrew Moss and Henk Muller

Department of Computer Science,
University of Bristol

Abstract. We present a domain-specific-language (DSL) for writing instances of a class of filter programs. The values in the language are symbolic and independent of a concrete precision. Efficient code generation is required to fit the program onto a target device limited in both memory and processing power. We construct an interpreter for the DSL in a language specific to the device which contains the semantics of the target instruction set embedded within a declarative meta-language. The compiler is automatically generated from the interpreter through specialisation. This extension of the instruction set allows the construction of an interpreter for the DSL that is both simple and clear. In particular it allows us to declare static representations of the symbolic values, and have the specialisation of the code produce operate upon these values in the instruction set of the target device.

1 Introduction

Our target domain is the sensor architecture of a wearable computer. This domain has tight constraints on the energy usage of system components; requiring the use of micro-controllers which are power efficient. Such devices have a limited instruction set, little data memory, and a small store for program code. The current method for programming such devices is to write code directly in assembly language. A C compiler exists, but the size of the code produced makes it impractical for real programs.

The class of programs that execute in this domain are hard real-time processes. They input the raw sensor readings, which are filtered to identify features for further processing by the application on the main processor. The sensor is responsible for polling the data, filtering it, and communicating the result over a bus to the main processor.

From the designer's viewpoint, the filter is a set of equations defining the relations between input and output. The equations are composed of simple arithmetic operations but their semantics are defined over arbitrary precision bit-strings. This abstract viewpoint is a long way from the actual implementation as a series of instructions.

* This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 47–62, 2005.

Our goal is to construct a DSL for the designer that matches their abstract view as closely as possible. This abstract program compiles into a concrete executable for the target device. In order to compile the code, the arbitrary precision values must be refined into appropriate approximations using static precision. It is important that the efficiency of this generated code should match that of an assembly language programmer — otherwise the generated program will fail to fit upon the device.

To meet this compilation goal the equational system specified in the DSL must be converted into a sequence of device instructions. The equations of the DSL contain no explicit control flow; there is a single implicit loop around the set of equations in the program. This loop forms an infinite loop around the statements in the generated program; each filter is designed to execute for as long as there is power to the sensor. Within the loop there are a series of expressions that compute the value of each of the variables in the filter. Each iteration of the main loop is a single input/output cycle on the sensor and can be scheduled between the hard temporal constraints.

Our compilation technique is to write an interpreter for the DSL, and automatically generate a compiler from that interpreter through specialisation. The novel aspect of the technique is that the interpreter cannot be constructed directly in the target language. Instead the interpreter is constructed in two parts; an interpreter of the target instruction set is embedded within a declarative meta-language to form an extension of the target language, and an interpreter of the DSL is written in this extension. The extension consists of operations in the meta-language and calls to the target interpreter.

In order to compile a DSL program into the target language, the DSL interpreter is specialised with respect to the program. The specialiser operates on the meta-language, and has no direct knowledge of either the DSL or the target language. Specialisation of the DSL interpreter produces a result in which the calls to the target interpreter are left in the residual. These calls to the target interpreter are *syntactically isomorphic* to the target instruction set. By careful construction of the DSL interpreter we produce a residual program that is written in the meta-language but only contains calls to the target interpreter. Thus the resultant code is syntactically isomorphic with the target language and can be converted by a simple syntax conversion. This process compiles code from the DSL to the target language using a specialiser that performs source-to-source transformations *only* within the meta-language.

2 Target Device

Our target device is a PIC-16F84[1] micro-controller. We have chosen this device as it is representative of many real-world applications in the robotics and pervasive computing fields. The main attractions of the device are low cost (under a dollar) and low power consumption (several milli-watts). Unfortunately this simple design creates limited resources and an awkward programming model.

Each micro-controller uses 8-bit logic and arithmetic. Operations are performed in a nominated working register, the other registers are devoted to status

and control flags and a scratch area for the programmer. As external memory is not connected to these devices, the scratch area (about 60 8-bit locations) forms the only memory available for the programmer to use[1] The program is limited by the size of the program memory to about 1000 instructions.

The program and data memories are not shared, and it is not possible for the device to modify its program memory while it is running. This limits the possibilities for generative or self-modifying code but it does make analysis of PIC programs easier.

The instruction set of the PIC supports basic arithmetic operations (addition and subtraction) but more complex operations (eg multiplication and division) must be written by the programmer. There is support for logical AND, and both inclusive and exclusive OR. Control flow is limited, with decisions being performed by conditional *skip* instructions. These can either execute the next instruction, or skip over it, depending on the value of a bit in the register file. Logical shifts are not supported, although rotation can be performed through a register and the carry bit, a single bit at a time. Logical shifts can be constructed from combinations of rotate sequences and masking.

The PIC uses a normal model of control-flow; each instruction in the program has an integer label and a register called the PC selects the current instruction. Execution of each instruction affects the PC and so directs the flow of control through the program. The important point is that the target machine is imperative and the active program point is part of the state being passed through the computation. This differs from the control flow in our meta-language and we shall explain how we have embedded this imperative state machine in a logic language in Section 5.

The code density of PIC programs is low because of the decisions made to select this instruction set. Logical shifts and rotations of multiple bits take several instructions which are costly both in processor cycles, and slots in the limited program space. Each conditional split in the control flow must be constructed from multiple instructions. Implementing multiple precision arithmetic is costly as we must extract bit-string from different locations and merge them in order to perform operations. This requires both shifting of data within registers and conditional code to interpret values at different precisions. However, as we will show this implementation can be achieved by specialising away the conditional code flows and choosing efficient combinations of operations. Implementing arbitrary precision (that is dynamically changing precision) would not be feasible because this approach could not reduce the code synthesised for the PIC.

3 Approach

We are constructing an interpreter of a domain language (D). This interpreter can then be partially evaluated with a specific program in that language. The evaluation compiles the program into the language that the interpreter is written

[1] There is also an EEPROM but it requires several cycles and instructions to access, and has a limited lifespan.

in. If we could construct this interpreter directly in our target language (T), then our compiler would be complete as shown in Equation 1 (the first Futamura projection). However, T is not rich enough to support such an interpreter, as argued in Section 2, and thus int_T^D is not directly constructible.

$$\forall i : [\![[\![spec]\!](int_T^D, P_D)]\!](i) = [\![P_T]\!](i) \tag{1}$$

If we assume that do have int_T^D available, and that we have a rich meta-language (M) that we use as an intermediate, then we can write a second interpreter (int_M^T) that can be composed with the first. So int_M^T executes the instructions of int_T^D which executes the target program. Partial evaluation of the first interpreter with respect to the second, will produce an interpreter of D that is written in M. The structure of this interpreter will retain the semantics of the target language (T), as parts of the predicates that model each instruction will be left in the residual code. This process is shown in Equation 2 where we term the language of the final interpreter M', as it contains the embedding of the semantics of the instructions in T. Programs in M' have all of the expressive power of M, but the parts of them using the predicates that model T are syntactically isomorphic to programs in T.

$$[\![spec_M]\!](int_M^T, int_T^D) = int_{M'}^D \tag{2}$$

We have constructed both $int_{M'}^D$, and $int_M^{M'}$ directly. These are the DSL interpreter and the target interpreter respectively. Handwriting the DSL interpreter requires some care, as we must ensure that when it is specialised against a program all of the predicates that do not share a syntactic isomorphism with T are correctly analysed as static and removed from the residual. Ensuring that all calls to isomorphic predicates remain within the residual is easier, as we have a single dispatch point within the target interpreter that we can explicitly tag as *rescall* in the BTA. The Ciao package supports these explicit annotations.

One advantage of this manual construction is illustrated by our compilation process in Equation 3. The only external tool that we require is a specialiser for the meta-language, there is no need to write any analysis or transformation tools that natively understand either the source language (D) or the target language (T).

$$\forall i : [\![[\![spec_M]\!](int_{M'}^D, prog_D)]\!](i) = [\![prog_T]\!](i) \tag{3}$$

A predicate written in M' is *syntactically isomorphic* to a program in T iff it contains a conjunction of terms that are syntactically isomorphic to T. A term is syntactically isomorphic to a program in T iff it is a call to a predicate in the target interpreter or a call to a predicate that is itself syntactically isomorphic to T. More informally, when a program in the meta-language reduces to one that consists only of calls to predicates that model the instruction set of the target, then the residual program is syntactically isomorphic with a target program as we can perform a simple syntactic substitution to rewrite it as PIC assembly code.

In the remainder of this paper, we describe the construction of the languages and interpreters required to implement this approach. The DSL language (D) is described in Section 4 and the construction of its interpreter in M' is covered in Section 6. The target has been described in the preceding section, embedding the interpreter of the target language in M is detailed in Section 5.

4 Domain Specific Language — *D*

The aim of a DSL is to allow the construction of clear programs. This clarity is achieved by abstracting away details that are constant within the domain. In the case of D all necessary control-flow in the program has been removed. Each program written in D is a set of equations defined over variables. Some variables are nominated as representing the input and output streams, some as stateful variables that preserve their values between iterations of the filter, and the rest are transient. An implicit outer loop exists around the equations that loops forever.

In order to remove the control flow within the loop we use the semantics of a simulation language for variable access. The implicit outer loop separates the execution of the program into discrete cycles. Each variable in the language maintains two states, one for the current cycle, and one for the next cycle. All variable updates write to the state for the next cycle. All variable accesses read from the state for the current cycle. Between cycles the contents of the next state are copied into the current state. This bi-state for each variable removes dependencies in the program code and means that equations can be evaluated in any order without affecting the result of the computation for each cycle. An example of this type of program is shown in Figure 1.

```
input(z), output(x), state(q,a,h,p,r)
pminus = q + (a * p * a)
zpred = h * x
resid = z - zpred
k = (h * pminus) / (r + (h * pminus * h))
x = (resid * k + x)
p = (1 - (k * h)) * pminus
```

Fig. 1. Example DSL program : Kalman Filter

4.1 Variables

Each variable in the DSL has arbitrary precision; it is represented as a pair of bit positions and a bit-string. The bit-string contains the value of the bits between the given upper and lower bit of the variable. Each item in the bit-string has the value 0 or 1, and each position n has a coefficient of 2^n except for the upper position which has a coefficient of -2^n. The radix point is between positions 0

and −1. Each value is 2s-complement, and we round towards negative infinity. This means that all of the bits in the positions above the bit-string are sign-extended from the top bit given, and all of the bits in the positions below the bit-string are zero.

This definition of the value represented by each variable means that although we only hold a finite portion of the bit-string that exactly represents the value, all of the possible bit-positions for the variable are well-defined. This definition is shown in Figure 2.

$$\mathrm{bit}(n, \mathrm{var}(u, l, bs)) = \begin{cases} s & n \geq u \\ bs[n-l] & l \leq n \leq u \\ 0 & n < l \end{cases}$$

$$\text{where} \quad \begin{aligned} s &= bs[u-l] \\ (x_0, x_1, ...)[n] &= x_n \end{aligned}$$

Fig. 2. Definition of bit-positions within a variable

The most basic function that we can define on these values is transformation; mapping from a value represented at one precision, to the closest representation of the value at another precision. This function is not invertible, as information is lost when mapping from a higher precision to a lower precision. This lost information means the function is not injective, and will map several values onto the same representation in the lower precision.

The transformation function is shown in Figure 3. The simplicity of the function definition is a result of the bit being well defined in all positions in the original precision. This results in a lack of corner cases and thus a uniform definition.

$$\begin{aligned} &\mathrm{trans}(\mathrm{var}(u, l, bs), ou, ol) \;=\; \mathrm{var}(ou, ol, obs) \\ &\text{where} \quad \forall n : ol \leq n \leq ou \\ &\quad obs[n - ol] \;=\; \mathrm{bit}(n, \mathrm{var}(u, l, bs)) \end{aligned}$$

Fig. 3. Transformation function

Each variable in the target language must have a fixed precision; the DSL program *implicitly* defines a minimum value for these precisions. It is beyond the scope of this paper to describe the different techniques of assigning precisions to the intermediate variables in a computation. Broadly the different approaches can be described as analytical techniques [2] that preserve correctness, or statistical techniques that manage the propagation of error [3,4,5]. We assume that a solution for the set of precisions is known that correctly represents the intention of the programmer, and this set of precisions is supplied to the DSL interpreter — that the bit-positions of each variable are static data. With this assumption

Operation	Intermediate Precision
$(au, al) + (bu, bl)$	$(\max(au, bu) + 1, \min(al, bl))$
$(au, al) - (bu, bl)$	$(\max(au, bu) + 1, \min(al, bl))$
$(au, al) \cdot (bu, bl)$	$(au + bu, al + bl)$
$(au, al) \,/\, (bu, bl)$	$(au - bl, al - bu) \subset (au - bl, \inf)$

Fig. 4. Intermediate and output precisions

we focus on the automatic generation of a code operating in a given fixed-point form, rather than determining the value of that form.

4.2 Operations

Defining a transformation function first makes the definition of individual operations simpler. We no longer have to consider the various cases of which bits are contained in a value representation, and which are not. The set of operations that we support in the language is addition, subtraction, multiplication and division. Each operation first transforms both operands to an intermediate precision (this is the necessary precision to preserve correctness for each bit in the output), performs the operation between two values of that precision and then transforms the resultant value to an output precision. The intermediate precision may be larger than the output precision, for example in the case of addition where carry chains must be computed to determine the correct result. For each operation the intermediate precision with respect to a given pair of input precisions is shown in Figure 4. The value produced at this intermediate precision is then transformed to the precision of the target variable as the output precision.

For addition, subtraction and multiplication it is possible to statically determine this required precision from the precision of the operands alone. For a division operation this produced precision is dependent on the actual values supplied to the operation, and the safe approximation is a bit-string of infinite length as any divisor that is not a power of 2 will produce a repeating binary string as a result. We can still use the expression $bu - al$ as a safe approximation of the upper bit in the precision as we assume that we cannot divide by zero. The safe approximation is not representable so we use the precision that represents a subset of the values representable in the safe case. This is unavoidable when performing division on a machine with finite resources. The pseudo code in Figure 5 shows the process for performing each operation in the language.

The basic operations can be combined into expressions by the programmer. We assume standard associativity for each operator. Each expression is a simple tree of operations that we flatten into a sequence of basic operations. This requires the introduction of temporary variables to hold each intermediate result within the expression. The language that we have described thus far is executable and can be used by the designer to develop programs and test their correctness. Arbitrary precision arithmetic in the DSL is captured in the pseudo

```
perform(op,var(au,al,av),var(bu,bl,bv),ru,rl) =
    (iu,il) = Select from Fig.4 based on op
    ai = trans(var(au,al,av),iu,il)
    bi = trans(var(bu,bl,bv),iu,il)
    rs = perform op on ai and bi
    defined(ru,rl) ->
        return var(ru,rl,trans(var(iu,il,rs),ru,rl))
    undefined(ru,rl) ->
        ru = iu
        rl = il
        return var(ru,rl,rs)
```

Fig. 5. Pseudo code for each operation

code by undefined precisions for the output of an operation. This operation is not feasible on the target device. The implementation on the target requires ru and rl to be static values passed to the DSL interpreter. For intermediate values these precisions are constrained in the same manner as the temporary variables in Section 4.1.

5 Embedding the Target Language — $M' = M + T$

The meta-language (M) is Ciao [6] (a dialect of Prolog). Prolog is used because of the maturity of analysis and specialisation tools for the language and their state of integration into Ciao — in the form of Ciaopp [7]. The choice of language for M must be well-suited for the construction of the target interpreter. Most of the operations within the interpreter are mappings from one domain to another. These maps have a natural form as logical predicates. Prolog is both typeless and symbolic with a simple encoding of anything as data (even predicates) that allows a rapid cycle of formulating ideas and testing them as code. The most important feature in Prolog for our needs is that it features dynamic syntax. When writing code at several different language levels it is necessary to test the parts in isolation without constructing a correctly typed framework for them.

Each instruction in the target language maps to a predicate. M' is the combination of the Prolog language with calls to these predicates. Each transforms an initial state to an output state as directed by its parameters. An example is given in Figure 6. The *addwf* instruction retrieves two values from the initial memory state (one from the working register 0, and one specified in the first parameter), adds the two values, and then depending on the second parameter, either stores the result in the working register, or in the specified register. The carry and zero bit are set according to the result.

The memory state is represented by a simple list of pairs of integers, one with the location label and one with the current value bound to that label. The lookup and store operations retrieve the value from a label, and produce a new state with the label set to the given value. These operations are expressed clearly in a declarative language as shown in Figure 7.

```
addwf(S,F,D,S3) :-
    lookup(0,S,W),
    lookup(F,S,X),
    Ans is W+X,
    AnsM is Ans mod 256,
    (D=:=1 -> store(S,F,AnsM,S2)
            ; store(S,0,AnsM,S2)),
    statusBits(Ans,S2,S3).
```

Fig. 6. Sample implementation of T in M

```
mTransAd( (L,V1), L, (V1->V2), (L,V2) ).
mTransAd( (L,V), L2, (_->_), (L,V) ) :- L \== L2.
mTrans( S, L, X, S2) :- map(S,mTransAd(L,X),S2).
lookup(Ad,(_,S),V) :- mTrans(S,Ad,(V->_),_).
store((D,S),Ad,V,(D,S2)) :- mTrans(S,Ad,(_->V),S2).
```

Fig. 7. State transition predicates

These predicates map directly onto the target instruction set although they are executable in a logical language. In the PIC micro-controller each instruction is located at an address in memory and the semantics of the instruction-set specify how the state transitions affect the PC, indirectly controlling which instruction is executed next. In the target language T a program is comprised of a list of instructions. The control-flow is contained implicitly in the sequencing of the list. When T is embedded within M the control-flow through the instructions is the Prolog control-flow around the calls to the target predicates. In general this can be more complex than a simple sequence, although programs in M' that are syntactically isomorphic to T are a simple sequence of conjunctions by definition.

The interpreter of T implements the semantics of the individual instructions, rather than emulating the machine that executes them. This subtle difference allows Prolog code to be freely interleaved with calls to the target interpreter. The resulting code can be executed natively in M as the explicit threading of the state gives a well defined meaning to the program. If a partial evaluation removes any surrounding Prolog control flow decisions (reducing them to conjunctions) then the resultant code has a syntactic one-to-one mapping with a PIC assembly language program. The syntactic mapping has to relocate the program to an absolute address in the program memory, but as all control-flow in T is relative this is a trivial mapping.

One difficulty this abstract control-flow poses is how to interpret the *skip* instructions that conditionally skip or execute the next instruction. In the PIC with each instruction bound to a location the PC can be incremented to implement skipping. When a program in T is embedded in M there is no direct relationship between one instruction and the next in the control flow. This relation now depends on the Prolog code which calls the two instruction predicates. This difficulty is overcome using the higher-order features of the symbolic meta-

```
pic(Inst) :- Inst =.. [P,skipping(S),S].
pic(Inst) :- Inst =.. [P,skipping(S),_,S].
pic(Inst) :- Inst =.. [P,skipping(S),_,_,S].
pic(Inst) :- Inst =.. [_,(_,_)|_],
             call(Inst).
```

Fig. 8. State dispatcher

```
extendSign(S,(_,_,8),S).
extendSign(S,(Ad,Bit,Fill),Sout) :-
    Fill<8,
    OrMask  is 255 - (1<<Fill)+1,
    AndMask is (1<<Fill)-1,
    pic(btfss(S,Ad,Bit,S2)),
    pic(andlw(S2,AndMask,S3)),
    pic(btfsc(S3,Ad,Bit,S4)),
    pic(iorlw(S4,OrMask,Sout)).
alignByteInW((D,M),V,B,Sout) :-
    member(var(V,_,Vu,_,_,_,_),D),
    B>Vu,
    bitAddr((D,M),V,Vu,SignByte),
    aligned((D,M),V,Vu,SignBit),
    extendSign((D,M),(SignByte,SignBit,0),Sout).
```

Fig. 9. Example DSL interpreter operation

language. The entire state is encapsulated within a functor (eg $S \mapsto skip(S)$) when the skip condition holds. Then each PIC call is wrapped in a dispatcher that checks whether to execute or skip the predicate that is passed in. There is minimal change to programs written in M', and the interpreter merely requires the addition of a dispatcher as shown in Figure 8.

Implementing the target interpreter in M creates a language M' that allows a clear construction of the DSL interpreter. The clarity is achieved through interleaving declarative logic (symbol manipulation, backtracking and query solving) with calls that implement the semantics of instructions in T. The predicates implementing T must be passed ground values for instruction parameters, and must be fully deterministic. When backtracking is used to enumerate possible code generations in the DSL interpreter each execution of an instruction must generate a single transition in the state. Multiple transitions for a single instruction with ground parameters will generate spurious call traces and erroneous partial evaluations.

6 Interpreter of D in M'

Using the combination of the target interpreter and the meta-language that form M' it is now possible to write an interpreter for D. Throughout this paper we have argued that it is not possible to write such an interpreter directly in T.

The overhead of interpreting instructions, and maintaining a state for D and its mapping onto the device would consume all available resources.

Writing the interpreter in M' solves these problems. The state of the interpreter is composed of a static precision (upper and lower bit-position) for each variable, and a dynamic value (bit-string). The bit-string requires a mapping between partitions of the string and memory locations on the target device. The mapping can be defined declaratively in M' and passed to the interpreter as a static value. The only remaining dynamic values are the values of the bit-string — these must fit within the program memory for the filter to be executable.

We consider two representations of the bit-string. In a packed representation the lowest eight bits of the string occupy one location, each partition of eight bits occupies a sequential location in memory. In a modulo representation each bit n of a location maps to a position that is $n \; mod \; 8$. These two representations have a trade-off; packed variables take less memory but mod variables require less instructions to operate upon.

In M' we can declare disjoint clause bodies for each representation that map to different calls to target instructions. This method requires no run-time overhead and no extra code in the interpreter as back tracking at specialisation time will select the appropriate clause in each context.

Clarity is the result of operating directly on the same symbolic values in both interpreters. Performing an operation on bit-strings has a simple declaration as a uniform loop over a window of bits. Writing the same operation on locations which hold partitions of the bit-string is more complex as there are more cases to consider. All loops and queries within the interpreter that are dependent on the positions and sizes of bit-strings will be statically unrolled into target call sequences. Depending on the relative alignment of bits between a pair of registers, we must perform a different series of target instructions to implement the DSL operation.

In order to encode each DSL operation as a uniform loop over bits, we require an intermediate operation in the interpreter; alignment. This operation is composed of target calls and declarative logic. We show the simplest case of alignment in Figure 9. In this case we are attempting to extract a range of bit positions that are above those stored in the state, we extend the sign of the top bit in the represented value.

The alignment operation uses the variable declarations in the DSL state to find the bit-positions that are stored. It can then map these bits onto the representation in memory. There are several possible cases to be considered, each of which forms a predicate body. The requested bits may be above, below or within the stored bit-string. The representation of the value affects whether the bits are within a single location or span several locations.

Given the alignment operation, each DSL operation is constructed as a uniform loop over bit-positions. The iterations of the loop that access bits outside the string will be specialised away. The structure of a pseudo DSL operation is given in Figure 10, showing the uniform operation on 8-bit partitions of the value, regardless of the precision and representation in memory. The $x_{[u,l]}$

$$o = a \cdot b \iff \forall p \in [o_l..o_u] \ : \ p - o_l \equiv 0 \pmod 8$$
$$carry_{[p+n,p+8]}, o_{[p+7,p]} = a_{[p+7,p]} \cdot b_{[p+7,p]} \cdot carry_{[p+7,p]}$$

Fig. 10. Mapping Of Intermediate Binary Operation

notation indicates the bits l to u in the variable x, which are instances of the alignment operation.

7 Application

We have given a formulation of the DSL interpreter that operates on a program, a declaration of the variable precisions and a declaration of variable representations. When this interpreter is specialised with respect to these parameters it produces a residual program in M' that we can convert to PIC assembly language.

The trade-off in choosing how to represent each variable is complex and hard to express. The trade-off is a constraint problem where the set of constraints to solve is specific to each program in the DSL. This constraint problem could be specified as a transformation of the program into a constraint language, but the formulation would be difficult.

As M' is a logic language we can use the declaration of representation to generate possible variable representations for the program. Each of these generated values can be supplied to the interpreter to automatically generate code using the representation. Each of these generated codes can be compared to find a target program that matches a given constraint (eg code size or memory usage). This example shows how the search space of the DSL interpreter can be explored to find solutions (target programs) that match criteria that are difficult to express directly as search problems.

The low code density of the PIC limits the complexity of an operation that we can demonstrate code generation upon. We will use the alignment case shown in Figure 9 and an entry point that includes an interpreter state with a single variable. The variable uses a packed representation within the PIC memory, its lowest bit is aligned with the lowest bit in a register location, and its higher bits are spanned onto the next location. This call to the alignment operation is shown in Figure 11.

The specialiser removes the static arguments to the call, which include the variable being accessed. The resultant code has eliminated the declarations that access the interpreter state and produced constant values as the parameters to the PIC calls. The variables in the resultant code are renamed to preserve sharing, which leaves the state threading intact in the produced code. The output is a straight-line code sequence composed entirely of calls to the PIC predicates, and can be syntactically transformed into the PIC program shown.

The interpreter state and all interpretive overhead has been removed from this operation, as the code does not modify the layout of variables in memory. Supplying different memory layouts, and variable representations produces the

```
:- entry alignByteInW(( [var(x,packed,6,-4,10,2,_)],S ), x, 7,
                      ( [var(x,packed,6,-4,10,2,_)],S2)).

alignByteInW(A,E) :-
    pic(btfss(A,11,1,B)),
    pic(andlw(B,0,C)),
    pic(btfsc(C,11,1,D),
    pic(iorlw(D,255,E)).

PIC program:
    BTFSS   11,1
    ANDLW   0
    BTFSC   11,1
    IORLW   255
```

Fig. 11. Specialisation of getSign predicate

appropriate access code. In larger interpreter fragments (such as multi-precision addition and multiplication) there is a more complex trade-off between the choice of representation, and the size of the residual code.

An alternative approach to implementing the DSL on the target device would be to write a library performing multi-precision operations. The contrast with our approach is dramatic, both in terms of complexity and the efficiency of the final code. The declaration of the alignment operation defines a series of condition checks and an appropriate body to execute in each specific context. This overhead is removed entirely by the specialiser.

Writing each DSL operation as a library function would require a set of cases that contain all of the used calls (eg 8-bit / 16-bit). Each case would need to be written separately, increasing the complexity of the implementation. Any unused bits in the computation, such as using an 8-bit multiplier on a pair of 6-bit values would introduce unnecessary overhead and the performance of the program would suffer. Our approach avoids this problem by using the specialiser to automatically determine which expansion is necessary in each context. A library approach would also require the precision for each variable to be passed to the routine. In contrast our approach reduces these values to constants, and they are implicitly defined when needed so they require no storage. The disadvantage is that unrolling the code in this way increases the program size.

8 Related Work

The approach of code generation by abstract operations is detailed in *delayed code generation* [8]. This method of producing object code for a smalltalk compiler uses intermediate operations between the source and target languages that contain abstract values. These values can be operated on through *deferred* operations that can eliminate intermediate steps in the object code. This approach differs from our method in that these operations are performed upon syntax

trees within the compiler. We execute our operations directly through a partial evaluation and reduce expressions based on the static analysis performed within the specialiser.

The uses of partial evaluation have been studied extensively [9,10], and in particular the use within meta-programming to compile domain specific languages [11]. The novelty of our approach in comparison to these techniques is the construction of the domain interpreter within an extension of the target language. This extension allows us to use the expressive power of the declarative meta-language to write our interpreter clearly and simply. The elements of the target language are residualised during the specialisation, where-as the operations in the meta-language are entirely static. This produces a syntactic isomorphism between the residual program and the target language.

The connection between macro languages and multi-stage computations has been investigated in a functional setting by Ganz et al [12]. They formalise their macro language (MacroML) as a MetaML interpreter and use the features in MetaML to show that their language is type-safe (that macros cannot create type errors in the final program) and stage-safe (that macro expansion does not rely on run-time evaluations).

This work differs from our own in that we are using an untyped language as the meta-language, and rather than explicit staging constructs we are using online partial evaluation to separate the stages and perform the macro computation. Our work focuses on how to use an implicitly staged computation to perform macro operations; these are not limited to the operations in a conventional macro language as we can freely mix data values between the stages, allowing some runtime values to effect compile-time computations. Of course these runtime values are restricted by the static analysis used within the specialiser.

The approach in Sh [13] uses static meta-programming to embed a shader language in C++ templates. The shader programs are constructed at run-time by recording the operations executed in the meta-language. This reuses the bulk of the C++ compiler for parsing, grammar checking and code generation. This approach is similar to our embedding within a host compiler, although the recording of operations is performed at partial-evaluation time by the specialiser leaving no runtime overhead.

Previous work in compiling embedded languages [14,15] has looked at embedded typed languages within an existing typed functional language. This allows the meta-language to act as a host compiler and produce code for the DSL. This differs from our approach as we are embedding a lower level untyped language that contains the semantics of the executable target domain. In particular, the construction of our interpreter is similar to the handwritten cogen of [14], in that we have deliberately written the output of a theoretical specialisation. We cover this aspect of the work in Section 3.

Herrman and Langhammer show a similar approach to our work in [16]. A DSL for image processing operations is constructed for a similar class of filter programs. Efficiency is also a concern in that domain and their system generates code from an interpreter to remove the interpretative overhead. The construction

of their interpreter uses explicit staging constructs rather than a specialisation approach.

9 Conclusions and Future Work

We have presented a language that uses a high level symbolic representation of our problem domain. This representation has been compiled into an efficient executable form for an extremely low level micro-controller with limited resources.

We have shown that embedding the target language in our choice of meta-language allows the construction of an interpreter that would not be feasible otherwise. This interpreter is both simple, and clear, and when specialised it produces a residual program that is syntactically isomorphic with with the target language. This achieves compilation from the domain language into the target language by a specialiser that only operates in, and on the meta-language.

Our next step will be to investigate the compiler on a realistic target from the domain. We will show how efficient the compilation process is, and investigate optimisation techniques on the generated code.

Acknowledgements

The authors offer thanks to John Gallagher for creating the term *syntactic isomorphism* and also for constructing a clear overview of the work. We would also like to thank the anonymous reviewers for their helpful comments which aided the structure and content of this paper.

References

1. Microchip. PIC16F84 data sheet. Technical Report DS35007B, Microchip Technology Inc, 2001.
2. Willems,M. and Bürsgens,V. and Meyr,H. FRIDGE: Floating-Point Programming of Fixed-Point Digital Signal Processors. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, pages 1000–1005, San Diego, Sep. 1997.
3. Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems-II:Analog and Digital Signal Processing*, 47(9):840–848, September 2000.
4. Radim Cmar, Luc Rijnders, Patrick Schaumont, Serge Vernalde, and Ivo Bolsens. A methodology and design environment for DSP ASIC fixed-point refinement. In *DATE*, pages 271–, 1999.
5. Keding, H. and Hürtgen, F. and Willems, M. and Coors, M. Transformation of Floating-Point into Fixed-Point Algorithms by Interpolation Applying a Statistical Approach. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Toronto, Sep. 1998.

6. M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, 1999.
7. Manuel V. Hermenegildo, Francisco Bueno, German Puebla, and Pedro Lopez. Program analysis, debugging, and optimization using the ciao system preprocessor. In *Proceedings of the 1999 international conference on Logic programming*, pages 52–66, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.
8. Ian Piumarta. *Delayed Code Generation in a Smalltalk-80 Compiler*. PhD thesis, Department of Computer Science, University of Manchester, Oct 1992.
9. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
11. P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.
12. Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *ICFP*, pages 74–85, 2001.
13. Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.
14. Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27, London, UK, 2000. Springer-Verlag.
15. Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.
16. T. Langhammer C. Herrmann. Automatic staging for image processing. Number MIP-0410. 2004.

On Domain-Specific Languages Reengineering

Christophe Alias and Denis Barthou

Laboratoire PR*i*SM, Université de Versailles, France
Christophe.Alias@prism.uvsq.fr
Denis.Barthou@prism.uvsq.fr

Abstract. Domain-specific languages (DSL) provides high-level functions making applications easier to write, and to maintain. Unfortunately, many applications are written from scratch and poorly documented, which make them hard to maintain. An ideal solution should be to rewrite them in a appropriate DSL. In this paper, we present `TeMa` (`Te`mplate `Ma`tcher), an automatic tool to recognize high-level functions in source code. Preliminary results show how `TeMa` can be used to reformulate Fortran code into Signal Processing Language (SPL) used in SPIRAL. This opens new possibilities for domain-specific languages.

1 Introduction

With domain-specific languages, algorithms are described in a more abstract and compact way than with traditional imperative programming languages. Expressing algorithms at a higher level of abstraction adds portability and improves programmer productivity for code writing and maintenance. Moreover, the higher the representation of the program, the more aggressive the compiler optimizations can be: for instance a generative approach can yield from a mathematical formula in SPIRAL [16] a finely tuned code for a particular architecture, exploring both algorithmic variations and traditional code optimizations. The same benefit appears also for code verification.

However, this approach assumes that the user specifies his algorithm using a domain-specific language. Starting from an existing C or Fortran program and finding in it fragments that correspond to domain specific templates would be desirable but seems difficult to obtain. The reason is that it amounts to algorithm recognition, an old problem in computer science. Basically, one would like a compiler or analyzer to automatically find that lines 10 to 23 are an implementation of a DFT for instance. A natural solution would be to search the code for patterns of known functions (matrix-matrix product, tensor product or direct sum for the DFT). Such a facility would enable many important techniques:

- Program comprehension and reverse engineering: we can rewrite part of the code with a higher level language, enhancing code maintenance and portability.
- Program optimization: if we have the necessary items in our library or if we can use a generative approach, we may replace lines 10 to 23 by an

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 63–77, 2005.

automatically tuned version. We may even replace the relevant part of the code by a completely different implementation.
- Program verification: if we know that the program specification asks for a DFT and the analyzer does not find it, we may suspect an error.
- Hardware-software co-design: if we recognize in the source program a piece of code for which we have a hardware implementation (e.g. as a co-processor or an Intellectual Property component) we can remove the code and replace it by an activation of the hardware.

The pattern could be a naive implementation of the function but obviously the approach is interesting only if the detection abstracts away some variations between the code fragment and the reference implementation. Program variations can arise from data structure variations (coming from scalar promotion, array expansion, structures instead of arrays,...), control variations (coming from loop fusion, unroll, skewing,...), organization variations (coming for permutation of program statements) or semantic variations (using associativity or commutativity for instance). Obviously, the abstraction obtained depend on the range of variations handled by the detection.

In this paper, we present an approach to automatically find, in linear time, all possible instances of a given template in a program. Implemented in the `TeMa` tool and connected with a more expressive method already described in [1], the method is able to find the parameters of the template corresponding to a particular instance. It handles many implementation variations and extends previous works by handling variations concerning data structures using arrays. Experimental results on SPEC benchmarks show that our method is able to abstract away many variations. Finally, preliminary results show that this technique is able to reformulate Fortran code into Signal Processing Language (SPL) used in SPIRAL. This opens new possibilities for domain-specific languages.

The paper is organized as follows: Section 2 introduces the notations and definitions used in the paper. Section 3 describes the algorithmic content of `TeMa` and provides a classification of program variations handled. Finally, section 4 presents the `TeMa` tool and provides experimental results.

2 Background

SPL [21] (Signal Processing Language) is a domain-specific language for describing matrix factorizations, and thus fast algorithms for computing matrix-vector products. In particular, it can be used for describing fast signal transforms such as FFT or WHT. An SPL program is an expression involving a variety of operations including composition, direct sum and tensor product. Let us give an example of SPL program. Briefly recall that given two matrices A and B, the *tensor product* of A and B, is the matrix $A \otimes B = [a_{ij}B]$. The WHT (Walsh-Hadamard Transform) over a sampled signal of 2^n elements can be written $\text{WHT}_{2^n} = F_2 \otimes \ldots \otimes F_2$ n times , where $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ denotes the FFT trans-

form over 2-dimensionnal vectors. The SPL program computing the WHT over 2^3-dimensionnal input vectors can thus be written:

```
(tensor (F 2) (tensor (F 2) (F 2)))
```

In addition to improve readability and thus maintenance, recovering an SPL program from a C or Fortran program allows to benefit of the SPL compiler optimizations.

The approach investigated in this paper is to recognize within the program the slices corresponding to naive implementations of SPL functions, then to rebuild the SPL formula. The naive implementations to find are expressed with program patterns. A *pattern* is a schema of program with wildcards values and functions. For example, figure 1 gives the pattern of a reduction, where wildcards are denoted by □. One contribution of this paper is the algorithm to

```
s = □
do i = 1,n
| s = □(s,□)
enddo
return s
```

Fig. 1. Pattern of a reduction

quickly find all possible instances of a pattern within a program. In the `TeMa` tool presented thereafter, it is connected with a more expensive method already described in [1] to check if the program slices found are effectively instances of the pattern. In case of success, our exact method provides the corresponding values of □.

Finding instances of a pattern in a program means finding all program slices *equivalent* to an instance of the pattern. But what does exactly means «equivalent» ? We will consider a weak version of semantic equivalence called *Herbrand-equivalence* and denoted by $\equiv_{\mathcal{H}}$. Instead of indicating whether two algorithms compute the same (mathematical) function, Herbrand-equivalence just indicates if they used the same mathematical formula, syntactically. In this way, Herbrand-equivalence can be considered as a true algorithmic equivalence. Even if Herbrand-equivalence is weaker than semantic equivalence, and seems to be easier to check, it has been unfortunately proven undecidable [2].

Our detection algorithm uses a powerful extension of automata called tree-automata. A *tree automaton* is a tuple $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, where Σ is a signature, Q the set of states, $Q_f \subset Q$ the set of final states, and Δ a set of transition rules of the type $f(q_1 \dots q_n) \longrightarrow q$, where $n \geq 0$, $f \in \Sigma$ and $q, q_1, \dots, q_n \in Q$. Tree automata were introduced by Doner [5,6] and Thatcher and Wright [17,18] in the context of circuit verification. Most of usual operations on word automata (determinization, minimization, cartesian product, ...) extend naturally to tree automata [4].

3 Detecting SPL Functions

In this section, we present our method to detect SPL functions in a given C or Fortran program. We also present our preliminary approach to build an SPL program from relevant program slices. Finally, we evaluate the detection capabilities of our method in terms of program variations handled.

3.1 Overview of the Method

Figure 2 gives the main steps of our method. The slicing method search through the program the slices which *potentially* implement an SPL function. Then we check whether the slices are equivalent to a given SPL function by applying our exact instantiation test, already described in [1].

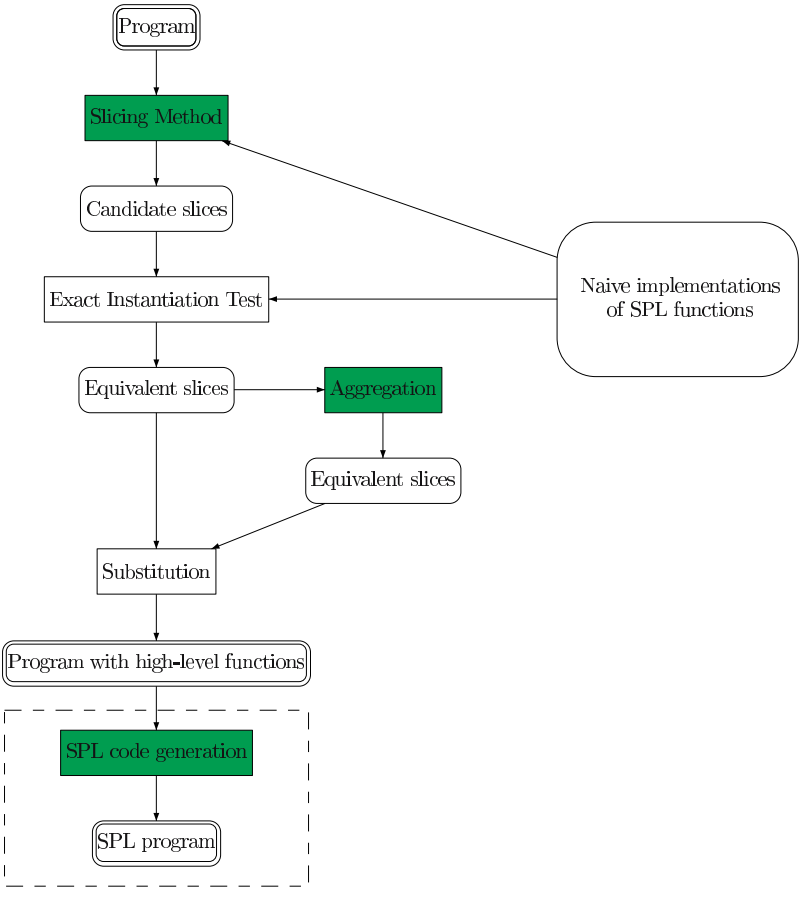


Fig. 2. Overview of the method

The aggregation allows to detect slices with multiple output statements, which is not allowed by previous steps. This is typically the case of matrix operations. Once implementations of SPL functions are found in program, it remains to substitute them by the relevant call to the SPL function (substitution). The program is then ready to be translated into an SPL program.

The contributions of this paper are the slicing method and the aggregation. We also propose a preliminary approach to recover SPL programs from relevant program slices. These important steps are described thereafter.

3.2 Slicing Method

The aim of our detection algorithm is to provide the parts of the program which potentially compute the same arithmetic expression than an instance of the pattern. The main idea is to walk through the pattern and the program def-use chains as long as pattern and program operators are equal. If the method reaches the last statement of the pattern, all reached program statements will be yield as a candidate slice, meaning that they may be Herbrand-equivalent since the expressions computed may have the same sequence of operators.

Algorithm *Build_Automaton*

Input: *The pattern or the program.*
Output: *The corresponding tree automaton.*

1. Associate a new state to each assignment statement.
2. For each state:

$$q = \boxed{\texttt{r = } f(\phi(Q_1) \dots \phi(Q_n))}$$

   Add the transitions: $f(q_1 \dots q_n) \longrightarrow q$, for each $q_i \in Q_i$.
3. For each state:

$$q = \boxed{\texttt{r = } \square(\phi(Q_1) \dots \phi(Q_n))}$$

   Add the transitions: $q_i \longrightarrow q$, for each $q_i \in Q_i$ *(input transitions)*.
   And: $f(q \dots q) \longrightarrow q$, for each operator f used in the pattern and the program, including constants (0-ary operators) *(looping transitions)*.

Fig. 3. *Build_Automaton*

The pattern and the program are assumed to be given in scalar SSA-form, a classical form in compilation that provides def-use chains. We first associate to the program and the pattern a tree automaton allowing to step them easily. This is done by using the algorithm described in figure 3. Basically, each state corresponds to a statement (step 1), and the transitions to a state are driven by def-use chains, and labeled by the statement operator (step 2). Pattern wildcards

are handled as Kleene star in word automata. Note that the wildcard value is a particular case of wildcard function $\square(\ldots)$ with arity 0. Step 3 of the method builds a loop for these states with any operator which appears in the program.

Consider the pattern and the program given in figure 4. For sake the of clarity, we have chosen a pattern and a program with unary operators which will lead to the word automata given in figure 5. But of course, our method can handle operators with any arity. The ϕ-functions can be seen as multiplexers selecting the last definition for a given value.

```
r1 = 1
do i = 1,n
| r2 = □(φ(r1,r3))
| r3 = 1+r2
enddo
r4 = exp(r3)
```

```
z1 = 1
t = 0
a = tan(t)
do i = 1,10
| z2 = 1/(φ(z1,z3))
| z3 = 1+z2
enddo
r = exp(z3)
```

Fig. 4. Pattern (left) and Program (right)

The idea is now to step simultaneously the two automata up to the pattern final state while the operators are equal. The two automata have as many entry points as constant leaves (1(), 2() here), and we have to start a comparison from each couple of leaves. The operations corresponds to the definition of the *cartesian product* of the pattern and program automata. The detected slices can then be computed by collecting all program states along the paths from initial states to each state with a final pattern state ($q_{\text{final}}, .$). The detection step is summarized in the algorithm described in figure 6.

Let us summarize our algorithm. Given a pattern and a program we first compute their tree-automata by applying *Build_Automaton*. The slices of "good" candidates are then obtained by stepping simultaneously $\mathcal{A}_T$ and $\mathcal{A}_P$. This task is achieved by *Output_Slices*. We finally apply the exact equivalence test described in [1] to check whether the slices are instances of patterns or not.

3.3 Aggregation

Our method is able to detect template occurrences with only one output statement. We present thereafter an extension to detect slices with several outputs by using an aggregation hierarchy of domain-specific functions.

Motivating example. The templates to match often use an array. Yet our method is able to detect the occurrences with only one output statement. Figure 7 provides an example of matching, where the template is the daxpy function of BLAS 1. Our slicing method yields the candidates slices S_1 and S_2, but the candidate $S_1 \cup S_2$ where $a = 2$, $\boldsymbol{x} = [s(1), s(2), s(3), u]$ and $\boldsymbol{y} = [1, 1, 1, v]$ is missing since its outputs are shared by several statements.

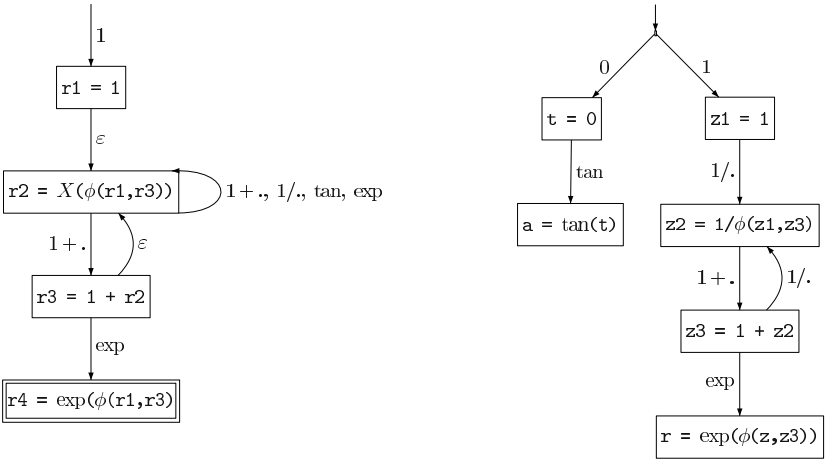


Fig. 5. Pattern automaton (left), and Program automaton (right)

Algorithm *Output_Slices*

Input: $\mathcal{A}_T$ *and* $\mathcal{A}_P$*, pattern and program automata.*
Output: $\{s_1 \dots s_n\}$*, the last statements of each candidate slice.*

1. Compute the Cartesian product $\mathcal{A} = \mathcal{A}_T \times \mathcal{A}_P$.
2. Mark the nodes with a final state of $\mathcal{A}_T$, and emit the $\mathcal{A}_P$ part of marked states.
3. For each marked node q:
   Compute the set of previous states $\mathcal{S}lice(q) = \{q',\ q' \longrightarrow^* q\}$.
   Then return the $\mathcal{A}_P$ part of $\mathcal{S}lice(q)$.

Fig. 6. *Output_Slices*

Aggregation hierarchy. One can remark that daxpy is constituted of 1-dimension daxpy instances. Another solution would be to detect «atomic» daxpy using the slicing method, then to *aggregate* them to make a larger daxpy.

In a more general manner, consider an algorithm A which produces an array, and a family of algorithms $(A_{\boldsymbol{i}})_{\boldsymbol{i}}$, where $A_{\boldsymbol{i}}$ outputs the $\boldsymbol{i}$-th array cell of A for each possible input:

$$A_{\boldsymbol{i}}(I) \equiv_{\mathcal{H}} A(I)[\boldsymbol{i}]$$

For each relevant input I and array index $\boldsymbol{i}$. Then A is said to be an *aggregation* of the $A_{\boldsymbol{i}}$.

Aggregation induces a hierarchy between algorithms, and particularly between templates. Typically, a daxpy is an aggregation of several scalar daxpy , and a matrix-vector product is an aggregation of dot products. Figure 8 provides an aggregation hierarchy between some BLAS 1 and 2 functions. $A \rightarrow B$ means "B is an aggregation of A instances".

```
do i = 1,n
  y(i) = a*x(i) + y(i)
enddo
return y
```

```
S1  do i = 1,3
S1    s(i) = 2*s(i) + 1
S1  enddo
S2  u = 2*u + v
```

Fig. 7. Two detections of daxpy

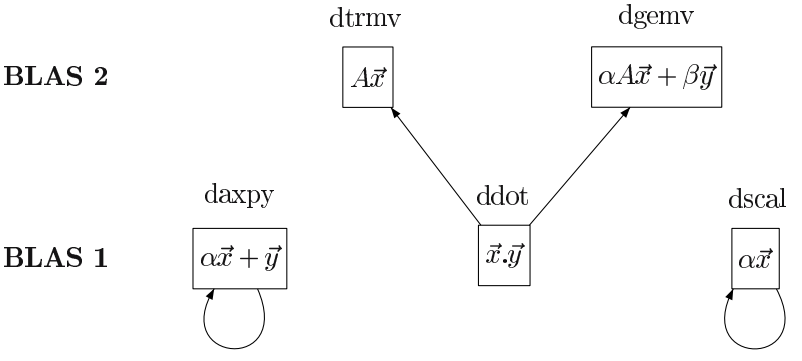


Fig. 8. Aggregation hierarchy of BLAS 1 and 2 functions

A solution is to detect the templates of the hierarchy by using the slicing method. Then we aggregate them in a bottom-up manner, from the leaves functions to the top functions. If A is an aggregation of $(A_i)_i$, all combinations of A_i instances are aggregated, and yielded as A instances. The aggregation is just a concatenation of slice outputs, as stated in the motivating example.

3.4 SPL Code Generation

Once the program is rewritten by using SPL functions, it remains to generate the corresponding SPL program. The preliminary approach investigated in this paper, but not yet implemented, is to select the program slices which can be completely unrolled, then to use the data-flow dependences to build the corresponding SPL program. The output of the SPL generation step is thus a set of unrollable program slices, and their corresponding SPL program. Unrolling is possible whenever the program slice uses `for` loops with bounds as expressions with constants and surrounding loop counters. The reaching definitions can be easily computed on the unrolled program slice by using usual methods [15]. These restrictive conditions lead to select the slices which can be unrolled, then to translate them into an SPL program. We believe that this approach is able to recover SPL programs corresponding to relevant program slices.

3.5 Program Variations Detected

The efficiency of our approach directly depends on its capacity to recognize SPL functions in a source code. A common way to evaluate an algorithm recogni-

tion system is to provide the different kinds of pattern variations it can handle [20,9,13,14]. We provide thereafter a detailed description of each variation. We also state whether our algorithm is able to detect them.

Organization variations. Any permutation of independent statements and introduction of temporary variables. The following example provides an organization variation with legal permutations (LP), garbage code (GC) and temporaries (T):

```
s = a(0)
c = 0
do i = 1, n
  s = s + a(i)
  c = c + 1
enddo
return s + c
```

```
    s = a(0)
    c = 0
GC  garbage = 0
    do i = 1, n
LP    c = c + 1
T     temp = a(i)
      do j = 1, p
GC      garbage = garbage + 1
      enddo
      s = s + temp
GC    garbage = garbage + a(i)
    enddo
    OUTPUT = s + c
```

Our algorithm works on a def-use graph, which avoids the artificial precedence constraints due to the text representation of the program. This allows our algorithm to handle legal permutations and garbage code. Our method compares two by two the operators used in the template and the program without handling variables, this allows to handle temporaries.

Data structure variations. The same computation with a different data structure. The following example gives a data structure variation with arrays and non-recursive structures:

```
s(0) = a(0)
do i = 1, 2*n
  s(i) = s(i-1) + a(i)
enddo
OUTPUT = s(2*n)
```

```
s.sum1 = a(0)
do i = 1, n
  s.sum1 = s.sum1 + a(i)
enddo
s.sum2 = a(n+1)
do i = n+2, 2*n
  s.sum2 = s.sum2 + a(i)
enddo
OUTPUT = s.sum1 + s.sum2
```

One of the important add-on of the paper is the ability of the detection to cope with different representation of arrays. Transformations such as scalar promotion (transforming an array section into as many scalars) or array expansion (the reverse) are handled thanks to the aggregation step.

Control variations. Any control transformation as if-conversion, dead-code suppression and loop transformations as peeling, splitting, skewing, etc. The following example give a control variation with a simple peeling:

```
s = a(0)
do i = 1,n
  s = s + a(i)
enddo
OUTPUT = s
```

```
s = a(0)
s = s + a(1)
do i = 2,n-1
  s = s + a(i)
enddo
s = s + a(n)
OUTPUT = s
```

In a general manner, we are abe to hande any variation which does not affect the operators nest of the expression computed by the program.

Each of these variations provide an Herbrand-equivalent slice, which our algorithm is able to detect in a general way. But we are not able to detect non-Herbrand-equivalent variations, such as *semantics variations*, which uses semantics properties of operators such as associativity or commutativity. Nevertheless, experimental results given thereafter shows that our method finds a large amount of correct candidates.

4 Experimental Results

In this section, we present TeMa, the implementation of our algorithm recognition system. We provide experimental results on SPEC benchmarking suite demonstrating the power of our slicing method. In addition, we show how TeMa can be used to recover an SPL program from a naive implementation of the Walsh-Hadamard transformation.

TeMa (Template Matcher) is the implementation of our algorithm recognition system, including the slicing method, the exact instantiation test, the aggregation method described in figure 2 and the substitution. For the moment, TeMa does not implements the SPL code generation. TeMa has been implemented in Objective Caml, and represents 10 kloc. TeMa is declined in two versions: a batch version for automatic usage such as benchmarking, or systematic discovery of patterns in a large application ; and an interactive version with a GUI which aims to be used in re-engineering, program comprehension or software maintenance. Our front-end is able to handle C and Fortran 90 programs. C front-end uses the LLVM compiler infrastructure [11], which is based on gcc front-end. Thus TeMa is able to handle any C real-life application. We have also implemented our own Fortran 90 front-end. Most Fortran 90 programs are correctly handled, but some syntactic constructions are not yet accepted, and need to be modified by hand. Our front-end has handled with success all fortran programs of SPEC benchmarking suite.

We have applied our slicing method to detect potential calls to the BLAS library [12] in LINPACK [7] and four programs involved in the SPEC benchmarking suite [8]. Our pattern base is constituted of direct implementations of BLAS functions from the mathematical description. Figure 9 shows the results.

It appears that 50% of candidates do not match, 25% are instances of patterns with one-dimension vectors, and 25% of candidates are correct and can be

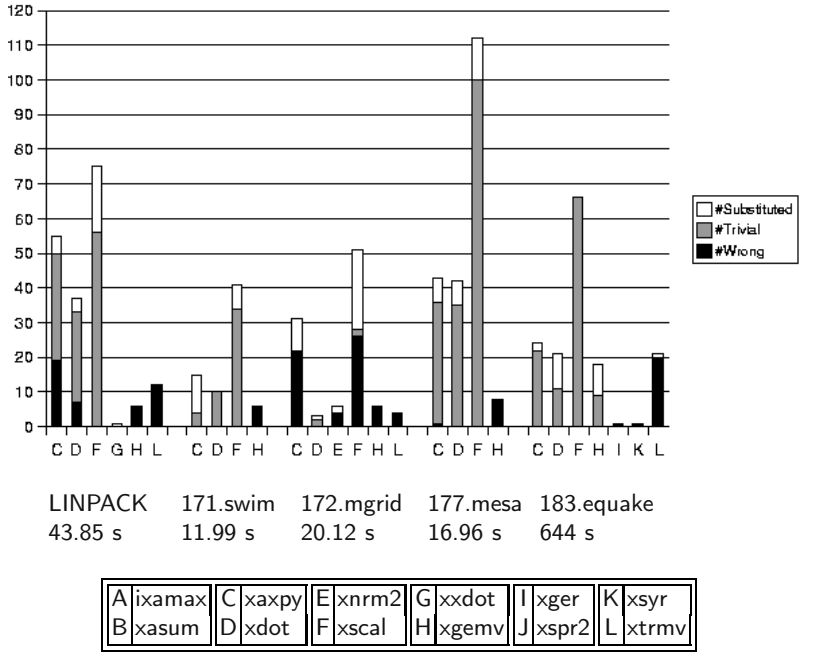


Fig. 9. For each SPEC program, we provide each BLAS function found, the number of non-equivalent slices (# Wrong), the number of equivalent slices with one statement (# Trivial), and the number of other equivalent slices (# Substituted). The execution times are given for a Pentium 4 1,8 GHz with 256 Mo RAM.

replaced by a call to BLAS. We present the different kind of candidates involved in these categories. Most of the incorrect detections are due to the approximation of the dependences with ϕ-functions. Neither loop iteration count, nor `if` conditions, nor complex dependences due to array index functions are taken into account.

25% of the slice is constituted of interesting candidates whose substitution can potentially increase the program performance. Our algorithm seems to have discovered all of them, and particularly hidden candidates. Indeed, most slices found are interleaved with the source code, and deeply destructured. Our method has been able to detect a dot product in presence of a splitting and a loop unroll, which constitute important program variations that a `grep` method would not catch. The same remark applies on `equake` program. Two versions of matrix-vector product appear, one hand optimized and the other not. Both are detected whereas a method based on regular expressions would detect only the second.

`TeMa` allows to recover SPL programs by recognizing and substituting SPL functions including matrix composition, direct sum and tensor product. Consider the following program, which is a naive implementation of the Walsh-Hadamard Transform (WHT):

```
      c(1) = f2
      do iter = 2,5
      | rank = 2 ** (iter - 1)
⊗     | do i = 0,1
⊗     | | do j = 0,1
⊗     | | | do k = 0,rank-1
⊗     | | | | do l = 0,rank-1
⊗     | | | | | c(iter,i*rank+k,j*rank+l) = f2(i,j)*c(iter-1,k,l)
⊗     | | | | enddo
⊗     | | | enddo
⊗     | | enddo
⊗     | enddo
      enddo
      wht = c(5)
```

`TeMa` detects the slice marked by ⊗ as a tensor product between `f2` and `c(iter - 1)`, and substitutes it in the following manner:

```
c(1) = f2
do iter = 2,5
| rank = 2 ** (iter - 1)
| c(iter) = f2 ⊗ c(iter-1)
enddo
wht = c(5)
```

Applying by hand our preliminary SPL code generation method, we finally obtain the following SPL program:

```
(tensor (F 2)(tensor (F 2)(tensor (F 2)(tensor (F 2)(F 2)))))
```

Even if the SPL generation is not yet implemented, the rewriting of the program with high-level functions increases readability, making `TeMa` a promising tool to improve program comprehension and help programmers in the tedious task of software maintenance.

5 Related work

We first present related work about program slicing as a tool to help software maintenance, then we present some methods for pattern detection, and more specifically *algorithm recognition*.

Program slicing was first introduced by Mark Weiser [19], to help programmers to debug their code. He defined a *slicing criterion* as a pair (p, V), where p is a program point and V a subset of program variables. A program slice on the slicing criterion (p, V) is a subset of program statements that preserves the behavior of the original program at the program point p with respect to the program variables in V. Weiser has shown that computing the minimal subset of

statements which satisfies this requirement is *undecidable* [19]. However an approximation can be found by computing consecutive sets of indirectly relevant statements, according to data-flow and control-flow dependences.

Cimetile et al. [3] defined a method to identify slices verifying given pre-conditions and post-conditions. They first compute a symbolic execution of the program, which assign to each statement its pre-condition, then they use a theorem prover to extract the slices. They need user interaction to associate post-condition variables to program variables. Moreover, as the problem of finding invariant assertions is in general *undecidable*, symbolic execution can require user interaction in order to prove some assertions and assert some invariants. No practical evaluation of their method, or theoretic study of complexity is given, but their method seems to be costly. Moreover, the need of user interaction makes the method inappropriate in a fully automatic framework.

Several approaches encode the knowledge about the functions to be identified in the form of programming plans, and can be classified as either top-down or bottom-up methods. Top-down methods [9,10] use the knowledge about the goals the program is assumed to achieve and some heuristics to locate both the program slice and the plan from the library which can achieve these goals. Bottom-up methods [13,20] start from the program statements and try to find the corresponding plans. Wills [20] represents programs by a particular kind of dependence graph called *flow-graph*, and patterns by *flow-graph grammar* rules. The recognition is performed by parsing the program's graph according to the grammar rules. She finally obtain a *parsing tree* which represents a hierarchical description of a plausible project of the program. This approach is a pure bottom-up code-driven analysis based on exact graph matching. Patterns are represented by grammars rules, encoding a hierarchy among them, but making the pattern base difficult to maintain. Organization variation is partially supported and temporary variables can be handled by adding specific rules. All others algorithmic variations can be handled only if they are explicitly described in the pattern base.

Metzger and Wen [14] have built a complete environment to recognize and replace algorithms. They first normalize the program and pattern AST by applying classical program transformations (if-conversion, loop-splitting, scalar expansion...). Then they look for good candidate slices within the program. The candidate slices are SCCs of the dependence graph, containing at least one `for` statement. Their equivalence test is based on an isomorphism between the slice and pattern AST. Obviously, this approach is low cost, and scalable. One may point out the large amount of candidate slices given by their method, but it is not a real problem due to the low complexity of their equivalence test. Organization variations, resulting from the permutation of independent statements or the introduction of temporaries are not handled by the algorithm itself, but by pre-treatments applied to the program. Reuse of temporaries accross loop iterations for instance is not handled. In the same way, the control variations supported are bounded to pre-treatments.

6 Conclusion

In this paper, we have presented an automatic method to find high-level functions in a given program, and its implementation in the tool `TeMa`. We have also proposed a preliminary approach to reformulate Fortran or C code into Signal Processing Language (SPL). Our detection method has been validated by recognizing BLAS functions in different kernels of the SPEC benchmarking suite. Our method is able to detect a large amount of program variations such as loop transformations (unroll, splitting, tiling, etc...) and appears to be scalable, and thus applicable to real-life applications. In addition, the rewriting of the program with high-level functions increases readability, making `TeMa` a promising tool to improve program comprehension and help programmers in the tedious task of software maintenance.

In future works, we would like to automatize the generation of SPL code, and validate it on benchmark applications. Additionnaly to portability and software maintenance, it should also increase performance since SPL enable specific algorithmic optimizations, which were not possible at a lower level of semantics.

References

1. C. Alias and D. Barthou. Algorithm recognition based on demand-driven dataflow analysis. In *10th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, November 2003.
2. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *8th International Euro-Par Conference*, page 309. Springer, LNCS 2400, 2002.
3. A. Cimetile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.
4. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997. release October, 1rst 2002.
5. J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965.
6. J. E. Doner. Tree acceptors and some of their applications. *Journal of Comput. and Syst. Sci.*, 4:406–451, 1970.
7. J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474. Springer-Verlag, 1988.
8. J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
9. S.-M. Kim and J. H. Kim. A hybrid approach for program understanding based on graph-parsing and expectation-driven analysis. *Journal of Applied A.I.*, 12(6):521–546, September 1998.
10. W. Kozaczynsky, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. on S.E.*, 18(12):1065–1075, December 1992.
11. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'2004*, Palo Alto, California, Mar 2004.

12. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.
13. B. Di Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *IWPC'04*, pages 164–174. IEEE Computer Society Press, 1996.
14. R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.
15. S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
16. M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Perf. Computing and Applications*, 1(18):21–45, 2004.
17. J.W. Thatcher and J.B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 1965.
18. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem. *Mathematical System Theory*, 2:57–82, 1968.
19. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
20. L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992.
21. Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 298–308, 2001.

Bossa Nova: Introducing Modularity into the Bossa Domain-Specific Language

Julia L. Lawall[1], Hervé Duchesne[2],
Gilles Muller[2], and Anne-Françoise Le Meur[3]

[1] DIKU, University of Copenhagen, Denmark
[2] OBASCO Group, École des Mines de Nantes-INRIA, LINA, France
[3] Jacquard Group, LIFL/INRIA, Université des Sciences et Technologies de Lille, France

Abstract. Domain-specific languages (DSLs) have been proposed as a solution to ease the development of programs within a program family. Sometimes, however, experience with the use of a DSL reveals the presence of subfamilies within the family targeted by the language. We are then faced with the question of how to capture these subfamilies in DSL abstractions. A solution should retain features of the original DSL to leverage existing expertise and support tools.

The Bossa DSL is a language targeted towards the development of kernel process scheduling policies. We have encountered the issue of program subfamilies in using this language to implement an encyclopedic, multi-OS library of scheduling policies. In this paper, we propose that introducing certain kinds of modularity into the language can furnish abstractions appropriate for implementing scheduling policy subfamilies. We present the design of our modular language, Bossa Nova, and assess the language quantitatively and qualitatively.

1 Introduction

Domain-specific languages (DSLs) have been proposed as a solution to ease the development of programs within a program family. A DSL is designed according to the results of a domain analysis, and provides high-level, domain-specific abstractions that facilitate programming in the domain and enable verification of domain-specific properties. Such languages have made programming accessible to non-experts in areas as varied as web-services [1,4,31] and animation [10].

Despite the success of DSLs, such languages are limited by the scope of the initial domain analysis. The ease of programming with a DSL may, however, lead programmers in unanticipated directions. In particular, experience with a DSL may reveal subfamilies within the family targeted by the language. We must then consider how to extend the DSL to provide appropriate abstractions for capturing these subfamilies. A solution should maintain the character of the DSL, to leverage the expertise and support tools developed around the language.

One approach that can enable a DSL to adapt to unanticipated needs is to embed the language in an existing richer general-purpose language [10,14,29].

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 78–93, 2005.

The DSL can then inherit host-language features such as types, modules, and objects as the need arises. The inherited features, however, are determined by what the host language provides, and not by domain needs. Accordingly, code can be difficult to understand, because information is not structured according to its role in the domain, and difficult to verify, because the general-purpose nature of the inherited features precludes introducing constraints to ease verification. We propose instead that DSL extensions should be individually designed according to domain requirements and in harmony with existing abstractions of the DSL. The language can then be embedded or directly compiled, as convenient. We illustrate our proposal in the context of the Bossa DSL for process scheduling [19,20], which we extend with two forms of modularity.

The Bossa DSL. A process scheduler is the part of an operating system (OS) kernel that allocates the CPU to processes. Because the time at which a process gets access to the CPU affects the timing of all its subsequent actions, process scheduling has a profound effect on application behavior. From real-time systems to multimedia applications to energy-restricted embedded systems and beyond, applications have varied scheduling needs that cannot be met by a single scheduler. Not surprisingly, many scheduling policies have been proposed [2,6,9,15,24,25,28,30,33,34,35]. Still, few of these scheduling policies are available in commonly used OSes. Furthermore, implementing a scheduling policy in a legacy OS kernel is outside the expertise of most application developers.

To ease the implementation of scheduling policies in legacy OSes, we have developed the Bossa framework. Bossa extends a legacy OS with a documented scheduling interface [19] and provides a DSL for implementing scheduling policies [20]. It has been used to implement a variety of scheduling policies, including those for interactive, multimedia, and real-time applications. We have observed significant benefits in the understandability, conciseness, and safety of Bossa schedulers, as compared to direct coding at the OS level. These features have enabled undergraduate students with no previous kernel programming experience to implement schedulers in the Linux kernel without crashing the machine.

The Bossa DSL was designed to facilitate the implementation of one scheduling policy at a time. The ease of scheduler programming in Bossa, however, has lead us to begin implementing an encyclopedic multi-OS library of scheduling policies. This work has highlighted some properties of scheduling policies that were not taken into account in the original design of the language. We have observed that the policies found in the scheduling literature are often classified in families, and that the Bossa implementations of policies within a family have much in common. Furthermore, the code specific to a policy variant is intertwined with the code generic to the family, making it difficult to identify policy-specific features. These issues have called for the introduction of modularity in the Bossa DSL, to enable the separation of concerns and to enhance code reuse.

This paper. In this paper, we address the needs identified in implementing an encyclopedic multi-OS library of scheduling policies by extending the Bossa DSL with two forms of modularity: *modules* and *transition aspects*. Modules separate scheduling concerns while transition aspects permit a module to adapt other

modules in a controlled way. The extensions are designed according to a careful analysis of the requirements of the scheduling domain. We assess the extensions on the implementation of a variety of scheduling policies. As compared to an embedded-language approach where the DSL inherits features from the host language, we find that our approach leads to policies that are more understandable, because information is structured according to the needs of the domain, and more verifiable, because we can constrain the module system in a way that eases verification. The resulting language, Bossa Nova, has been implemented by translation into the Bossa DSL, for which an implementation has previously been developed [19,20]. Experiments with Bossa Nova have shown no performance overhead as compared to Bossa.

This work represents a case study in what happens when a DSL meets real programming needs. Specifically, we illustrate:

- Motivations for introducing new abstractions into a DSL.
- Goals that should be taken into account in designing these abstractions.
- The choice of specific features that the abstractions should provide to meet these goals.

While the design choices presented here are specific to Bossa, our contributions are to identify individual motivations, goals, and features that should be taken into account in extending DSLs and to illustrate the benefits that can be achieved by this approach.

The rest of this paper is structured as follows. Section 2 presents the Bossa DSL. Section 3 motivates the need for modularity, and presents our design choices. Section 4 assesses the resulting language, Bossa Nova, on numerous examples and compares the proposed forms of modularity to existing approaches. Finally, Section 5 describes related work and Section 6 concludes.

2 Bossa in a Nutshell

We introduce the Bossa DSL using excerpts of an implementation of an Earliest-Deadline First (EDF) scheduling policy [7,22], shown in Figure 1. This policy manages a set of periodic processes, each of which is associated with a deadline within its current period. Process election chooses the process with the nearest deadline. The complete policy and a grammar of the Bossa DSL are available at the Bossa web site, `http://www.emn.fr/x-info/bossa`. We focus on the main features of the language: declarations and event handlers.

Declarations. The declarations of a scheduling policy define the process attributes, process states, and processes ordering used by the policy.

The `process` declaration (Figure 1, lines 2-3) lists the policy-specific attributes associated with each process. For the EDF policy, these are the period and the Worst-Case Execution Time (WCET) supplied by the process, a timer that is used to maintain the period, the offset of the deadline within each period, and the process's absolute deadline within the current period.

```
scheduler EDF = {                                                          1
  process = { time period; time wcet; timer period_timer;                  2
              time deadline; time absolute_deadline; }                     3
  states = { RUNNING running : process; READY ready : select queue;        4
             READY yield : process; BLOCKED blocked : queue;               5
             BLOCKED period_yield : queue; TERMINATED terminated; }        6
  ordering_criteria = { lowest absolute_deadline }                         7
  handler (event e) {                                                      8
    On block.* { e.target => blocked; }                                    9
    On bossa.schedule {                                                    10
      if (empty(ready)) { yield => ready; }                                11
      select() => running;                                                 12
      if (!empty(yield)) { yield => ready; }                               13
    }                                                                      14
    On unblock.timer.period_timer {                                        15
      e.target.absolute_deadline = now() + e.target.deadline;              16
      start_relative_timer(e.target.period_timer, e.target.period);        17
      switch e.target in {                                                 18
        case period_yield: {                                               19
          e.target => ready;                                               20
          if (!empty(running) && (e.target > running)) { running => ready; } 21
        }                                                                  22
        case running, ready: { e.target => ready; }                        23
        case READY, BLOCKED, TERMINATED: { }                               24
      }                                                                    25
    }                                                                      26
    ...                                                                    27
  }                                                                        28
}                                                                          29
```

Fig. 1. An excerpt of the EDF scheduling policy

The `states` declaration (lines 4-6) lists the set of process states that are distinguished by the policy. Each state is associated with a state class (`RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state. For example, the `ready` state is in the `READY` state class, meaning that it contains processes that are ready to run. A state is also associated with an implementation as either a process variable (`process`) or a queue (`queue`). Finally, the `ready` state is designated as `select`, indicating that processes are elected from this state.

The `ordering_criteria` (line 7) describes how to compare two processes in terms of a sequence of criteria based on the values of their attributes. Higher or lower values are favored using the keywords `highest` and `lowest`, respectively. The EDF policy favors the process with the lowest absolute deadline.

Event handlers. Event handlers describe how a policy reacts to scheduling-related events that occur in the kernel. Examples of such events include process blocking and unblocking and the need to elect a new process.

The EDF policy defines 11 event handlers. Handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event, if there is one. The event-handler syntax is based on that of a subset of C, to make the language easy to learn. The syntax provides specific constructs and primitives for manipulating processes and their attributes. These include constructs for testing the state of a process (*exp* `in` *state*), testing whether there is any process in a given state (`empty(`*state*`)`), testing the relative priority of two processes (exp_1 `>` exp_2), and changing the state of a process (*exp* `=>` *state*).

A `block.*` event occurs when a process blocks. The associated handler of the EDF scheduling policy (line 9) simply sets the state of the target process to `blocked`. A `bossa.schedule` event occurs when the kernel would like the policy to elect a new process. The associated handler (lines 10-14) uses the primitive `select()` to choose the highest priority process according to the ordering criteria from the state designated as `select`, *i.e.*, `ready`. It also manages any yielded process. Finally, an `unblock.timer.period_timer` event occurs when a process's `period_timer` timer expires, indicating the start of a new process period. The associated handler (lines 15-26) resets the absolute deadline of the target process (line 16), restarts the timer (line 17), and reschedules the target process for its computation in the new period (lines 18-25). If the process is in the `period_yield` state, meaning that it has completed its computation during its previous period, then its state is changed to `ready`, indicating that it is newly able to run (line 20). If the target process is in the `running` state or the `ready` state, then it is repositioned in the `ready` queue according to its new priority (line 23).

3 Modularity for Bossa

In the Bossa DSL presented above, a scheduling policy is implemented as a single unit, defining a complete set of event handlers. In our experience in developing a library of scheduling policies, we have observed that when scheduling policies are part of the same family, there is much commonality between their implementations. We first illustrate this commonality in the case of policies for managing periodic processes, such as EDF, and argue that this commonality motivates the need for modularity. We then propose a module system tailored to the needs of scheduling policies, which forms the basis of a modular variant of the Bossa DSL, named Bossa Nova. Finally, we briefly describe a second form of modularity, a variant of aspects, that we have also found useful in Bossa Nova.

3.1 The Need for Modules

Many scheduling policies have been developed for managing periodic processes, including Deadline Monotonic (DM) [7], Earliest-Deadline First (EDF) [7], Fixed Utilization Priority (FUP) [13], Least Compute Time (LCT) [13], Least-Laxity First (LLF) [7], Rate Monotonic (RM) [7], and Shortest Completion Time (SCT) [13]. The set of periodic policies thus amounts to a program subfamily. In implementing these policies in Bossa, we have observed that only the ordering criteria

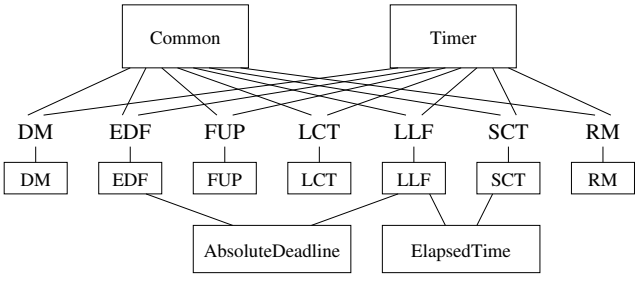


Fig. 2. Modular decomposition of a subset of the family of periodic policies (modules are boxed, policies are unboxed)

and the code calculating the values used in this criteria differ among them. Indeed, among these policies, the average size of the Bossa implementation is 123 lines, of which 100 are common to all of the policies.

One strategy in the face of this large amount of common code is to implement a scheduling policy by copying code from the Bossa implementation of another policy in the same subfamily. Nevertheless, we find that code that is common to the subfamily is mixed with code that is specific to a given policy, requiring careful rewriting of the copied code. This issue suggests the need for a modular programming strategy that separates these concerns, leading to a collection of standard modules that are useful in implementing policies of a given subfamily. Such a modular decomposition of the periodic policies is illustrated in Figure 2.

3.2 Modules for Process Scheduling

The design of our module system is guided by the requirements of the scheduling domain. As a scheduling policy is a critical component of an OS, its implementation must be understandable and verifiable. Our experience in implementing a library of scheduling policies has further shown the need for code reuse. Accordingly, we structure the module system according to the following principles. To enhance understandability, the module system organizes a scheduling policy as a centralized *scheduler*, giving a global view of the policy behavior, and a collection of modules, each implementing a single scheduling functionality. To enhance verifiability, the module system provides fine-grained control over external access to module elements, making it clear where it is valid to reason in terms of properties local to a module. Finally, to enhance reusability, modules do not refer directly to the other modules making up a given policy. Instead, information about the relationships between modules is localized in the scheduler.

In the rest of this section, we describe how the requirements of understandability, verifiability, and reusability influence the design of the interaction between the module system and the main features of Bossa: process states, process attributes, and event handlers. In each case, the various constraints identified

```
scheduler EDFSched = {                                                    1
  states = { RUNNING running : process; READY ready : select queue;       2
             READY yield : process; BLOCKED period_yield : queue;         3
             BLOCKED blocked : queue; TERMINATED terminated; }            4
  modules { EDF(),                                                        5
            AbsoluteDeadline(),                                           6
            Timer (running, ready, period_yield),                         7
            Common (running, ready, yield, blocked, terminated) }         8
  process { EDF.period reads Timer.period,                                9
            EDF.wcet reads Timer.wcet,                                    10
            EDF.absolute_deadline reads AbsoluteDeadline.absolute_deadline, 11
            AbsoluteDeadline.period_timer reads Timer.period_timer }      12
  ordering_criteria { EDF }                                               13
  handler { unblock.timer.period_timer : AbsoluteDeadline, Timer; }       14
}                                                                         15
```

Fig. 3. The scheduler used in the Bossa Nova implementation of the EDF policy

are checked by the Bossa Nova compiler. We use the Bossa Nova implementation of the EDF scheduling policy shown in Figures 3 and 4 as an example.

Process states. A main activity of a scheduling policy is to adjust process states, taking into account both OS requirements and the strategy of the policy. Thus, the set of process states manipulated by a policy gives a sense of the scope of the policy's scheduling strategies. To provide a global view of the policy, we define the states centrally in the scheduler, as shown in lines 2-4 of Figure 3. States are then passed to the modules as needed, as shown in lines 5-8 of Figure 3.

A module may only explicitly refer to the states among its parameters. A state change operation *exp* `=>` *state*, however, implicitly references the current state of the process *exp*. We allow this state to be any state defined by the scheduler. This strategy implies that the module does not have to be aware of the complete set of states defined by the policy, and thus facilitates code reuse, but limits the ability to reason about state contents across the handlers of a module. When a module needs to be sure that only it can affect the set of processes in a given state, it can annotate the associated parameter as `unshared`. Such a parameter must be instantiated by the scheduler to a state that is not passed to any other module and no other module can remove processes from the state. An example of such a parameter is `period_yield` (Figure 4, line 16), in which the Timer module stores processes that have completed their computation within a given period. The state used by the scheduler to instantiate this parameter (Figure 3, line 7) is only passed to this module. A global analysis of the scheduling policy shows that no other module removes processes from this state.

Process attributes. Process attributes record process information that persists across successive events. Because this information is typically specific to a single scheduling functionality, process attributes are declared locally to the mod-

```
module EDF() {                                                          1
  process = { requires time period; requires time wcet;                 2
              requires time absolute_deadline; }                        3
  ordering_criteria = { lowest absolute_deadline }                      4
}                                                                       5
module AbsoluteDeadline() {                                             6
  process = { time deadline; time absolute_deadline; requires timer period_timer; }  7
  handler (event e) {                                                   8
    On unblock.timer.period_timer {                                     9
      e.target.absolute_deadline = now() + e.target.deadline;           10
      next();                                                           11
    }                                                                   12
  }                                                                     13
}                                                                       14
module Timer(RUNNING process running, READY select queue ready,        15
             BLOCKED unshared queue period_yield) {                     16
  process = { time period; time wcet; timer period_timer; }             17
  handler (event e) {                                                   18
    ...                                                                 19
    On unblock.timer.period_timer {                                     20
      start_relative_timer(e.target.period_timer, e.target.period);     21
      switch e.target in {                                              22
        case period_yield: {                                            23
          e.target => ready;                                            24
          if (!empty(running) && e.target > running) { running => ready; }  25
        }                                                               26
        case running, ready: { e.target => ready; }                     27
        case READY, BLOCKED, TERMINATED: { }                            28
      }                                                                 29
    }                                                                   30
  }                                                                     31
}                                                                       32
module Common(...) { ... }                                              33
```

Fig. 4. The modules used in the Bossa Nova implementation of the EDF policy

ule defining the functionality. To facilitate communication between modules, all process attributes are implicitly exported for read access. To enable reasoning about the behavior of a module across its various handlers, however, write access is only allowed in the defining module. Finally, to enhance reusability, a module imports an attribute without mentioning the name of the defining module, by simply annotating the attribute declaration with `requires`. The link between exported and imported attributes is made in the scheduler.

As shown in lines 2-3 of Figure 1, the Bossa implementation of the EDF policy declares five attributes, relating to the management of the process period (`period`, `wcet`, and `period_timer`) and the process deadline (`deadline` and `absolute_deadline`). In the Bossa Nova implementation (Figure 4), the

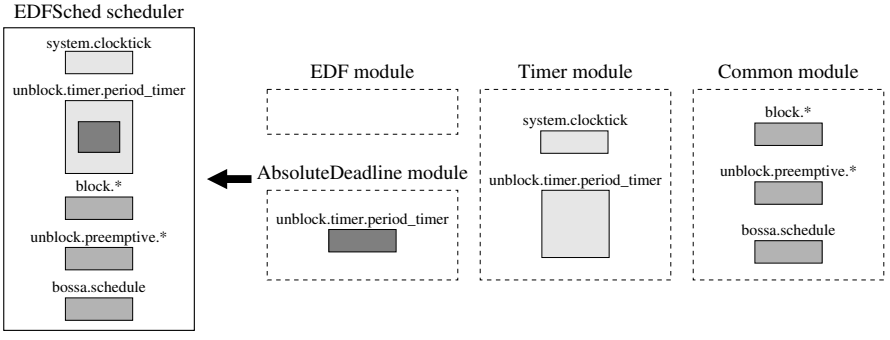


Fig. 5. Composition of handlers in the EDF scheduling policy

former are localized in the Timer module and the latter are localized in the AbsoluteDeadline module. The EDF module imports the `period`, `wcet`, and `absolute_deadline` attributes, which it declares using `requires`, as shown in lines 2-3 of Figure 4. These attributes are instantiated in the `process` declaration of the scheduler (Figure 3, lines 9-11), which declares that the EDF `period` and `wcet` attributes read the corresponding attributes of the Timer module and the EDF `absolute_deadline` attribute reads that of the AbsoluteDeadline module.

Event handlers. Event handlers react to OS events. Because multiple scheduling functionalities may need to react to the same event, multiple modules may define a handler for a given event. In this case, the scheduler lists the names of the modules defining a given handler in the order in which the definitions are to be composed. Execution begins with the handler defined by the first module in the list. A handler uses `next()` to invoke the next handler in the composition.

Figure 5 illustrates the composition of some of the handlers in the EDF policy. Both the Timer module and the AbsoluteDeadline module define an `unblock.timer.period_timer` handler. That of the Timer module (Figure 4, lines 20-30) represents a complete implementation of a minimal handling of the event: it restarts the timer and reschedules the target process. This handler is thus at the end of any composition sequence. The handler of the AbsoluteDeadline module (Figure 4, lines 9-12) extends this behavior by updating the locally defined `absolute_deadline` attribute and then invoking the next handler.

There are some constraints on the use of `next()` to ensure the integrity of the individual modules and of the composition. A handler typically updates process attributes and changes process states. When these operations are essential to the implemented scheduling functionality and are not idempotent, it is essential that the handler be invoked exactly once. To meet this requirement, a handler that appears before the end of a composition sequence must use `next()` exactly once along every control-flow path. In exceptional cases, a module may simply provide a default definition for a handler, but not require that this definition be used. Such a handler can be declared to be `overrideable`. If all subsequent handlers in a composition are declared as `overrideable`, then `next()` may be

omitted along some or all control-flow paths. Finally, handlers used at the end of a composition sequence may not invoke `next()`. This constraint ensures that the module writer intended the handler to be used in this position.

3.3 Aspects for Process Scheduling

Modules isolate the data and code associated with a given scheduling functionality. We have found, however, that the data associated with a scheduling functionality may need to be updated in response to actions, such as state changes, that take place in other modules. For example, the module ElapsedTime, used by several periodic policies (see Figure 2), maintains the running time of each process. Obtaining this value requires storing the current time whenever the process enters the `RUNNING` state and recording the difference between the current time and the stored time whenever the process leaves this state. Such state changes can occur in any module. As it is not desirable to let other modules make arbitrary side effects to the process attributes of ElapsedTime, the ElapsedTime module itself must be able to adapt other modules with the appropriate computations.

The need to update an attribute value when executing a particular kind of code elsewhere in the policy implementation amounts to a crosscutting concern. Accordingly, we look to Aspect-Oriented Programming [18] for inspiration, and add a form of aspects to Bossa Nova. We refer these aspects as *transition aspects*, because they account for some kind of transition in the system. A transition aspect implementing the behavior required by ElapsedTime on state transitions is as follows, where the state class names refer to the associated sets of states:

```
transition(process p) = {
  On READY => RUNNING { p.start_time=now(); }
  On RUNNING => READY, BLOCKED { p.elapsed_time+=now()−p.start_time; }
}
```

Transition aspects are similar to aspects in languages such as AspectJ [17], but are restricted to the updating of attributes in response to current conditions. Accordingly, an aspect can only be attached to a state change or attribute reference, and cannot itself perform state changes. When multiple aspects apply to a single construct, they are ordered such that an aspect that defines an attribute appears before all aspects that reference the attribute; mutually recursive references are rejected by the Bossa Nova compiler. This ordering ensures that each attribute reference obtains an up-to-date value.

4 Evaluation

We now evaluate the forms of modularity provided by Bossa Nova. We first consider the benefits of modularity in programming scheduling policies and then compare the dedicated approach used in Bossa Nova to the approaches to modularity found in general-purpose languages.

Family	Periodic				Round Robin	Proportion
Module	Common	Timer	AbsoluteDeadline	ElapsedTime	RR	Proportion
Lines of code	68	47	28	45	35	29

		Policy-specific module	Scheduler	Modular	Monolithic
Periodic	DM	23	22	160	109
(sharing illustrated in Figure 2)	EDF	26	34	203	123
	FUP	20	27	162	110
	LCT	9	26	150	106
	LLF	45	39	272	161
	SCT	42	35	237	147
	RM	9	26	150	106
	Family total			503	862
Round Robin	Basic round robin	15	28	146	96
(sharing Common and RR)	Best [3]	74	30	207	158
	Family total			182	254
Proportion	Basic proportion	48	32	177	124
(sharing Common and Proportion)	Move-to-rear [6]	41	29	167	123
	Family total			179	247

Fig. 6. A comparison of the lines of code used in the modular and monolithic implementations of various scheduling policies. "Family total" is explained in the text.

4.1 Benefits of Modularity in Implementing Scheduling Policies

We evaluate Bossa Nova with respect to a selection of policies from our on-going development of an encyclopedic, multi-OS library of scheduling policies. All of these policies are available at the Bossa web site.

Code sharing. We use the families of periodic, round-robin, and proportional scheduling policies to illustrate the effect of modularity on the amount of code that must be written to implement a new policy in a given family. Figure 6 shows the number of lines of code in the shared modules, and in a variety of scheduling policies in these families. All the policies use modules; the SCT, LLF, and Best policies also use transition aspects. In each case, the modular implementation is around 50% larger than the monolithic implementation. This increase is due to the introduction of the scheduler and the repetition of keywords (`process`, `handler`, *etc.*) between modules. In general, the extra code is eliminated by the Bossa Nova compiler, which generates a monolithic Bossa DSL implementation.

To amortize the cost of creating generic modules, they must be used by many different policies. The "Family total" entry associated with each family in Figure 6 shows the total number of lines in the policy-specific modules and schedulers added to the number of lines in one instance of each of the generic modules used by the family. This value excludes the Common module, which we may assume to be sufficiently widely used that its amortized cost is negligible. For the periodic family, the total number of lines that must be implemented in the modular case is 58% of the total number of lines required in the monolithic case. For the round-robin and proportional families, the ratio is 72%, reflecting the fact that fewer policies have been implemented in these families.

Separation of concerns. Even when a scheduling policy is not part of a family, modularity can be useful to separate concerns. The Borrowed Virtual Time

policy provides scheduling for both real-time and interactive processes [9]. This policy uses three main process attributes: the actual virtual time (AVT), the effective virtual time (EVT), and the warp. These attributes depend on several other process attributes, and the relevant calculations appear in multiple event handlers, making the monolithic implementation long (almost 300 lines) and difficult to understand. In the Bossa Nova implementation, each of the AVT, EVT, and warp is managed by a separate module. These modules highlight how the attributes are computed and the relationships between them. For the AVT, a twelve-line computation is required to compute the value whenever a process becomes newly able to run. This code is isolated in a transition aspect that delimits the computation and makes explicit the conditions under which it applies.

Isolation of OS-specific behavior. Often the details of the interaction with the target OS are orthogonal to the concerns of a given scheduling policy. In this case, we can use a module to isolate OS-specific behavior, thus simplifying the policy implementation and making it easy to use a scheduling policy with multiple OSes. In the examples in this paper, the Common module encapsulates the interaction with the OS (see Figure 2). We have implemented this module for use with Linux 2.4. In this implementation, the treatment of unblocking and yielding is specific to this OS, while other operations are generic.

4.2 Comparison to the Approaches of General-Purpose Languages

We have designed module and aspect systems specific to the problem of implementing scheduling policies, rather than reusing existing approaches. To justify our choice, we compare our module and aspect systems to existing general-purpose approaches, in terms of the understandability, verifiability, and reusability of scheduling code.

Understandability. Key to the understandability of a Bossa Nova scheduling policy is the scheduler, which gives an overview of the set of modules used by the policy, the information that is defined by each module, and the information that is shared between modules. Module systems that can provide such a global view include Units [11] and a variant of CLOS mixins [5], in which a module either defines new behaviors or describes how to combine the information provided by other modules. Both of these approaches, however, allow a combining module to be itself combined with other modules, and thus there may be no single unit that gives a complete view of the program. Furthermore, in both approaches, a combining module cannot declare data, as we require for the declaration of process states. Other approaches, such those found in Object-Oriented languages or ML [23], allow modular units to be created and used throughout the program and thus provide no global view. While one could organize programs written in such languages so that a single module creates the connections between all of the other modules, there would be no guarantee that this style is respected.

Verifiability. The verifiability of a Bossa Nova scheduling policy is enhanced by the constraints on access to process states, process attributes, and event handlers, that make it possible to reason about module behavior across a sequence

of events. General-purpose module systems either forbid external access to module information, *e.g.* using Java's `private` modifier, or allow unlimited external access, *e.g.* using Java's `public` modifier. There is, however, no built-in way to provide read and write access in the defining module but only read access in other modules or to constrain the number of invocations of a given function.

Reusability. The reusability of the modules of a Bossa Nova scheduling policy is enhanced by the property that a module does not explicitly mention the names of other modules. In Object-Oriented languages, relationships between classes are expressed using inheritance, which requires naming the superclass. Traits [27], CLOS mixins, and Units allow defining modules that are externally combined, and thus do not mention the names of other modules. Traits, however, does not allow modules to define local state, and CLOS mixins and Units do not provide a unique global view.

Aspects. Aspect ordering is a major problem in understanding, verifying, and reusing aspects defined using traditional approaches. For example, AspectJ relies on a combination of defaults based on the order in which aspects are declared and explicit directives [17]. These approaches are fragile, burden the programmer, and do not take the semantics of the aspects into account. In our approach, aspect ordering is determined by def-use relationships reflecting the semantics of the aspects and the needs of the domain.

In summary, while some general-purpose approaches to modularity provide some of the features that we require for Bossa Nova, none provides either the domain-specific distinctions between different kinds of values or the fine-grained control over the use of program entities that we require.

5 Related Work

To illustrate the strategies taken to incorporate advanced language features in DSLs, we consider some other DSLs that provide module systems.

Some embedded DSLs inherit the module system of the host language. Leijen and Meijer embed a language for constructing database queries in Haskell [21]. They argue that it is possible to exploit the Haskell module system, but their use of this module system does not exhibit any domain-specific properties. Elliot takes a similar approach in a DSL for animation [10]. Other embedded DSLs provide domain-specific module systems. Verischemelog is a hardware description language embedded in Scheme [16]. As this language targets Verilog users, it explicitly provides a module system based on that of Verilog, rather than using that of the host Scheme implementation. Thus, Verilog's use of modules is compatible with our approach: language abstractions are designed according to domain needs rather than relying on what is provided by a host language.

Module systems have also been developed from scratch for DSLs, as we have done for Bossa Nova. Risla is a compiled DSL for use in banking applications [32]. After several years of use, the abstraction facilities of the language were found to

be insufficient and the language was extended with a module system. The task of implementing the extension was facilitated by the use of language specification tools. The nesC DSL for networked embedded systems was designed from the start around the use of components, implemented using a dedicated module system [12]. NesC applications have been found to require little new code, instead relying on a large number of small components, suggesting the appropriateness of the module system and the overall language design to the targeted domain.

6 Conclusion

In this paper we have presented the Bossa Nova language, which provides modules and transition aspects for implementing scheduling policies. These forms of modularity enable substantial code reuse when implementing multiple scheduling policies within a single family, allow separation of concerns in complex policies, and separate policy-specific code from OS-specific details. Furthermore, the use of forms of modularity dedicated to the scheduling domain improves the understandability and verifiability of scheduling code as compared to the use of approaches found in general-purpose languages. These observations suggest that rather than inheriting language features, as done in an embedded language, it is more fruitful to construct DSL extensions directly, based on an analysis of the needs of the domain.

We have used the language to extend our library of scheduling policies with policies from a range of policy families, including both classical policies and policies developed in recent research. In on-going work, we are adding to the set of policies and families represented. As a practical example, we are currently applying Bossa Nova and our library of scheduling policies to the use of a standard PC as a Personal Video Recorder (PVR).[1] A PVR must provide a variety of video services, such as encoding, decoding, and picture-in-picture. These services need to maintain a specific rate, but may have unpredictable computation requirements. Existing PVR software does not provide any quality of service guarantees and indeed less is known about how a scheduling policy can provide such guarantees for processes with unpredictable computation requirements than for processes with strict computation bounds. The ease of generating new policy variants and combinations in Bossa Nova can aid in evaluating existing policies and new variants in this setting.

References

1. D. L. Atkins, T. Ball, G. Bruns, and K. C. Cox. Mawl: A domain-specific language for form-based services. *IEEE Trans. Software Eng.*, 25(3):334–346, 1999.
2. A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, Phoenix, AZ, Dec. 1999.

[1] `http://www.emn.fr/x-info/bossa/bossabox`

3. S. A. Banachowski and S. A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Multimedia Computing and Networking*, volume 4673, San Jose, CA, Jan. 2002.
4. C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Trans. Internet Tech.*, 2(2):79–114, 2002.
5. G. Bracha and W. R. Cook. Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, Oct. 1990.
6. J. L. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, pages 63–73, Seattle, WA, Nov. 1997.
7. F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.
8. *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99)*, Austin, TX, Oct. 1999.
9. K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.
10. C. Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Trans. Software Eng.*, 25(3):291–308, 1999.
11. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, June 1998.
12. D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.
13. C. Guss. Lecture notes: ECSE-421 embedded systems, 2004. `http://http://www.ece.mcgill.ca/~info421/`.
14. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), Dec. 1996.
15. K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, Dec. 1998.
16. J. Jennings and E. Beuscher. Verischemelog: Verilog embedded in Scheme. In DSL99 [8], pages 123–134.
17. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001.
18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, LNCS 1241, pages 220–242, Jyväskylä, Finland, June 1997.
19. J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third International Conference on Generative Programming and Component Engineering*, LNCS 3286, pages 436–455, Vancouver, Canada, Oct. 2004.

20. J. L. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies (invited paper). In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04*, pages 80–91, Verona, Italy, Aug. 2004.
21. D. Leijen and E. Meijer. Domain specific embedded compilers. In DSL99 [8], pages 109–122.
22. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.
23. D. B. MacQueen. An implementation of Standard ML modules. In *LISP and Functional Programming*, pages 212–223, Snowbird, Utah, July 1988.
24. J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, Saint-Malo, France, Oct. 1997.
25. J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [26], pages 141–148.
26. *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.
27. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, LNCS 2743, pages 248–274, Darmstadt, Germany, July 2003.
28. Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference*, pages 134–139, New Orleans, LA, June 1999.
29. O. Shivers. A universal scripting framework, or Lambda: the ultimate "little language". In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, LNCS 1179, pages 254–265, Singapore, Dec. 1996.
30. D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 145–158, New Orleans, LA, Feb. 1999.
31. P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium*, LNCS 2257, pages 192–208, Portland, OR, Jan. 2002.
32. M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, LNCS 1101, pages 9–18, Munich, Germany, July 1996.
33. D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Trans. Netw.*, 5(4):475–488, Aug. 1997.
34. W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 149–163, Bolton Landing, NY, Oct. 2003.
35. W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [26], pages 149–156.

AOP++: A Generic Aspect-Oriented Programming Framework in C++*

Zhen Yao, Qi-long Zheng, and Guo-liang Chen

National High Performance Computing Center at Hefei,
Department of Computer Science and Technology,
University of Science and Technology of China
Hefei, Anhui 230027, China
yaozhen@ustc.edu, qlzheng@ustc.edu.cn, glchen@ustc.edu.cn

Abstract. This paper presents AOP++, a generic aspect-oriented programming framework in C++. It successfully incorporates AOP with object-oriented programming as well as generic programming naturally in the framework of standard C++. It innovatively makes use of C++ templates to express pointcut expressions and match join points at compile time. It innovatively creates a full-fledged aspect weaver by using template metaprogramming techniques to perform aspect weaving. It is notable that AOP++ itself is written completely in standard C++, and requires no language extensions. With the help of AOP++, C++ programmers can facilitate AOP with only a little effort.

1 Introduction

Aspect-oriented programming (AOP)[1] is a new programming paradigm for solving the code tangling problems of object-oriented programming (OOP) by separating concerns in a modular way. Many crosscutting concerns that cannot be expressed by entities such as *classes* or *functions* in OOP can be abstracted and encapsulated into *aspects*. The so called *aspect code* and *component code* can be decoupled cleanly instead of tangled together. Aspects can have influence upon the component code in many ways. For instance, aspects can change the static structure of the component code by *introductions*, as well as change the dynamic behaviour by *advices*. The influence is injected by the *aspect weaver*. First the weaver identifies points in component code where aspects are to be inserted, which are called *join points*. Several join points can form a *pointcut* expression which is declared by each aspect to specify its scope. Then the weaver weaves the aspect code into every join point of the component code.

In existing AOP systems such as AspectJ[13] for Java and AspectC++[7] for C++, the aspect code is often written in a meta level language which is different from (usually a superset of) the language used by the component code.

* The work of this paper is funded by Intel HiEd HPC Research Project under grant 4507146713, and Science Foundation of Anhui, China under grant 050420205.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 94–108, 2005.

That means AOP usually requires language extensions as well as special AOP-aware (pre-)compilers, therefore aspects and components cannot be expressed in a uniform manner.

Though both C++ and Java are object-oriented programming languages, C++ is very different from Java in many aspects. C++ has many characteristic language constructs such as template, multiple inheritance and operator overloading. They are not optional features of the language, but indispensable parts that make C++ a harmonious whole. Template mechanisms and subsequent generic programming (GP)[5][9] and template metaprogramming techniques[20][10] enable the application of generative programming[12] concepts in C++ to create active libraries[19] supporting multiple programming paradigms[14] including OOP, GP and even functional programming (FP)[2][3]. Different paradigms in C++ can cooperate with each other harmoniously to solve complex programming problems in a more natural manner.

The C++ language is so complicated that even some commercial C++ compilers fail to support all its features (especially the complex template mechanisms) well. The syntax and semantics of C++ is too complex to add new language extensions, especially significant radical extensions such as aspect-orientation. We should not expect an AOP-aware (pre-)compiler to behave better than professional C++ compilers at dealing with complex generic components. Even so, it would be less likely to persuade developers to learn and accept the language extensions and use a specific AOP-aware compiler instead of their favorite standard C++ compilers.

AOP++ presented in this paper is a generic aspect-oriented programming library/framework for C++ that adopts an approach which is quite different from that used by AspectJ or AspectC++. It is remarkable that the aspect weaver and all the aspect code are completely written in standard C++. No extra language extension is required, so no proprietary AOP-enabled compiler is ever needed. With the help of AOP++, C++ programmers can facilitate AOP with only a little effort.

AOP++ can be considered as an active library that defines a new domain-specific sublanguage for AOP and extends the C++ compiler's ability through its aspect weaver. It makes heavy use of complex template metaprogramming techniques to perform aspect weaving at compile time within the framework of standard C++.

AOP++ is tightly coupled with the C++ language. As a consequence, all language features, especially template mechanisms and generic programming are supported intrinsically by AOP++. That means it can apply AOP to the realm of GP paradigm such as STL containers and generic components from other modern C++ template libraries. In fact, the aspect code itself is written as C++ templates and is generic by nature.

The infrastructure of AOP++ is depicted in Fig. 1. It is remarkable that aspect weaving all takes place in standard C++ compiler by metaprograms. Details will be described in later sections.

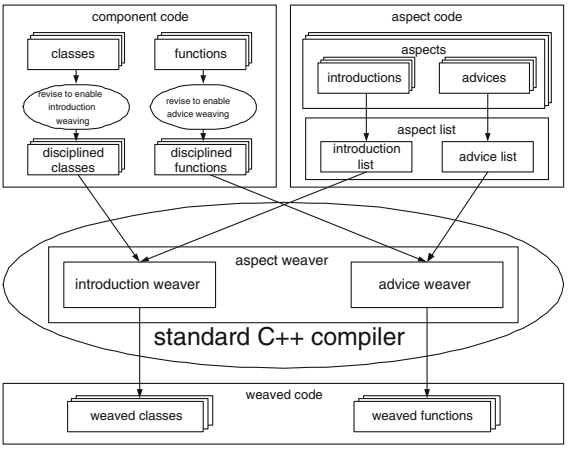


Fig. 1. Infrastructure of the AOP++ Framework

The rest of this paper is organized as follows: Section 2 gives an overview of the AOP++ framework, including its overall infrastructure, disciplines of the component code and definitions of pointcut expressions and introductions / advices / aspects; A typical example is presented in Sect. 3 to demonstrate the power of AOP++; Implementation approaches are explained in Sect. 4; Section 5 gives a brief discussion of related work; Finally, Section 6 summarizes the paper and gives some directions of the possible future work.

2 The AOP++ Framework

2.1 Overview

AOP++ is mainly composed of the following parts:

Pointcut Expression is an important building block of AOP++ for identifying join points in the component code. There are two categories of pointcut expression: *type pattern* represents a collection of types, which is used to specify the scope of introductions; while *method pattern* represents a collection of functions, which is used primarily to specify the scope of advices. Both type pattern and method pattern can be defined recursively. *Type operators* can be applied to existing ones to build up composite type patterns or method patterns.

Base Classes for Aspects are the implementation basis for the user-defined aspect code. They provide basic mechanisms for the aspect code to interact with the component code through a collection of reflection API.

Aspect Weaver is the core of AOP++. It does aspect weaving at compile time. Two weavers are included: the *introduction weaver* weaves introductions into user-defined classes, while the *advice weaver* weaves advices into user-defined functions.

2.2 Disciplines of the Component Code

In order to make AOP++ work, the component code must be written according to some simple disciplines.

2.2.1 Class Definitions

User-defined classes must be defined in a special way to enable introduction weaving, that is, to inherit from template class `aop::introd`. The declaration looks like this:

```
class SimpleClass : public aop::introd<SimpleClass> { /* ... */ };
```

If a user-defined class originally inherits from other class(es), for example:

```
class ComplexClass
    : public A, public B, private C { /* ... */ };
```

The disciplined code looks like this:

```
class ComplexClass : public aop::introd<ComplexClass>
    ::public_bases<A, B>::private_base<C> { /* ... */ };
```

2.2.2 Function Implementations

There are two ways to discipline function implementations to enable advice weaving in AOP++.

The common way is planting a local hook variable of type `aop::advice`, `aop::ctor_advice` or `aop::dtor_advice` in the function body, with function type (and name) provided as template parameters, while function arguments are provided to the constructor of the hook variable. Here are some examples:

```
void MyClass::mf(int i) {
  // do not forget the implicit "this" argument
  aop::advice<void (MyClass::*)(int), &MyClass::mf> hook(this, i);
  // original implementation code
}
void f(double d, const MyClass& c) {
  aop::advice<void (double, const MyClass&), f> hook(d, c);
  // original implementation code
}
MyClass::MyClass(int i) {
  aop::ctor_advice<MyClass(int)> hook(this, i);
  // original implementation code
}
MyClass::~MyClass() {
  aop::dtor_advice<MyClass()> hook(this);
  // original implementation code
}
```

There is another way to enable advice weaving for functions in dynamic linked libraries (or shared libraries). This style of advice weaving is also performed at compile time but is *pluggable* at runtime, we call it *pluggable advice weaving*. For example, a member function `void Dynamic::mf()`'s implementation in file "dynamic.cpp" is compiled into a dynamic linked library. The disciplined code can be put in another file named "dynamic_aop.cpp":

```
void Dynamic::mf()
{ return aop::dynamic_advice<void (Dynamic::*)(), &Dynamic::mf>(this); }
```

2.3 Type Pattern

Type pattern in AOP++ is a mechanism to represent collections of types at compile time.

A type pattern can be simply a type (atomic type pattern), or it can be defined recursively using `or_type`, `and_type`, `not_type`, `derived` or `const_or_not`, which are called type operators. For example, the type pattern that matches any type which is derived from `ClassA` or `ClassB` is

```
aop::or_type<aop::derived<ClassA>, aop::derived<ClassB> >
```

The same type pattern can be represented in AspectJ as `ClassA+ || ClassB+` and in AspectC++ as `derived(ClassA) || derived(ClassB)`.

There is also a special wildcard type pattern `any_type` which obviously matches any type, and a type pattern `null_type` which matches no type. In addition, AOP++ provides several predefined type patterns which are frequently used, such as `integral_type`, `class_type`, `arithmetic_type` and `abstract_type`, etc.

Type patterns in AOP++ have a peculiar capability that is lacking in other AOP frameworks to represent template types. When applying AOP to C++ libraries such as the STL, we need to represent concept like "`std::vector<T>` where `T` is any derived type of `MyClass`", and it is easy to write out the type pattern in AOP++:

```
std::vector<aop::derived<MyClass> >
```

AOP++ provides a mechanism called compile-time lambda expression[10] to presents type patterns which impose specific relationships between template parameters. For example, we can express the type pattern "`std::pair<T, T>` where the two `T`'s are the same." in AOP++ as follows:

```
std::pair<_, _>
```

Note that the predefined type patterns described previously all correspond to lambda type pattern expressions composed of type traits in the Boost Type Traits Library[4]. For instance, `aop::integral_type` and `boost::is_integral_type<_>` are equivalent type patterns.

AOP++ also provides a mechanism to do type pattern matching. Users can determine whether a type is contained in a type pattern expression at compile time by using `aop::matches` template:

```
aop::matches<TypePattern, Type>::value
```

`value` is of type `bool` that evaluates to `true` or `false` at compile time.

The introduction weaver uses a similar way to determine whether a class is in the scope of an introduction.

2.4 Method Pattern

Method pattern can be viewed as a special kind of type pattern that concerns with functions or overloaded operators.

AOP++ uses several helper class templates to wrap up functions to types, so as to manipulate them more easily.

Given a global function `f` and a member function `A::mf`:

```
void f(int i, std::string s);
void A::mf(int i, std::string);
```

The corresponding method patterns can be defined as following:

```
aop::method<void (int, std::string), f>
aop::method<void (A::*)(int, std::string), &A::mf>
```

Because of that there is no type to represent constructors or destructors directly in C++, AOP++ uses an alternative way to define method patterns for them. Given:

```
Point::Point(int x, int y);
Point::~Point();
```

Their corresponding method patterns are:

```
aop::ctor<Point (int, int)>
aop::dtor<Point ()>
```

In AspectJ, the same constructor can be represented as "`Point.new(int, int)`".

Wildcards can be used for method names in method patterns as well. The pattern below matches "any non-static non-const member function defined in class `MyClass` or any of its derived classes":

```
aop::methods<aop::any_type (aop::derived<MyClass>::*)(...)>
```

It is equivalent to "`* MyClass+.*(..)`" in AspectJ or "`% derived(MyClass)::%(...)`" in AspectC++.

It is possible to specify a collection of member functions with exactly the same name. The macro below defines a method pattern named `draw_methods` that matches "any non-static non-const member function named '`draw`' in class `Component` or any of its derived classes, which takes a reference to `Graphics` as its parameter and has no return value":

```
AOP_DEFINE_METHODS(void (derived<Component>::*)(Graphics&), draw,
                   draw_methods);
```

Similar to type patterns, method patterns can be combined by applying type operators `and_type`, `or_type` and so forth to form composite method patterns.

Type pattern and method pattern in AOP++ are simplified pointcut concepts. There is no distinct `call`, `execution`, `within` or `cflow` and so forth pointcuts in AOP++. AOP++ simply uses method pattern to represent the execution of functions.

2.5 Introduction

Introduction is used for modifying user-defined classes and their hierarchies. It changes the static structure of the component code at compile time. Introduction can introduce new base classes for user-defined classes, or add new members (member variables or member functions) to them.

A user-defined introduction in AOP++ is a class template which is derived from template class `aop::introd_base`. Extra base classes, member variables and member functions to be introduced into user-defined classes can be specified by just declaring base classes, member variables and member functions of the user-defined introduction class itself respectively.

An important part of user-defined introduction is an inner type named `scope`, which specifies the scope within which the introduction takes effects in terms of a type pattern.

For example, the code listed below shows how to declare a user-defined introduction named `MyIntrod` which introduces an extra base class, a member variable and a member function to the user-defined classes `A` and `B`:

```
template <typename Arg>
struct MyIntrod : aop::introd_base<Arg>
                    ::public_base<IntroducedBaseClass> {
  typedef aop::or_type<A, B> scope;

  int              introduced_member_variable;
  void             introduced_member_function();
  static const int introduced_static_member_variable = 0;
  static void      introduced_static_member_function();
};
```

There are also abstract introductions. An abstract introduction just leaves its scope definition empty, or holds pure virtual functions, waiting for derived introductions to complete the definition.

An introduction can be declared to "dominate" some other introductions by defining in its definition a type pattern named `dominates` which includes the dominated introductions as follows:

```
typedef aop::or_type<IntroductionA<Arg>, IntroductionB<Arg> > dominates;
```

That means, it will reside at a higher level than those dominated introductions in the introduction hierarchies generated by the introduction weaver (see Sect. 4 for detail).

2.6 Advice

Advice is used for defining additional code that should be executed at runtime. It changes the dynamic behaviour of the component code at runtime. However, advice weaving is done at compile time.

AOP++ currently supports three kinds of advices: *before*, *after* and *around*. Around advice is only available for pluggable advice weaving. All user-defined advices should be class templates that inherit from `aop::advice_base`. Inner type named `scope` is a method pattern specifying the functions to be advised.

The before, after and around methods in advice are member functions, each takes the same parameter list as that of the method to be advised, or a tuple which wraps up all the arguments by reference as its only parameter in situations when the advised methods have different signatures, or even takes no argument and gets these arguments via member functions of its `advice_base`. AOP++ will automatically choose the correct way to pass the actual arguments to the before, after and around methods. These arguments can be read or even modified in the before, after and around methods, to perform extra work or change the behaviour of the advised functions.

Below is a simple example illustrating how to define an advice containing before and after methods for tracing any AOP++-enabled functions.

```
template <typename Arg>
struct TracingAdvice : aop::advice_base<Arg> {
  typedef aop::any_type scope;

  void before()
  { clog << "TracingAdvice::before " << this->method_name() << endl; }

  void after()
  { clog << "TracingAdvice::after  " << this->method_name() << endl; }
};
```

There are also abstract advices. An abstract advice just leaves its scope definition empty, or holds pure virtual `before` / `after` / `around` functions, waiting for derived advices to complete the definition.

An advice can be declared to dominate some other advices by defining in its definition a type pattern named `dominates` which includes the dominated advices as follows:

```
typedef aop::or_type<AdviceA<Arg>, AdviceB<Arg> > dominates;
```

That means, it will precede those dominated advices in the advice chain generated by the advice weaver (see Sect. 4 for detail).

2.7 Aspect

Like data and functions can be encapsulated into a class, several introductions and advices can be encapsulated into a single aspect to emphasize their logical relation. Figure 2 shows an aspect that adds synchronization support for generic containers. We take `std::vector` for example here. (Suppose we could revise the standard containers to make them AOP++-enabled.)

```
template <typename Arg>
struct vector_monitor
     : aop::aspect_base<Arg> {
typedef recursive_read_write_mutex Mutex;
typedef recursive_read_write_lock  Lock;

struct monitorable : aop::introd_base<Arg> {
  typedef std::vector<aop::any_type> scope;

  mutable Mutex mutex;
};

struct read_monitor : aop::advice_base<Arg> {
  typedef aop::methods<aop::any_type
   (std::vector<aop::any_type>::*)(...) const>
   scope;

  Lock lock;

  read_monitor()
      : lock(this->this_object->mutex)
  {}
  void before() { lock.read_lock();  }
  void after()  { lock.unlock();     }
};

struct write_monitor : aop::advice_base<Arg> {
  typedef aop::methods<aop::any_type
    (std::vector<aop::any_type>::*)(...)>
    scope;

  Lock lock;

  write_monitor()
      : lock(this->this_object->mutex)
  {}
  void before() { lock.write_lock(); }
  void after()  { lock.unlock();     }
};

typedef aop::type_list<
    monitorable,
    read_monitor,
    write_monitor> aspect_list;
};
```

Fig. 2. The Synchronized Aspect for `vector`s

We can also specify aspect precedence by defining a type pattern named `dominates` which includes the dominated aspects. If an aspect "dominates" another aspect, that means all introductions and advices in it dominate those in the other.

2.8 Reflection

It is important for aspects to have the ability interacting with the corresponding component code. AOP++ provides a rich reflection API through `introd_base` and `advice_base` for the purpose. Aspect programmers can access type information of the component code, get arguments of the method being advised and so on. The APIs can be divided into two categories — compile time reflection and runtime reflection.

2.9 Putting It All Together

Once the component code is correctly disciplined and all the introductions, advices and aspects are defined, we need to tell AOP++ which of them are expected to take effect on the component code:

```
namespace aop {
  typedef template_list<
            Introd1, Introd2, ...,
            Advice1, Advice2, ...,
            Aspect1, Aspect2, ...
          > aspect_list;
}
```

Only introductions / advices / aspects in the `aop::aspect_list` will be woven into the component code. Users can maintain different `aspect_list` configurations for different projects, and define a preprocessor macro `AOP_ASPECTS` to specify the header including the expected `aspect_list`. Aspect weaving can also be disabled by defining a null `aspect_list` or a preprocessor macro `AOP_DISABLE`.

3 Example: Implementing the Observer Pattern

In this section, we demonstrate how to use AOP++ to implement the famous *observer pattern* described in [8]. Given classes `FigureElement`, `Point`, `Line` and `Canvas` as in Fig. 3.

```
class FigureElement {
public:
  virtual void setXY(int, int) = 0;
  virtual ~FigureElement();
};

class Point : public FigureElement {
  int _x;
  int _y;
public:
  Point(int x, int y);
  void setXY(int x, int y);
  void setX(int x);
  void setY(int y);
  int x();
  int y();
};
```

```
class Line : public FigureElement {
  Point p1;
  Point p2;
public:
  Line(int x1, int y1, int x2, int y2);
  Line(const Point& p1, const Point& p2);
  void setXY(int x, int y);
  void setP1(const Point &p);
  void setP2(const Point &p);
};

class Canvas {
  std::list<FigureElement*> elements;
public:
  void addFigureElement(FigureElement* fe);
  ...
};
```

Fig. 3. Definition of `FigureElement` and Its Derived Classes

```
template <typename _Observer>
struct Subject {
  typedef _Observer Observer;
  typedef std::list<Observer*> ObserverList;

  void attach(Observer* obs) {
    observers.push_back(obs);
  }

  void detach(Observer* obs) {
    observers.remove(obs);
  }

  void notify() {
    typedef
      typename ObserverList::iterator
      iterator;

    for (iterator it = observers.begin();
         it != observers.end(); ++it) {
      (*it)->update(static_cast<
        typename Observer::Subject*>(this));
    }
  }

  virtual ~Subject() {}

private:
  ObserverList observers;
};

template <typename _Subject>
struct Observer {
  typedef _Subject Subject;

  virtual void update(Subject* subject) = 0;
  virtual ~Observer() {}
};
```

Fig. 4. Generic Components for the Observer Pattern

```
template <typename Arg>
struct MoveMethods {
  AOP_DEFINE_METHODS(void (aop::derived<FigureElement>::*)(int, int),
                                                  setXY, setXY_method);
  AOP_DEFINE_METHODS(void (Point::*)(int),         setX,  setX_method);
  AOP_DEFINE_METHODS(void (Point::*)(int),         setY,  setY_method);
  AOP_DEFINE_METHODS(void (Line::*)(const Point&), setP1, setP1_method);
  AOP_DEFINE_METHODS(void (Line::*)(const Point&), setP2, setP2_method);

  typedef aop::or_type<
      setXY_method, setX_method, setY_method, setP1_method, setP2_method> scope;
};
```

Fig. 5. The Move Events of `FigureElements`

```
template <typename Arg>
struct SubjectObserverProtocol
     : aop::aspect_base<Arg> {

struct SubjectIntrod
     : aop::introd_base<Arg>
         ::public_base<Subject<Canvas> > {
  typedef FigureElement scope;
};

struct StateChangedAdvice
     : aop::advice_base<Arg> {
  typedef
      typename MoveMethods<Arg>::scope
      scope;

  void after()
  { this->this_object->notify(); }
};

struct ObserverIntrod
     : aop::introd_base<Arg>
     ::public_base<Observer<FigureElement> > {
  typedef Canvas scope;

  virtual void update(FigureElement* fe) {
    // update the canvas according to fe
  }
};

struct SubjectAddedAdvice
     : aop::advice_base<Arg> {
  AOP_DEFINE_METHODS(
      void (Canvas::*)(FigureElement*),
      addFigureElement, scope);

  typedef aop::advice_base<Arg> base;
  typedef typename base::arg_list arg_list;

  void after(arg_list& args) {
    // attach the Canvas to the FigureElement
    aop::arg<1>(args)
        ->attach(this->this_object);
  }
};

typedef aop::type_list<
    SubjectIntrod,
    StateChangedAdvice,
    ObserverIntrod,
    SubjectAddedAdvice> aspect_list;
};
```

Fig. 6. The Aspect for implementing the Observer Pattern

A `Canvas` holds a list of `FigureElement` objects and is responsible for rendering them. Now we want the `Canvas` to be notified to update its display whenever any one of its `FigureElement` is moved. Obviously the `Canvas` acts as the observer, while the `FigureElement` and its derived classes act as the subjects.

The first step is to discipline the above classes to be AOP++-enabled.

The second step is to define generic components supporting the observer pattern, which is shown in Fig. 4. This approach is similar to those used in the Loki Library[6].

Before writing aspects using AOP++, let's define the pointcut expression that denotes the move event of a `FigureElement` in Fig. 5.

Now the aspects can be written as in Fig. 6.

It is also possible to make the concrete subject and observer classes as well as the state-changed pointcut as template parameters so as to generalize the

above `SubjectObserverProtocol` aspect to accommodate general situations. The generalized aspect is called a *parameterized aspect*.

Several other design patterns such as the *visitor pattern*[8] can also be applied in this manner. The generic components are similar to those in the Loki Library, while their integration with user-defined components is automatically done by reusable parameterized aspects in an aspect-oriented manner.

4 Implementation

AOP++ makes extensive use of template metaprogramming in its implementation.

Pointcut expressions including type patterns and method patterns are all represented using the C++ type system. Type patterns are defined by simple types and type operators, also with some predefined shortcuts for defining often-used type patterns. Method patterns are method pointers wrapped up in `method`, `methods`, `ctor` or `dtor` templates.

There are two aspect weavers in AOP++, the introduction weaver and the advice weaver. A mechanism similar to the template `aop::matches` in Sect. 2.3 is used to determine whether a join point is covered by the scope of an introduction or advice. The following is a simplified description of the work flow of the aspect weaver.

Introduction weaving takes place while defining the base class(es) for a user-defined type. It first filters all introductions (both standalone and those embedded in aspects) from the `aop::aspect_list`, then determines whether the scope of an introduction covers the current join point (that is, the user-defined class which is being defined). If the answer is yes, the introduction is relevant and will be woven into the definition of the class; otherwise it is simply discarded. Then all relevant introductions will be instantiated with the current join point in their template parameter using a class template similar to `Loki::GenLinearHierarchy` template[6], and form a linearized class hierarchy in which the introductions are lined up and inherit one another (the order will be influenced by the domination declarations of introductions) with the user-defined base classes at the top of the hierarchy while the class being defined at the bottom. Hence extra base classes, member functions and member variables are "injected" into user-defined classes.

Consider a class `C` defined as follows:

```
class C : public aop::introd<C>::public_bases<A, B> { /* ... */ };
```

Suppose three introductions `Introd1`, `Introd2` and `Introd3` are injected into class `C`. The resulting inheritance structure generated by our introduction weaver is shown in Fig. 7.

Advice weaving takes place while defining the extra hook variable (which uses the scope guard idiom) of a user-defined method. The weaver first filters all advices (both standalone and those embedded in aspects) from the

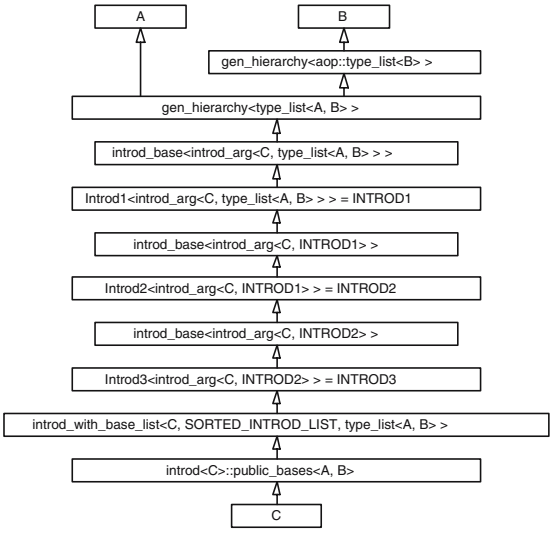


Fig. 7. The Generated Inheritance Structure of Class `C`

`aop::aspect_list` whose scope covers the current join point (that is, the function being defined), and creates a list of all the relevant advices (called an *advice chain*)in the corresponding `aop::(dynamic_)advice`, `aop::(dynamic_)ctor_advice` or `aop::(dynamic_)dtor_advice` classes. The constructor of the advice class will call the `around` and `before` member functions of every relevant advice one by one in order (the order will be influenced by the domination declarations of the advices), the destructor will call corresponding `after` member functions in reverse order.

The arguments of the method are passed to constructor of the hook variable, from which they will then be passed to every `before` / `after` / `around` member functions of the advices automatically in appropriate manner.

5 Related Work

AOP++ invents a brand-new approach to support aspect-oriented programming paradigm in C++. By using template metaprogramming techniques, AOP++ creates a full-fledged aspect-oriented programming framework which does not depend on any language extensions or privileged AOP-aware compilers. All concepts and components in AOP++ are all built on standard C++ constructs. This makes it easy to be accepted by C++ programmers.

AOP++ distinguishes itself from several previous approaches to simulate AOP in C++[16][17][21] by its characteristic aspect weaver.

The previous approaches [16] and [17] exploit techniques similar to *mixin-based programming*[18] to wrap up existing component with mixin layers, while in [21], the component class must be declared as a template with an extra "Aspect"

template parameter. In all of them, the aspect user has to declare which aspects are desired for each class by defining an aspects list for every user-defined class *explicitly* and *manually*. the aspect list can be long and the manual definition is error-prone and cumbersome. Furthermore, all code that refers to the user-defined classes has to be changed to use the classes wrapped up with aspects explicitly. The most important crosscutting nature of AOP cannot be expressed.

On the contrary, aspect weaving in AOP++ is done *implicitly* and *automatically* at compile time behind the scenes by the aspect weaver according to the scope definition of each aspect. The component programmer need not concern about what aspects will be injected into which user-defined classes or functions at all.

6 Conclusion and Future Work

The main contributions of AOP++ are:

1. It innovatively makes use of C++ templates to express pointcut expressions and match join points at compile time.
2. It innovatively creates a full-fledged aspect weaver by using C++ template metaprogramming techniques to perform aspect weaving.
3. It successfully extends and applies AOP to the GP paradigm in standard C++. It bridges the gap between AOP and GP. This includes dual meaning: (1) GP techniques can be used in the aspect code and (2) AOP++ makes it possible to apply AOP to generic components in modern C++ template libraries.

Due to limitations of the C++ language itself, there are also some limitations of AOP++, such as that the weaved code is a structural and behavioural approximation to what is expected, but not exactly the same, and it is difficult to implement some advanced features such as join points for field access, privileged aspects, etc.

The main limitation of AOP++ is that it is not 100% transparent to the component programmer. Existing component code has to be revised according to some disciplines in order to enable AOP++ to act on them, though the disciplines are simple and straightforward, and may be applied automatically by a simple pre-processor. We believe that this does not really contradict the philosophy of AOP which demands the separation of the component code and the aspect code. The reason is that the revision is done *just once* in a uniform manner regardless of whatever aspects will be woven later. The disciplines are the keys for bridging the component code and the aspect code. The benefit we achieved is that no language extensions are imposed upon programmers. In addition, all the client code which uses the component code needs not to be changed to enjoy the benefit of AOP++.

Our future work includes:

1. Further enhancement of AOP++, including more dynamic features such as support for `cflow`. It is likely that they will introduce more runtime overhead upon the user program.

2. Investigate the possibility of integrating support for AOP++ in modern IDEs to facilitate the development of AOP programs.
3. Study the development model of the AOP paradigm and the interactions between aspects in AOP++, construct reusable generic AOP libraries for tracing, debugging, performance profiling, program visualization and verification, concurrent programming, etc.
4. Study the relation between AOP and other new programming paradigms such as *explicit programming*[15] and try to combine them in the framework of AOP++.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In Proceedings of ECOOP 1997. Springer-Verlag. (1997)
2. McNamara, B., Smaragdakis, Y.: Functional Programming in C++. In Proceedings of the ACM SIGPLAN ICFP 2000. (2000)
3. The Boost Lambda Library. http://www.boost.org/libs/lambda/
4. The Boost Type Traits Library. http://www.boost.org/libs/type_traits/
5. Austern, M.: Generic Programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc. (1998)
6. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley. (2001)
7. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In Proceedings of TOOLS 2002. (2002)
8. Gamma, E., Helm. R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. (1994)
9. Stepanov, A., Lee, M.: The Standard Template Library. HP Technical Report HPL-94-34. (1995)
10. The Boost C++ Metaprogramming Library. http://www.boost.org/libs/mpl/
11. The Boost Array Library. http://www.boost.org/libs/array/
12. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley. (2000)
13. AspectJ. Home Page. http://www.aspectj.org/
14. Coplien, J.: Multi-Paradigm Design for C++. Addison-Wesley. (1998)
15. Bryant, A., Catton, A., Volder, K., Murphy, G.: Explicit Programming. In Proceedings of AOSD 2002. (2002)
16. Diggins, C.: Aspect-oriented programming & C++. Dr Dobbs Journal 29 (8) (2004)
17. Gal, A., Lohmann, D., Spinczyk, O.: Aspect-Oriented Programming with C++ and AspectC++. Tutorial held on AOSD 2004. (2004)
18. Smaragdakis, Y., Batory, D.: Mixin-Based Programming in C++. In Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers. Springer-Verlag. (2001)
19. Veldhuizen, T., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98). SIAM Press. (1998)
20. Veldhuizen, T.: Using C++ template metaprograms. C++ Report. 7(4). (1995)
21. Sunder, S., Musser, D.: A Metaprogramming Approach to Aspect Oriented Programming in C++. MPOOL'04 (ECOOP 2004)

Model Compiler Construction Based on Aspect-Oriented Mechanisms

Naoyasu Ubayashi[1], Tetsuo Tamai[2],
Shinji Sano[1], Yusaku Maeno[1], and Satoshi Murakami[1]

[1] Kyushu Institute of Technology, Japan
[2] University of Tokyo, Japan
{ubayashi, tamai}@acm.org
{sano, maeno, msatoshi}@minnie.ai.kyutech.ac.jp

Abstract. Model-driven architecture (MDA) aims at automating software design processes. Design models are divided into platform-independent models (PIMs) and platform-specific models (PSMs). A model compiler transforms the former models into the latter automatically. We can regard PIMs as a new kind of reusable software component because they can be reused even if a platform is changed. However, a generated PSM is useless if it does not satisfy system limitations such as memory usage and real-time constraints. It is necessary to allow a modeler to customize transformation rules because model modifications for dealing with these limitations may be specific to an application. However, current model compilers do not provide the modeler sufficient customization methods. In order to tackle this problem, we propose a method for constructing an extensible model compiler based on aspect orientation, a mechanism that modularizes crosscutting concerns. Aspect orientation is useful for platform descriptions because it crosscuts many model elements. A modeler can extend model transformation rules by defining new aspects in the process of modeling. In this paper, an aspect-oriented modeling language called AspectM (Aspect for Modeling) for supporting modeling-level aspects is introduced. Using AspectM, a modeler can describe not only crosscutting concerns related to platforms but also other kinds of crosscutting concerns. We believe that MDA is one of the applications of aspect-oriented mechanisms. The contribution of this paper is to show that a model compiler can actually be constructed based on aspect-oriented mechanisms.

1 Introduction

Model-driven architecture (MDA) aims at automating software design processes. Design models described in Unified Modeling Language (UML) are divided into platform-independent models (PIMs) and platform-specific models (PSMs). A model compiler transforms the former models into the latter automatically. The current MDA primarily focuses on platform-related issues. However, model transformations are not limited to these concerns, as in the case of application-specific optimization. A PSM generated by a model compiler is useless if the PSM does

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 109–124, 2005.

not satisfy system limitations such as memory usage and real-time constraints. It is necessary to allow a modeler to customize transformation rules because model modifications for dealing with these limitations may be specific to an application. It would be useful to apply the idea of active libraries[5] to model compiler construction. However, most current model compilers support only specific kinds of platforms, and do not provide the modeler sufficient customization methods.

This paper proposes a method for constructing an extensible model compiler based on aspect orientation[12] in order to tackle the above problem. Aspect orientation is a mechanism that modularizes crosscutting concerns as aspects. Platform descriptions also can be regarded as crosscutting concerns. For example, descriptions for conforming a model to a specific database middleware cut across many elements in a model. There are several reasons why adopting aspect-oriented mechanisms for describing database concerns is useful: persistence can be modularized; persistence aspects can be reused; and applications can be developed unaware of the persistent nature of the data[18]. The approach of [18] is effective not only at the programming level but also at the modeling level. In this paper, an aspect-oriented modeling language called AspectM (Aspect for modeling) is introduced for supporting modeling-level aspects. Using AspectM, a modeler can describe not only crosscutting concerns related to platforms but also other kinds of concerns related to model transformation. That is, a modeler can extend model transformation rules by defining new aspects in the process of modeling: that is, defining model transformation rules at the same level of ordinary modeling. Using AspectM, we can realize not only MDA but also techniques for supporting early aspects and crosscutting properties at the requirement-related and architectural levels[6]. MDA and aspect orientation are not different software development principles; rather, we believe that MDA is an application of aspect-oriented mechanisms. The contribution of this paper is to show that a model compiler can be actually constructed based on aspect-oriented mechanisms.

The remainder of this paper is structured as follows. In Section 2, we illustrate the process of model transformation using a simple example. In Section 3, we propose a method for model transformations based on aspect-oriented mechanisms. We introduce AspectM for supporting the method, and provide a technique for implementing AspectM in Section 4. We show a model transformation example using AspectM in Section 5. In Section 6, we evaluate AspectM qualitatively based on our experience. In Section 7, we introduce some related work, and discuss future directions of this research. Section 8 concludes the paper.

2 Motivation

Here, we illustrate typical model transformation steps in MDA, show how platform-specific concerns cut across model elements, and demonstrate how aspect-oriented mechanisms can be applied to describe these crosscutting concerns.

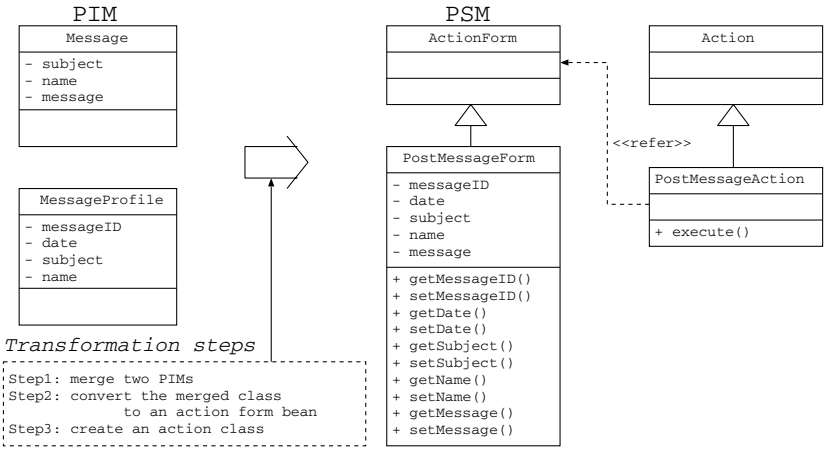


Fig. 1. An example of a model transformation

2.1 Model Transformation Steps in MDA

The steps of model transformation can be explained using the following simple bulletin board system as an example: *a user submits a message to a bulletin board, and the system administrator observes administrative information such as daily message traffic.* This system must be developed using the web application framework called Struts[22].

We define PIMs that do not depend on a specific platform, and transform these PIMs into PSMs targeted to Struts, the platform of this system. Figure 1 illustrates this transformation process. There are two PIMs in this example[1]. One is the `Message` class, and the other is the `MessageProfile` class. The former is a PIM defined from the viewpoint of a user. The latter, which includes administrative information such as message id and date, is a PIM defined from the viewpoint of a system administrator. Although these PIM classes represent different viewpoints in the system, the substance of the classes should be the same. All attributes included in the `Message` class and the `MessageProfile` class are necessary for handling a message. The following shows the steps of transformation of PIMs to a PSM.

Step 1: The two PIM classes, `Message` and `MessageProfile`, are merged into a single class whose name is `PostMessage`. Attributes/Operations that have the same name (signature) are merged into a single attribute/operation.

Step 2: In Struts, a request from a web browser is stored in an action form bean class. The `PostMessage` class is transformed to an action form bean class. First, the name of the `PostMessage` class is changed to `PostMessageForm` because an action form bean class must have a name ending with the string

[1] In general, PIMs and PSMs are described as sets of UML diagrams. In this example, we use only class diagrams for simplicity.

Form. Next, the parent class of the `PostMessageForm` is set to the `ActionForm` framework class because it is specified in Struts that a bean class must inherit the `ActionForm` class. After that, a set of accessors (setter/getter) is added to the `PostMessageForm` class. The transformations in Step 2 are needed for every data request. That is, the transformations cut across classes related to the requested data.

Step 3: In Struts, an action logic that handles a request from a web browser is defined as the `execute` operation in an action class. First, the action class `PostMessageAction` is created, and its parent class is set to the `Action` class prepared in Struts. Next, the `execute` operation is added to the `PostMessage-Action` class. The `execute` operation gets the data of the request from the corresponding action form bean class, and executes a business logic.

We can implement a model compiler that supports the above transformation steps because each step is clearly defined. We can also develop a new model compiler that supports other platforms. A series of PSMs can be generated from a single PIM by applying different model compilers. MDA enables us to shift from code-centric product-line engineering (PLE)[5] to model-centric PLE.

2.2 Advantages of Introducing Aspect Orientation

In this paper, we introduce aspect-oriented mechanisms for describing model transformation rules. As pointed out in the above, platform-specific descriptions are one of the crosscutting concerns that can be well dealt with by aspect orientation. Although MDA and aspect orientation at the modeling level are considered different technologies, they are closely related, as we claim in this paper. Applying aspect orientation to model compiler construction, we obtain the following advantages: a modeler can extend model transformation rules by defining new aspects in the process of modeling; a modeler can describe not only crosscutting concerns related to platforms but also other kinds of crosscutting concerns including optimization, persistence, and security; and aspect orientation can be applied at the modeling level, at which design information can be used to represent crosscutting concerns.

3 Applying Aspect Orientation to Model Compiler Construction

3.1 Aspect Orientation at the Modeling Level

Aspect-oriented programming (AOP), a modularization mechanism for separating crosscutting concerns, is based on the join point model (JPM) consisting of join points, pointcuts, and advice. Program execution points including method invocations and field access points are detected as join points, and a pointcut extracts a set of join points related to a specific crosscutting concern from all

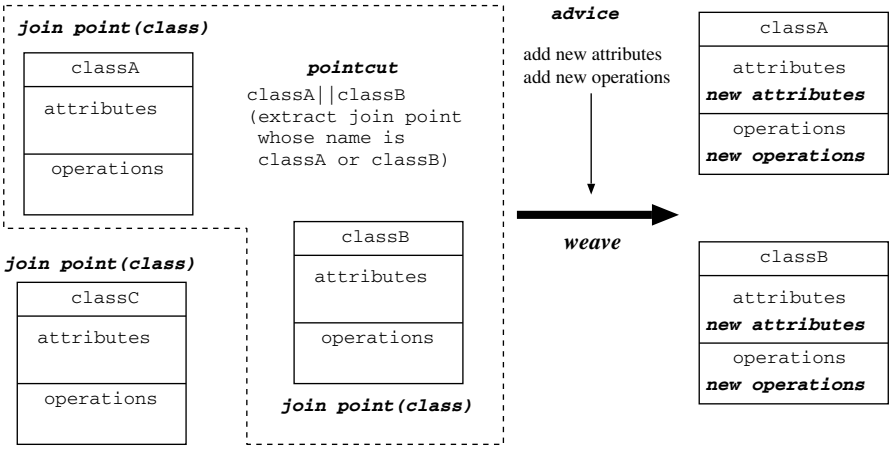


Fig. 2. Aspect orientation at the modeling level (example)

join points. A compiler called a weaver inserts advice code at the join points selected by pointcut definitions.

Although JPMs have been proposed as a mechanism at the programming level, they can be applied to the modeling level as shown in Figure 2. In this example, a class is regarded as a join point. The pointcut definition '`classA || classB`' extracts the two classes `classA` and `classB` from the three join points `class A`, `classB`, and `classC`. Model transformations such as *add new attributes* and *add new operations* are regarded as advice. In Figure 2, new attributes and operations are added to the two classes, `classA` and `classB`.

3.2 JPMs for Model Transformations

There has been research supporting modeling-level aspect orientation based on a specific AOP language such as AspectJ[13]: an aspect at the modeling level is converted to an aspect in AspectJ[21]. However, there are problems with these approaches: a PSM is limited to a specific AOP language; most current AOP languages are based on a few fixed set of JPMs; and we cannot separate cross-cutting concerns that cannot be separated by current AOP languages. Indeed, there are several kinds of JPMs as shown in [14]. In order to deal with this problem, multiple JPMs should be supported at the modeling level, and these JPMs should not correspond to specific kinds of AOP languages.

AspectM, an aspect-oriented modeling language proposed in this paper, supports six kinds of JPMs: PA (pointcut & advice), CM (composition), NE (new element), OC (open class), RN (rename), and RL (relation). Table 1 shows model transformation types and corresponding JPMs. With aspect composition based on these JPMs, the model transformation in Section 2 can be realized as shown in Table 2. Model elements including classes, methods, and relations specific to Struts are woven into the original PIMs.

Table 1. JPMs for model transformation

No	Model transformation type	PA	CM	NE	OC	RN	RL
1	change a method body	○					
2	merge classes		○				
3	add/delete classes			○			
4	add/delete operations				○		
5	add/delete attributes				○		
6	rename classes					○	
7	rename operations					○	
8	rename attributes					○	
9	add/delete inheritances						○
10	add/delete aggregations						○
11	add/delete relationships						○

Table 2. Model transformation steps for Struts

Step	Model transformation	PA	CM	NE	OC	RN	RL
step 1	1-1) merge `Message` and `MessageProfile` into `PostMessage`		○				
step 2	2-1) rename `PostMessage` to `PostMessageForm`					○	
	2-2) add an inheritance relation between `ActionForm` and `PostMessageForm`						○
	2-3) add accessors to `PostMessageForm`				○		
step 3	3-1) create an action class `PostMessageAction`			○			
	3-2) add an inheritance relation between `Action` and `PostMessageAction`						○
	3-3) add the `execute` method to `PostMessageAction`				○		
	3-4) add the body of the `execute` method	○					

PA is an AspectJ-like JPM. A join point is a method execution, and advice changes a behavior at join points selected by a pointcut. Three kinds of advice can be described: `before` (a pre-process is added), `after` (a post-process is added), and `around` (a process is replaced). PA is used when we want to add platform-specific logics to PIMs. CM is a Hyper/J-like JPM[4]. In this case, a join point is a class, and advice merges classes selected by a pointcut: operations with the same name are merged into a single operation, and attributes with the same name are merged into a single attribute. CM is used in the case of converting multiple PIM classes to a single PSM class. NE is a JPM for adding a new model element to a UML diagram. In this case, a join point is a UML diagram such as a class diagram. Advice adds a new class to a class diagram selected by a pointcut. NE can be used to add a platform specific class to PIMs. OC is a JPM for realizing the facility of an open class. In this case, a join point is a class, and advice inserts operations or attributes. OC, which is similar to an inter-type declaration in AspectJ, is used in the case of adding platform-specific operations or attributes to PIMs. Figure 2 in Section 3.1 is an example of an OC.

RN is a JPM for changing a name, in which a join point is a class, an operation, and an attribute. Advice changes the names of classes, operations, and attributes selected by a pointcut. RN is used for following the naming conventions specified in a platform. RL is a JPM for changing the relation between two classes, in which case, a join point is a class, and advice adds an inheritance, an aggregation, and a relationship between two classes selected by a pointcut. There is a case that a class must inherit a specific class defined in an application framework such as Struts. RL is used in this situation.

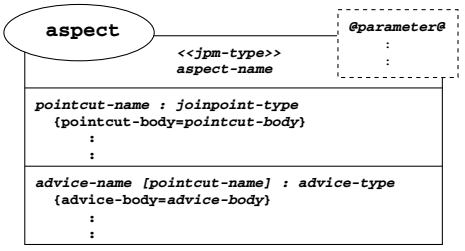
4 AspectM

AspectM is an aspect-oriented modeling language that supports the six JPMs introduced in Section 3. In AspectM, an aspect can be described in either a diagram or an XML (eXtensible Markup Language) format. AspectM is defined as an extension of the UML metamodel. Figure 3 shows the AspectM diagram notations and the corresponding XML formats. AspectM is not only a diagram language but also an XML-based AOP language. In this section, we show the syntax of AspectM, which has two aspects: an ordinary aspect and a component aspect. A component aspect is a special aspect used for composing aspects. In this paper, we use simply the term *aspect* when we need not to distinguish between an ordinary aspect and a component aspect. An aspect can have parameters for supporting generic facilities. By filling parameters, an aspect for a specific purpose is generated. Using these kinds of aspects, a set of transformation steps can be described as a generic software component.

4.1 Notation

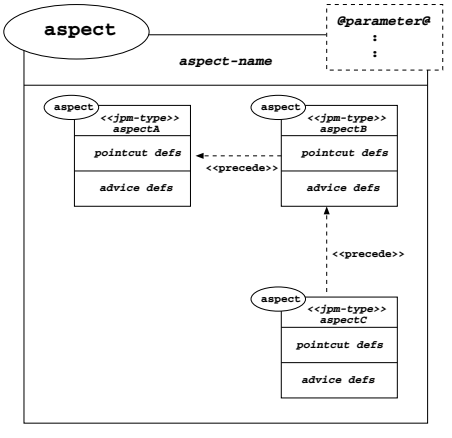
The notations of aspect diagrams are similar to those of UML class diagrams. An oval at the upper left portion of a diagram indicates that the diagram represents an aspect. A box at the upper right indicates parameters. This box can be omitted when there is no parameter. An aspect with parameters is called a *template*. Italic text in a diagram must be specified by a modeler. Types of JPMs (*jpm-type*), join points (*joinpoint-type*), and advice (*advice-type)* are shown in Table 3.

Diagrams of ordinary aspects are separated into three compartments: 1) aspect name and JPM type, 2) pointcut definitions, and 3) advice definitions. An aspect name and a JPM type are described in the first compartment. A JPM type is specified using a stereo type. Pointcut definitions are described in the second compartment. Each of them consists of a pointcut name, a join point type, and a pointcut body. In pointcut definitions, we can use three predicates: `cname` (class name matching), `aname` (attribute name matching), and `oname` (operation name matching). We can also use three logical operations: && (and), || (or), and ! (not). The following are examples of pointcut definitions: '`oname(setX) || oname(setY)`' (two operations `setX` and `setY` are selected from all join points); '`!aname(attribute*)`' (attributes not starting with a string `attribute` are selected); '`cname(classA) || cname(classB)`' (two classes `classA` and `classB` are selected); and '`cname(class*) &&oname(set*)`' (operations, which belong



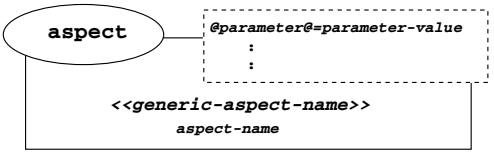

```
<aspect name=aspect-name type="ordinary"
        jpm=jpm-type>
  [<params>
    { <param> @parameter@ </param> } *
   </params>]
  { <pointcut name=pointcut-name
              type=joinpoint-type>
      <pointcut-body>
        pointcut-body
      </pointcut-body>
    </pointcut> } +
  { <advice name=advice-name
            type=advice-type
            ref-pointcut=pointcut-name>
     <advice-body>
        advice-body
     </advice-body>
    </advice> } +
</aspect>
```

a) ordinary aspect




```
<aspect name=aspect-name type="component" >
  [<params>
    { <param> @parameter@ </param> } *
   </params>]
  { definition of ordinary aspect
      or
    definition of component aspect} +
  { <aspect-precedence>
     <from>aspect-name</from>
     <to>aspect-name</to>
    </aspect-precedence> } +
</aspect>
```

b) component aspect




```
<aspect name=aspect-name
        template="template-name" >
  {<set-param name="@parameter@">
     parameter-value
   </set-param> } +
</aspect>
```

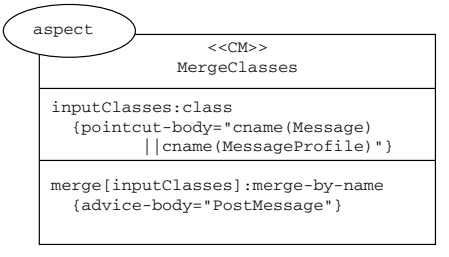
c) parameter setting

Fig. 3. AspectM diagram notations and XML formats

to classes starting with a string `class` and start with a string `set`, are selected if joinpoint-type is operation). Although we support only predicates for name matching in the current AspectM, we plan to support predicates including `is-attribute-of(`*class*`)`, `is-operation-of(`*class*`)`, `is-superclass-of(`*class*`)`,

Table 3. Types of JPM, join point, and advice

JPM type	Join point type	Advice type
PA	operation	before, after, around
CM	class	merge-by-name
NE	class diagram	add-class, delete-class
OC	class	add-operation, delete-operation add-attribute, delete-attribute
RN	class, operation, attribute	rename
RL	class	add-inheritance, delete-inheritance add-aggregation, delete-aggregation add-relationship, delete-relationship




```
<aspect name="MergeClasses"
        type="ordinary" jpm="CM">
  <pointcut name="inputClasses"
            type="class">
    <pointcut-body>
      cname(Message)||cname(MessageProfile)
    </pointcut-body>
  </pointcut>
  <advice name="merge" type="merge-by-name"
          ref-pointcut="inputClasses">
    <advice-body>PostMessage</advice-body>
  </advice>
</aspect>
```

Fig. 4. Example of aspect notation

and `is-subclass-of(`*class*`)`. Advice definitions are described in the third compartment. Each of them consists of an advice name, a pointcut name, an advice type, and an advice body. A pointcut name is a pointer to a pointcut definition in the second compartment. Advice is applied at join points selected by the pointcut. The left side of Figure 4 shows a transformation rule corresponding Step 1 in Section 2: the JPM type is CM; the two classes `Message` and `MessageProfile` are join points selected by the pointcut definition; and the merge-by-name type advice is applied at these join points.

Diagrams of component aspects are separated into two compartments: 1) aspect name, and 2) a set of ordinary aspects or component aspects. A component aspect consists of the aspects specified in the second compartment. A stereo type `<<precede>>` indicates the precedence of aspects as shown in Figure 3 b). This is important when multiple aspects are applied to the same join points.

By filling parameter values, an aspect for a specific purpose is generated. The name of a template is specified in a stereo type.

An aspect can be represented in XML format as shown in the right side of Figure 3. The notations [] and {} show an option and a repetition, respectively. The notations $*$ and $+$ in {} show an occurrence of more than zero and more than one. An aspect is represented by the `aspect` tag distinguished by the `type` attribute. A set of parameters is specified by the `params` tag. In an ordinary

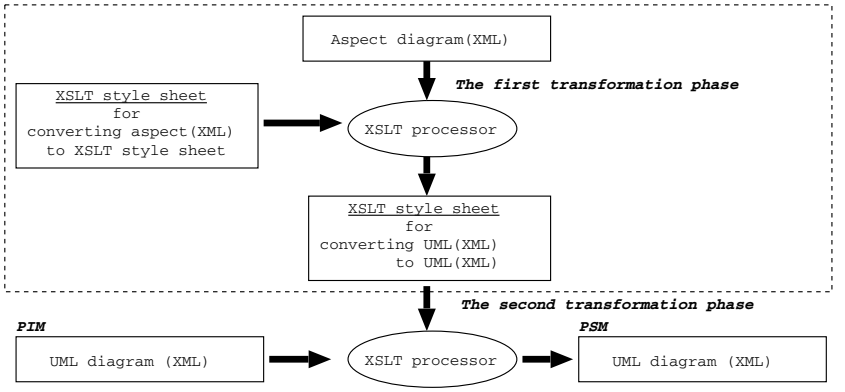


Fig. 5. Implementation of AspectM model compiler

aspect, pointcuts and advice are specified by the `pointcut` tag and `advice` tag, respectively. In a component aspect, definitions of ordinary aspects or other component aspects are specified after parameter definitions. After that, precedences of aspects are specified using the `aspect-precedence` tag. Parameters are set using the `set-param` tag. The right side of Figure 4 shows the XML descriptions corresponding to the diagrams on the left.

4.2 Implementation

We have developed a prototype of AspectM. The tool for supporting AspectM consists of a model editor and a model compiler. The model editor facilitates editing UML and aspect diagrams. The model editor can save diagrams in the XML format. The model compiler can be implemented as an XML transformation tool because UML class diagrams can be represented in XML. The AspectM model compiler, which consists of two phases, transforms PIM classes into PSM classes as shown in Figure 5. The first transformation phase converts an aspect in the form of XML to an XSLT (XSL Transformation) style sheet with additional Java classes using an XSLT processor. The second transformation phase converts PIM classes in XML form to the corresponding PSM classes in XML form using the style sheet generated in the first transformation phase.

5 MDA with AspectM

5.1 Aspect Descriptions for Model Transformations

Figure 4 shows Step 1 of transformation in the bulletin board system. In this step, the `MergeClasses` aspect, whose JPM type is CM, is defined for merging two PIM classes `Message` and `MessageProfile` into the `PostMessage` class.

Although the `MergeClasses` aspect in Figure 4 is useful, there is a problem in terms of reusability because the aspect cannot be applied to other models. The

pointcut body and the advice body are specific to the bulletin board system. In order to deal with the problem, a generic mechanism can be used. The following is a generalized version of the `mergeClasses` in XML form. A string enclosed by '@' is a parameter.

```
;; generic MergeClasses aspect
<aspect" name="Generic-MergeClasses" type="ordinary" jpm="CM">
  <params>
    <param>@input-classes@</param>
    <param>@merged-class@</param>
  </params>
  <pointcut" name="inputClasses" type="class">
    <pointcut-body>@input-classes@</pointcut-body>
  </pointcut>
  <advice name="merge" adviceType="merge-by-name"
          ref-pointcut="inputClasses">
    <advice-body>@merged-class@</advice-body>
  </advice>
</aspect>
;; specific mergeClasses aspect
<aspect name="MergeClasses" template="Generic-MergeClasses">
  <set-param name="@input-classes@">
    cname(Message)||cname(MessageProfile)
  </set-param>
  <set-param name="@merged-class@">PostMessage</set-param>
</aspect>
```

Steps 2 and 3 can be also realized with the same approach. The following aspect describes a transformation step that adds an inheritance relation between the `ActionForm` class and an action form bean class. The `<relation>` is a tag for adding or deleting a relation such as an inheritance, an aggregation, or a relationship.

```
<aspect name="Generic-InheritActionForm" type="ordinary" jpm="RL">
  <params>
    <param>@sub-class@</param>
  </params>
  <pointcut name="super-sub-classes" type="class">
     cname(org.apache.struts.action.ActionForm)||cname(@sub-class@)
  </pointcut>
  <advice name="inherit-action-form" type="add-inheritance">
    <ref-pointcut>super-sub-classes</ref-pointcut>
    <advice-body>
      <relation>
        <end1>org.apache.struts.action.ActionForm</end1>
        <end2>@sub-class@</end2>
      </relation>
    </advice-body>
  </advice>
</aspect>
```

We can compose a set of related aspects within a component aspect. The following aspect, which generates an action form bean from PIM classes, composes aspects that describe Step 1 and 2.

```
<aspect name="Generic-Classes2ActionFormBean" type="component">
  <params>
    <param>@input-classes@</param>
```

```
    <param>@merged-class@</param>
  </params>
  <aspect name="MergeClasses" template="Generic-MergeClasses">
    <set-param name="@input-classes@">@input-classes@</set-param>
    <set-param name="@merged-class@">@merged-class@</set-param>
  </aspect>
  <aspect name="SetActionFormBeanName" template="Generic-SetActionFormBeanName">
    <set-param name="@class@">@merged-class@</set-param>
  </aspect>
  <aspect name="InheritActionForm" template="Generic-InheritActionForm">
    <set-param name="@sub-class@">concat(@merged-class@,"Form")</set-param>
  </aspect>
  <aspect name="AddAccessors" template="Generic-AddAccessors">
    <set-param name="@class@">concat(@merged-class@,"Form")</set-param>
  </aspect>
</aspect>
```

Four generic aspects are used for defining this component aspect. The definitions of the two generic aspects `Generic-SetActionFormBeanName` and `Generic-AddAccessors` are omitted here due to limitations of space. Sub-aspects are applied in the order of appearance when the `aspect-precedence` tags are omitted. The `concat` is a library function for concatenating two strings.

5.2 Extension of Model Transformations

Adopting AspectM, we can extend the functionality of the model compiler by adding aspect definitions. This extensibility is effective for defining application-specific model transformations. The following is the aspect that deletes the `date` attribute when the two classes `Message` and `MessageProfile` are merged.

```
<aspect name="DeleteAttribute" type="ordinary" jpm="OC">
  <pointcut name="postMessageClass" type="class">
    <pointcut-body>cname(PostMessage)</pointcut-body>
  </pointcut>
  <advice name="deleteDate" adviceType="delete-attribute"
        ref-pointcut="postMessageClass">
    <advice-body>date</advice-body>
  </advice>
</aspect>
```

This kind of aspect is useful for product-line engineering in which a variety of PSMs are generated from a single set of PIMs. A specific PSM, a model of a specific product, may have to be optimized in terms of memory resources. The above aspect, which eliminates the `date` attribute unused in a specific product, is applied after the `MergeClasses` aspect is applied. Using AspectM, a process of tuning up can be componentized as an aspect.

5.3 Descriptions for Other Crosscutting Concerns

AspectM can also describe the type of crosscutting concern that AspectJ supports. The following is an aspect for logging setter method calls. `Log.write()` is a log writer.

```
<aspect name="LoggingSetter" type="ordinary" jpm="PA">
  <pointcut name="allSetter" type="method">
    <pointcut-body>oname(set*)</pointcut-body>
  </pointcut>
  <advice name="logSetter" adviceType="before" ref-pointcut="allSetter">
    <advice-body>Log.write()</advice-body>
  </advice>
</aspect>
```

6 Discussion

It is not easy to quantify the effectiveness of AspectM because MDA based on aspect orientation is still young. In this section, we evaluate AspectM qualitatively based on our experience.

We can extend the functionality of the model compiler by adding aspect definitions. However, it is not realistic for a modeler to define all of the aspects needed to construct a model compiler from scratch. It is necessary for model transformation foundations, aspects commonly applied to many transformations, to be pre-defined by model compiler developers. For example, it is preferable to prepare aspect libraries that support de facto standard platforms such as J2EE and .NET. It is also useful to construct aspect libraries that support platform-independent model transformations commonly applied to many applications: fusion of classes having certain kinds of patterns, generation of setter/getter methods, change of naming conventions, and so on. Although aspect libraries are effective, it may be inconvenient to construct them by defining aspect diagrams because the number of aspect definitions tends to be large. It would be more convenient to use XML formats in the case of aspect library development. If a set of aspect libraries could be provided by model compiler developers, modelers would have only to define application-specific aspects as shown in Figure 6.

AspectM includes JPMs supported by major AOP languages. However, it is still not clear whether all kinds of model transformations can be described by the six JPMs. We think that there are situations for which new kinds of JPMs must be introduced. It would be better if a modeler can modify the AspectM metamodel using the model editor. This function can be considered as a modeling-level reflection, a kind of compile-time reflection.

In AspectM, we regard mechanisms explained by extended JPMs as aspect orientation. This definition might be slightly different from that of ordinary aspect orientation. If AspectM is not available, the users cannot describe such a transformation rule within UML since AspectM deals with meta concerns and UML deals with base-level concerns. In AspectJ, for example, a crosscutting concern can be described in Java but the resulting code will be tangled. This means that an aspect in AspectJ is not a meta concern. For this reason, it could be argued that AspectM is not an aspect-oriented language but a meta language for model transformations. However, AspectM can describe not only model transformation rules but also ordinary crosscutting concerns such as logging. AspectM unifies lightweight meta-programming with ordinary aspect orientation by extending the idea of JPMs. It is not necessarily easy to separate platform-specific

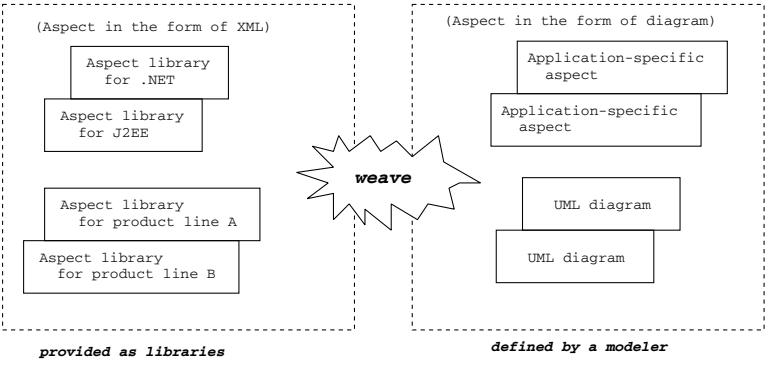


Fig. 6. Software development process using AspectM

concerns with only ordinary aspect orientation. In [18], not only AspectJ but also Java reflection is used for describing database concerns. The approach of AspectM can be considered reasonable.

7 Related Work

There has been research that has attempted to apply aspect-oriented mechanisms in the modeling phase. D.Stein et. al. proposed a method for describing aspects as UML diagrams[21]. In this work, an aspect at the modeling-level was translated into the corresponding aspect at the programming language level, for example an aspect in AspectJ. Y.Han et. al. proposed a meta model and modeling notation for AspectJ[11]. An aspect in AspectM is not mapped to an element of a specific programming language, but operates on UML diagrams. U.Aßmann and A.Ludwig claimed that aspect weaving could be represented as graph rewriting[1]. A UML diagram also can be regarded as graph. J.Sillito et. al. proposed the concept of usecase-level pointcuts, and showed the effectiveness of JPMs in early modeling phases[20]. E.Barra et. al. proposed an approach to an AOSD working method, using the new elements added in UML 2.0[3].

There is a standard model transformation language called QVT (Queries, Views, and Transformations)[17] in which model elements to be transformed are selected by query facilities based on OCL (Object Constraint Language)[23], and are converted using transformation descriptions. Since the purpose of QVT is to describe model transformations, QVT does not provide facilities for describing crosscutting concerns explicitly. AspectM can describe not only model transformation rules but also other kinds of crosscutting concerns.

Domain-specific aspect-oriented extensions are important. Early AOP research aimed at developing programming methodologies in which a system was composed of a set of aspects described by domain-specific AOP languages[12]. Domain-specific extensions are necessary not only at the programming stage but also at the modeling stage. J.Gray provided significant research on topics

including aspect orientation, model-driven developments, and domain-specific languages[8][9]. He proposed a technique of aspect-oriented domain modeling (AODM), and introduced a language called ECL (Embedded Constraint Language), an extension of OCL. ECL included the idea of QVT and provided facilities for adding model elements such as attributes and relations. Although the approach of AODM was similar to AspectM, the purpose of AODM was to realize domain-specific languages. He also proposed an approach that used a program transformation system as the underlying engine for weaver construction[10]. M.Shonle et. al. proposed an extensible domain-specific AOP language called XAspect that adopted plug-in mechanisms[19]. Adding a new plug-in module, we can use a new kind of aspect-oriented facility. CME (Concern Manipulation Environment)[4] adopted an approach similar to XAspect.

AspectM can be considered an XML-based AOP language. There are several AOP languages that can describe aspects in XML formats. AspectWerkz[2] is one such language. However, aspects in AspectWerkz are strongly related to an AspectJ-like JPM, and do not support multiple JPMs as in AspectM. Using AspectM, we can use multiple pieces of design information in describing modeling-level pointcuts. This is one of the advantages of applying aspect orientation to the modeling level. Another approach for enriching pointcuts is to adopt the functional XML query language XQuery[24]. M.Eichberg, M.Mezini, and K.Ostermann investigated the use of XQuery for specification of pointcuts[7].

Introducing AspectM, model transformation rules can be accumulated as reusable software components. This approach is similar to that of Draco[16] proposed by J. Neighbors in 1980s. In Draco, software development processes were considered as a series of transformations: requirements are transformed into analysis specifications; analysis specifications are transformed into design specifications; and design specifications are transformed into source code. These transformations were componentized in Draco. J. Neighbors claimed that software development processes could be automated by composing these transformation components. In AspectM, these components can be described by aspects.

8 Conclusion

We proposed a method for constructing an extensible model compiler based on aspect orientation. A modeler can extend model transformation rules by defining new aspects in the process of modeling. We believe that the idea of AspectM will provide a new research direction for model compiler construction.

References

1. Aßmann, U. and Ludwig, A.: Aspect Weaving as Graph Rewriting, In *Proceedings of Generative Component-based Software Engineering (GCSE)*, pp.24-36, 1999.
2. Aspectwerkz. http://aspectwerkz.codehaus.org/
3. Barra, E., Genova, G., and Llorens, J.: An approach to Aspect Modeling with UML 2.0, The 5th Aspect-Oriented Modeling Workshop, 2004.

4. Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD, http://www.research.ibm.com/cme/.
5. Czarnecki, K., and Eisenecker, U. W.: *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.
6. Early Aspects, http://early-aspects.net/.
7. Eichberg, M., Mezini, M., and Ostermann, K.: Pointcuts as Functional Queries, In *Proceedings of International Conference on Asian Symposium (APLAS 2004)*, pp.366-382, 2004.
8. Gray, J.: Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Vanderbilt University, 2002.
9. Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A, and Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling, In *Proceedings of International Conference on Generative Programming and Component Engineering (GPCE 2003)*, pp.151-168, 2003.
10. Gray, J. and Roychoudhury, S.: A Technique for Constructing Aspect Weavers Using a Program Transformation Engine, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pp.36-45, 2004.
11. Han, Y., Kniesel, G., and Cremers, A., B.: A Meta Model and Modeling Notation for AspectJ, The 5th Aspect-Oriented Modeling Workshop, 2004.
12. Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, In *Proceeding of European Conference on Object-Oriented Programming (ECOOP'97)*, pp.220-242, 1997.
13. Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.
14. Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, pp.2-28, 2003.
15. MDA, http://www.omg.org/mda/.
16. Neighbors, J.: The Draco Approach to Construction Software from Reusable Components, In *IEEE Transactions on Software Engineering*, vol.SE-10, no.5, pp.564-573, 1984.
17. QVT, http://qvtp.org/.
18. Rashid, A. and Chitchyan, R.: Persistence as an Aspect, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.120-129, 2003.
19. Shonle, M., Lieberherr, K., and Shah, A.: XAspects: An Extensible System for Domain-specific Aspect Languages, In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), Domain-Driven Development papers*, pp.28-37, 2003.
20. Sillito, J., Dutchyn, C., Eisenberg, A.D., and Volder, K.D.: Use Case Level Pointcuts, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2004)*, pp.244-266, 2004.
21. Stein, D., Hanenberg, S., and Unland, R.: A UML-based aspect-oriented design notation for AspectJ, In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pp.106-112, 2002.
22. Struts, http://struts.apache.org/.
23. Warmer, J., and Kleppe, A.: *The Object Constraint Language Second Edition — Getting Your Models Ready for MDA*, Addison Wesley, 2003.
24. XQuery, http://www.w3.org/TR/xquery/.

FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake

Department of Computer Science,
University of Magdeburg, Germany
{apel, leich, rosenmue, saake}@iti.cs.uni-magdeburg.de

Abstract. This paper presents FeatureC++, a novel language extension to C++ that supports Feature-Oriented Programming (FOP) and Aspect-Oriented Programming (AOP). Besides well-known concepts of FOP languages, FeatureC++ contributes several novel FOP language features, in particular multiple inheritance and templates for generic programming. Furthermore, FeatureC++ solves several problems regarding incremental software development by adopting AOP concepts. Starting our considerations on solving these problems, we give a summary of drawbacks and weaknesses of current FOP languages in expressing incremental refinements. Specifically, we outline five key problems and present three approaches to solve them: *Multi Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins* that adopt AOP concepts in different ways. We use FeatureC++ as a representative FOP language to explain these three approaches. Finally, we present a case study to clarify the benefits of FeatureC++ and its AOP extensions.

1 Introduction

Feature-Oriented Programming (FOP) [5] is an appropriate technique to implement program families and incremental designs [7,3,1,6]. It aims to cope with the increasing complexity, lacking reusability and customizability of nowadays software systems. Aspect-Oriented Programming (AOP) [16] is a related programming paradigm and has similar goals: It focuses mainly on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability [9,17]. However, it does not focus explicitly on incremental designs or program families.

One contribution of this article are our investigations in the symbiosis of FOP and AOP. Our aim is to combine the strengths of both approaches with regard to the implementation program families. Doing so, we firstly review well-known problems of FOP, in particular shortcomings in crosscutting modularity. We argue that certain features of AOP can help to solve these problems. Mainly, the ability to handle dynamic crosscutting and homogeneous crosscuts, as well as the growing acceptance, motivates us to choose AOP. We propose three ways

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 125–140, 2005.

to do this symbiosis (as we will explain): *Multi Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins*. These three approaches cope with the present problems of the FOP paradigm in different ways. They contribute several ideas in improving crosscutting modularity in the face of incremental software development and program families.

Current research in this direction focuses mainly on Java. *AspectJ*[1] and the *AHEAD Tool Suite (ATS)*[2] are prominent examples. Although used in a large fraction of applications like operating systems, realtime embedded systems, or databases C++ is rarely considered. Current solutions for C++ utilize templates [31], simple language extensions [29], or C preprocessor directives. These approaches are complicated, hard to understand, and not applicable to larger software systems. Thus motivated, this article presents FeatureC++[3], a language proposal for FOP in C++. Using FeatureC++, we explain the use and the benefits of the three AOP extensions integrated into a FOP language.

Besides basic concepts known from other FOP languages FeatureC++ further exploits useful concepts of C++, e.g. multiple inheritance or generic programming support. Moreover, it solves different problems of object-oriented languages in implementing incremental designs, namely (1) the constructor problem [30,13], which occurs when minimal extensions have to be unnecessarily initialized, (2) the extensibility problem [14], which is caused by the mixture of class extensions and variations, and (3) hidden overloaded methods in C++, which are hindering for step-wise refinements. Whereas these solutions are known from previous work, the consistent embedding into a C++-based FOP/AOP language is new. We perceive them as indispensable for successful FOP languages.

To underpin our language proposal we have implemented a first prototype, available at our web site. Using a case study, we illustrate how to use FeatureC++. The study reveals the advantages of FeatureC++ and its AOP extensions compared to common FOP approaches.

The remaining article is structured as follows: Section 2 gives necessary background information. In Section 3, we introduce the basic language concepts and features of FeatureC++. Section 4 reviews drawbacks and weaknesses of FOP and suggests three approaches to overcome them. In Section 5, we present a case study that explains the use of FeatureC++ and its advantages. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2 Background

Pioneer work on software modularity was made by Dijkstra [12] and Parnas [27]. Both proposed the principle of *separation of concerns* that suggests to separate each concern of a software system in a separate modular unit. Following this principle leads to maintainable, comprehensible software that can be easily reused, customized and extended.

[1] http://eclipse.org/aspectj/
[2] http://www.cs.utexas.edu/users/schwartz/Hello.html
[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

AOP was introduced by Kiczales et al. [16]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [16,11]. The idea behind AOP is to implement orthogonal features as *aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using *pointcuts* and *advices*, an aspect weaver brings aspects and components together. Pointcuts specify the *join points* of aspects and components, whereas advices define which code is applied to these points. AspectJ and *AspectC++*[4] are prominent AOP extensions to Java and C++.

FOP studies feature modularity in program families [5]. The idea of FOP is to build software (individual programs) by composing *features*. Features are basic building blocks that satisfy intuitive user-formulated requirements on the software system. Features refine other features incrementally. This *step-wise refinement* leads to a layered stack of features. Mixin Layers are one appropriate technique to implement features [31]. The basic idea is that features are often implemented by a collaboration of class fragments. A Mixin Layer is a static component encapsulating fragments of several different classes (Mixins) so that all fragments are composed consistently. Advantages are a high degree of modularity and an easy composition [31].

AHEAD is an architectural model for FOP and a basis for large-scale compositional programming [5]. *AHEAD* generalizes the concept of features and feature refinements. Features consist not only of code but of several types of artifacts, e.g., makefiles, UML-diagrams, documentation. The *AHEAD Tool Suite (ATS)* provides a tool chain for AHEAD and FOP based on Java. The included *Jak* language supports Java-based Mixin Layers.

3 FeatureC++ Language Overview

FeatureC++ is a C++ language extension to support FOP. The following paragraphs give an overview of the most important language concepts.

3.1 Introduction to Basic Concepts

To implement FeatureC++, we have adopted the basic concepts of the ATS: Features are implemented by Mixin Layers. A Mixin Layer consists of a set of collaborating Mixins (which implement class fragments). Figure 1 depicts a stack of three Mixin Layers $(1-3)$ in top down order. The Mixin Layers crosscut multiple classes $(A-C)$. The rounded boxes represent the Mixins. Mixins that belong to and constitute together a complete class are called refinement chain. Refinement chains are connected by vertical lines. Mixins that start a refinement chain are called *constants*, all others are called *refinements*. A Mixin A

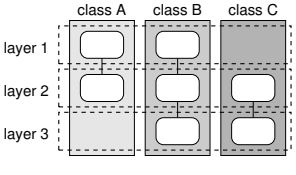


Fig. 1. Stack of Mixin Layers

[4] http://www.aspectc.org/

that is refined by Mixin B is called *parent* Mixin or parent class of Mixin B. Consequently, Mixin B is the *child* class or child Mixin of A. Similarly, we speak of parent and child Mixin Layers. In FeatureC++ Mixin Layers are represented by file system directories. Therefore, they have no programmatic representation. Those Mixins found inside the directories are assigned to be members of the enclosing Mixin Layers.

3.2 Basic Language Features

To reuse established language concepts and to increase the users acceptance, FeatureC++ adopts the syntax from the *Jak* language. The following paragraphs introduce the most important language concepts and features:

Constants and Refinements. Each constant and refinement is implemented as a Mixin inside exactly one source file. The root of a refinement chain is formed by these constants (see Fig. 2, Line 1). Refinements are applied to constants as

```
1  class Buffer {
2      char *buf;
3      void put(char *s) {/*...*/}
4  };
5  refines class Buffer {
6      int len;
7      int getLength() {/*...*/}
8      void put(char *s) {
9      if(strlen(s) + len < MAX_LEN)
10        super::put(s); }
11 };
```

Fig. 2. Constants and refinements

```
1  class Buffer {/*...*/};
2
3  // two buffer variations
4  class FileBuffer : Buffer {/*...*/};
5  class SockBuffer : Buffer {/*...*/};
6
7  // buffer extension: sync. support
8  refines class Buffer{ Lock lock; };
```

Fig. 3. Deriving variations vs. extensions

well as to other refinements. They are declared by the *refines* keyword (Line 5). Usually, they introduce new attributes (Line 6) and methods (Line 7) or extend[5] methods of their parent classes (see Fig. 2, Line 8). To access the extended method the *super* keyword is used (Line 10). *Super* refers to the type of the parent Mixin. It has a similar syntax to the Java *super* keyword and a similar meaning to the *proceed* keyword of *AspectJ* and *AspectC++*.

Solving the Extensibility Problem. FeatureC++ solves the *extensibility problem* [14]: implementation added to a class by creating a new subclass leaves the class' existing subclasses outdated.[6] It is caused by the divergence of variation and extension. Imagine an abstract buffer class with several subclasses, e.g., *FileBuffer*, *SockBuffer*. These classes are buffer variations. With common object-oriented languages the extensibility problem occurs: If one wants to extend the

[5] We do not use the term 'override' because we want to emphasize that usually method refinements reuse the parent method. This is more an extension than an overriding.

[6] The original definition regards the extension of designs by new operations and data types. However, following Cardone et al. [7] we deal with this problem in the context of class hierarchy extensions.

buffer class by subclassing, the existent buffer variations (the other subclasses) are not affected.

FeatureC++ solves the extensibility problem as follows: extensions are expressed as refinements whereas variations are derived using common inheritance. The variations *FileBuffer* and *SockBuffer*, depicted in Figure 3, inherit from the most specialized form of *Buffer* (in our example the synchronized buffer) regardless of their position and the position of the extension in the refinement chain. This facilitates the easy localized extension of (abstract) classes and the attended automatic extension of all variations.

Constructor Propagation. FeatureC++ solves the constructor problem [30,13]: in common object-oriented languages, e.g., Java and C++, constructors are not inherited automatically and have to be redefined for each subclass. The idea of FOP is to refine existing classes by many minimal extensions. In many cases these extensions do not need explicit new initializations. FeatureC++ solves the constructor problem by propagating all constructors of parent classes 'down' to their subclasses. That means, that all defined constructors of a refinement chain are available in the resulting generated class.

Besides constructors, also hidden overloaded methods are propagated down the refinement chain. Background is that C++ does not allow to access overloaded methods of a base class. These *hidden* methods are propagated too (see [2] for more details).

3.3 C++-Specific Language Features

The section so far has introduced features that are mostly adopted from Jak. The following language features are novel to FOP and exploit C++ capabilities. This makes FeatureC++ more powerful than current approaches, e.g. in supporting generic programming.

Multiple Inheritance. Multiple inheritance is a useful concept of object-oriented languages to express refinements. Figure 4 depicts a buffer refinement that adds synchronization and logging support using multiple inheritance. The corresponding functionality is implemented by inheriting from *Semaphore* and *Logging* and extending the buffer functionality.

```
1 refines Buffer : public
2         Semaphore,
3         Logging {/*...*/};
```

Fig. 4. Refining a buffer using multiple inheritance

```
1 refines template <class T> class Buffer {
2     void push(T &) {/*...*/}
3     T& pop() {/*...*/}
4 };
```

Fig. 5. Declaring a refinement as template

Generic Programming. To implement generic solutions, FeatureC++ supports generic programming, in particular class and method templates. Generic programming is essential to program families. The ability to parameterize refinements improves the variability in composing individually customized programs.

Figure 5 depicts a buffer refinement that uses a template parameter to determine the storage data type at instantiation time. Method templates are used analogously.

Further Language Features. C++ supports a lot of language features which are not available in Java. Currently, we support refinements of *destructors* and *structs*. Furthermore, we overload the keyword *this* to additionally providing access to the type of the enclosing Mixin. *this::Buffer* refers to the type of the current position in the refinement chain, instead of the type of the composed class.

4 Aspect-Oriented Extensions

FOP has several well-known problems in modularizing crosscutting concerns [24]. These problems degrade the modularity of program family members and decrease maintainability, evolvability, and customizability. We investigate solutions for the following selected problems, which are relevant for program family development: (1) weaknesses in expressing dynamic crosscutting, (2) inability to express homogeneous crosscutting concerns, (3) refinements have to be hierarchy-conform, (4) problem of method interface extensions, and (5) excessive method extensions.

We briefly review these problems (see [2,24] for a further discussion).

1. FOP has weaknesses in expressing dynamic crosscutting, which e.g. depends on the runtime control flow. FOP copes mainly with static crosscutting. Dynamic crosscutting is only supported in terms of intercepting and extending methods. AOP languages handle dynamic crosscutting in a more elegant and robust way. Novel innovative pointcut approaches (e.g. [26]) show the strength of AOP in this respect.
2. A second problem is that FOP languages deal only with heterogeneous crosscutting concerns, which apply different code at different positions. AOP, instead, copes mainly with homogeneous concerns that extend the base code at different join points with the same code fragments.
3. A third problem is that refinements to a given feature base must match the structure of this base, in particular, the class structure. A reorganization of the structure or the raising to a new abstraction level, as described in [24], is not possible.
4. A further problem occurs if method refinements need to extend the signature of the refined method, i.e. the argument list. This is only possible with an inelegant workaround.
5. A final problem are excessive method extensions in case of refinements that crosscut a large fraction of existing classes. For each method a crosscut depends on, the programmer has to introduce an extended method. This problem is caused by the inability of FOP to modularize homogeneous crosscutting concerns.

We perceive solutions to the listed problems as a benefit for implementing incremental designs and as an improvement of FeatureC++ against common FOP approaches. In the following, we present our investigations in solving these

problems using AOP language features as wildcards, pointcuts and advices. We present only preliminary approaches. A detailed analysis of the impact of these approaches on real-world applications, robustness, and code quality is part of future work.

Multi Mixins. Our first attempt was to tackle the problems of excessive method extensions and hierarchy-conformity. The idea is to allow Mixins to refine a whole set of parent Mixins instead of refining only one parent Mixin. Because of this refinement multiplicity we call these Mixins *Multi Mixins*.

```
1 refines class Buffer% {};
2
3 refines class Buffer {
4     void put%(...) {} };
```

Fig. 6. Two Multi Mixins

The sets of parent Mixins are specified by wildcards. Figure 6 shows two Multi Mixins that use wildcards to specify the Mixins and methods they refine. The unspecified sub-strings are denoted by '%'. The first Mixin refines all classes that start with "*Buffer*" (Line 1). The semantics of such *Class Multi Mixins* are straightforward: The term *Buffer%* has the same effect as if one creates a set of new refinements for each found Mixin that matches the pattern (*Buffer%*). The second Multi Mixin, called *Method Multi Mixin*, refines all methods of Buffer that start with "*put*" (Line 3). Similar to AOP languages, a join point API provides access to the arguments.

Both types of Multi Mixins ease the encapsulation of static homogeneous crosscuts by using wildcards to specify the set of target join points. Furthermore, Multi Mixins solve the problem of excessive method extensions by refining multiple methods using one extension. In this way also the hierarchical structure of the parent Mixin Layer is changed.

Aspectual Mixin Layers. The key idea behind *Aspectual Mixin Layers* is to embed aspects into Mixin Layers. Each Mixin Layer contains a set of Mixins and a set of aspects. Doing so, Mixins implement static, heterogeneous, and hierarchy-conform crosscutting, whereas aspects express dynamic, homogeneous, and non-hierarchy-conform crosscutting. In other words, Mixins refine other Mixins and depend, therefore, on the structure of the parent layer. These refinements follow the static structure of the parent features. Aspects refine a set of parent Mixins by intercepting method calls and executions as well as attribute accesses. Therefore, aspects are able to implement advanced dynamic crosscutting and homogeneous, non-hierarchy-conform refinements.

Figure 7 shows a stack of Mixin Layers that implements some buffer functionality, in particular, a basic buffer with iterator, a separated allocator, synchronization, and logging support. Whereas the first three features are implemented as common Mixin Layers, the *Logging* feature is implemented as an Aspectual Mixin Layer. The rationale behind this is that the logging aspect captures a whole set of methods that will be refined (dashed arrows). This refinement is not hierarchy-conform and depends on the runtime control flow (dynamic crosscutting). Moreover, the use of wildcards prevents the programmer from excessive method extensions.

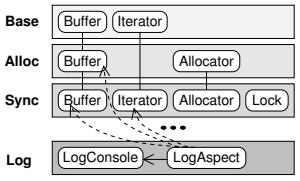


Fig. 7. Implementing a logging feature using Aspectual Mixin Layers

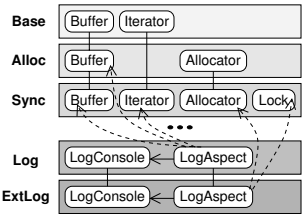


Fig. 8. Refining an Aspectual Mixin Layer

```
1 refines aspect LogAspect {
2     void print() { changeFormat(); super::print();  }
3     pointcut log() = call("%␣Buffer::put(...)") || super::log();
4 };
```

Fig. 9. An aspect embedded into a Mixin Layer

A further highlight of Aspectual Mixin Layers is that aspects can refine other aspects by using the *refines* keyword. To access the methods and attributes of the parent aspect, the refining aspect uses the *super* keyword. Figure 8 shows an Aspectual Mixin Layer that refines the logging aspect by additional join points to extend the set of intercepted methods. Beside this, the logging console (implemented as a Mixin) is refined by additional functionality, e.g. a modified output format. Generally, aspects can refine the methods of parents aspect as well as the parent pointcuts. Extending pointcuts increases the reuse of existing join point specifications (as in the logging example). Note that refining/extending aspects is conceptually different than applying aspects themselves. Whereas the latter case applies the aspects first, the former case results in a transformation of the aspect code before applying them to the target program.

To express aspects in Aspectual Mixin Layers we adopt the syntax of AspectC++. Figure 9 depicts an aspect refinement that extends a logging feature, including a logging aspect. It extends a parent method in order to adjust the output format (Line 2) and refines a parent pointcut to extend the set of target join points (Line 3). Both is done using the *super* keyword.

Aspectual Mixins. The idea of *Aspectual Mixins* is to apply AOP language concepts directly to Mixins. In this approach, Mixins refine other Mixins as with common FeatureC++, but they also define pointcuts and advices (see Fig. 10). In other words, Aspectual Mixins are similar to Aspectual Mixin Layers but integrate pointcuts and advices directly into Mixins.

In this sense, Aspectual Mixins are related to *Classpects* [28] that unify AOP and OOP language concepts. However, we see problems regarding this intermixing of Mixins (attributes, methods, refinements) and aspect elements (pointcuts, advices). Combining both may lead to a dependency of life time of the aspect and the Mixin subset. Usually, aspects are not instantiated directly by the user but triggered by the matching join points (as in AspectJ). Instead, Mixins are

```
1 refines class Buffer {
2     int length() {/*...*/}
3     pointcut log() = call("% Buffer::%(...)");
4 };
```

Fig. 10. Combining Mixins and AOP elements

instantiated by the user. Currently, we are not sure what the instantiation of an Aspectual Mixin results in: (1) The aspect subset is instantiated as well (as in *Caesar* [24]). (2) The aspect subset is instantiated only once (as in *AspectJ*). The problem of the former case is that often only one instance is needed. The latter case may lead to problems in accessing the internals of the Aspectual Mixin. For instance advices must not access instance attributes of the enclosing Aspectual Mixin. A deeper analysis of the consequences is important and part of future work.

4.1 Summary

All three approaches provide solutions for certain problems of FOP. They deal with the problems in different ways and contribute improved techniques for implementing incremental designs. Whereas Multi Mixins only solve the problem of hierarchy-conform refinements and method extensions, the Aspectual Mixins and Aspectual Mixin Layers can solve all stated problems. However, the Aspectual Mixin approach yields some problems regarding the instantiation and life time. Moreover, it is currently not clear if the mixture of aspect and Mixin subsets leads to deeper problems. Currently, Aspectual Mixin Layers are the only implemented variant (see [2]).

A further highlight of all three AOP extensions is a specific bounding mechanism that supports a robust incremental design. Originally it was proposed by Lopez-Herrejon and Batory [21]. They argue that with regard to program family evolution features should only affect features of prior development stages. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. This decreases aspect reuse and complicates incremental design. Consequently, our three extensions follow this principle. To achieve this bounding mechanism, the user-declared join point specifications must be restructured: Type names in wildcards are translated to match only the types of the current and the parent layers. Each wildcard expression that contains a type name is translated into a set of new expressions that refer to all type names of the parent classes. Figure 11 shows a synchronization aspect that is part of an Aspectual Mixin Layer. It has two parent layers (*Base*, *Log*) and several child layers. FeatureC++ transforms the aspect and the pointcut as depicted in Figure 12. This transformation works similar for Aspectual Mixins. In case of Multi Mixins we have to add a mechanism for combining wildcard expression logically. Unfortunately, we have no formal evidence that this transformation does not capture inadvertently classes of later development stages. This is part of future investigations.

```
1 aspect SyncAspect {
2   pointcut sync() :
3     call("%␣Buffer::put(...)");
4 };
```

Fig. 11. A simple pointcut expression

```
1 aspect SyncAspect_Sync {
2   pointcut sync() :
3     call("%␣Buffer_Sync::put(...)")
4     || call("%␣Buffer_Log::put(...)")
5     || call("%␣Buffer_Base::put(...)");
6 };
```

Fig. 12. Transformed pointcut

Finally, we want to emphasize that all three approaches are not specific to FeatureC++. All concepts can be applied to other FOP or AOP languages. We have implemented a first prototype of FeatureC++ including the support for Aspectual Mixin Layers. The implementation is described in [2] and a prototype is available at our web site.

5 A Case Study

This section introduces a case study that gives an overview of the functionality of FeatureC++. We choose the stock information broker example, adopted from [24], in order to point to the benefits of Aspectual Mixin Layers compared to common FOP approaches. We show how FeatureC++ overcomes the problems discussed in Section 4. The case study was implemented using our prototype.

Stock Information Broker. A stock information broker provides information about the stock market. The central abstraction is the *StockInformationBroker (SIB)* that allows to lookup for information of a set of stocks (see Fig. 13). A *Client* can pass a *StockInfoRequest (SIR)* to the *SIB* by calling the method *collectInfo*. The *SIR* contains the names of all requested stocks. Using the *SIR*, the *SIB* queries the *DBBroker* in order to retrieve the requested information. Then, the *SIB* returns a *StockInfo (SI)* object which contains the stock quotes to the client.

```
1  class StockInformationBroker {
2    DBBroker m_db;
3  public:
4    StockInfo &collectInfo(StockInfoRequest &req) {
5      string *stocks = req.getStocks();
6      StockInfo *info = new StockInfo();
7      for (unsigned int i = 0; i < req.num(); i++)
8        info->addQuote(stocks[i], m_db.get(stocks[i]));
9      return *info; }
10 };
11
12 class Client {
13   StockInformationBroker &m_broker;
14 public:
15   void run(string *stocks, unsigned int num) {
16     StockInfo &info = m_broker.collectInfo(StockInfoRequest(stocks, num));...}
17 };
```

Fig. 13. Stock Information Broker

All classes are encapsulated in a Mixin Layer. In other words, this Mixin Layer implements a basic stock information broker feature (*BasicSIB*). Figure 14 shows a relevant subset of this feature.

Pricing Feature as Mixin Layer. Now, we want to add a *Pricing* feature that charges the clients account depending on the received stock quotes. Figure 15 depicts this feature implemented using common FOP concepts. *Client* is refined by an account management (Lines 16-22), *SIR* is refined by a price calculation (Lines 2-5), and *SIB* charges the account when passing information to the client (Lines 10-12).

```
1  class StockInformationBroker {
2      DBBroker m_db;
3  public:
4      StockInfo &collectInfo(StockInfoRequest &req) {
5          string *stocks = req.getStocks();
6          StockInfo *info = new StockInfo();
7          for (unsigned int i = 0; i < req.num(); i++)
8              info->addQuote(stocks[i], m_db.get(stocks[i]));
9          return *info;   }
10 };
11
12 class Client {
13     StockInformationBroker &m_broker;
14 public:
15   void run(string *stocks, unsigned int num) {
16    StockInfo &info = m_broker.collectInfo(StockInfoRequest(stocks, num));...}
17 };
```

Fig. 14. The basic stock information broker (BasicSIB)

There are several problems to this approach: (1) The *Pricing* feature is expressed in terms of the structure of the *BasicSIB* feature. This problem is caused by the inability of FOP to express non-hierarchy-conform refinements. It would be better to describe the *Pricing* feature using abstractions as product and customer. (2) The interface of *collectInfo* was extended. Therefore, the *Client* must extend the method *run* in order to pass a reference of itself to the *SIB*. This is an inelegant workaround and increases the complexity. (3) The charging of clients cannot be dynamically altered, e.g. depending on the runtime control flow. Moreover, it is assigned to the *SIB* which is clearly not responsible for this function.

```
1  refines class StockInfoRequest {
2      float basicPrice();
3      float calculateTax();
4  public:
5      float price();
6  };
7
8  refines class StockInformationBroker {
9  public:
10     StockInfo &collectInfo(Client &c, StockInfoRequest &req) {
11         c.charge(req);
12         return super::collectInfo(req); }
13 };
14
15 refines class Client {
16     float m_balance;
17 public:
18     float balance();
19     void charge(StockInfoRequest &req);
20     void run(string *stocks, unsigned int num) {
21      StockInfo &info = super::m_broker.collectInfo(*this,
22           StockInfoRequest(stocks, num)); ... }
23 };
```

Fig. 15. The pricing feature using FOP (Pricing)

```
1  aspect Charging {
2      pointcut collect(Client &c, StockInfoRequest &req) =
3          call("% StockInformationBroker::collectInfo(StockInfoRequest &)")
4          && args(req) && that(c);
5      advice collect(c, req) : after(Client &c, StockInfoRequest &req)    {
6          c.charge(req); }
7  };
8
9  refines class StockInfoRequest {
10     float basicPrice();
11     float calculateTax();
12 public:
13     float price();
14 };
15
16 refines class Client {
17     float m_balance;
18 public:
19     float balance();
20     void charge(StockInfoRequest &req);
21 };
```

Fig. 16. The pricing feature using Aspectual Mixin Layers (Pricing)

Pricing Feature as Aspectual Mixin Layer. Figure 16 depicts the *Pricing* feature implemented by an Aspectual Mixin Layer. The key difference is the *Charging* aspect. It intercepts calls to the method *collectInfo* (Lines 2-4) and charges the calling client depending on its request (Lines 5-6). This solves the problem of the extended interface because the client is charged by the aspect instead by the SIB. An alternative is to pass the clients reference to the extended *collectInfo* method (not depicted). In both cases, the *Client* does not need to extend the *run* method.

A further advantage is that the charging of client's accounts can be made dependent to the control flow (using the *cflow* or *if* pointcut). This makes it possible to implement the charging function variable. Finally, our example shows that by using Aspectual Mixin Layers we have to refine only these classes that play the roles of product (*SIR*) and customer (*Client*).

Summary. Although the stock information broker example is very simple, it reveals the benefits of FeatureC++ and Aspectual Mixin Layers. FeatureC++ has all advantages of common FOP approaches. Furthermore, it is able to elegantly handle dynamic crosscutting, interface extensions, and non-hierarchy-conform refinements. Furthermore, Aspectual Mixin Layers can modularize homogeneous crosscuts and prevent excessive method extensions by using aspects (not shown). Due to the lack of space a description of a logging feature (homogeneous concern) that extends the broker application at multiple join points (preventing excessive method extensions) is omitted. The implementation is straightforward and was described many times. Table 1 summarizes the contribution of Aspectual Mixin Layers.

We readily admit that this simple case study cannot prove our ideas, and we do not intend to do so. This case study serves as proof of concept only and has

Table 1. Advantages of FeatureC++ Aspectual Mixin Layers

problem	solution	example
homogeneous crosscuts	pointcuts and advices	logging code is included in a set of methods
interface extensions	method interception, argument passing by aspects	the pricing aspect passes the clients reference to the SIB
hierarchy-conformity	refine only structure relevant Mixins; other are modified by aspects	refines *Client* as customer and *SIR* as product
dynamic crosscutting	use specific pointcuts (*cflow*, etc.)	charge clients depending on their runtime state
method extensions	wildcards in pointcut expressions	match all methods with price transfer

the aim to ease the understanding of our ideas. Mature case studies are supposed to flesh out our theses in future work.

6 Related Work

Work in several fields is related: programming support for incremental designs, AOP-related techniques, and the combination of AOP and FOP.

Programming support for incremental designs. One appropriate way to implement features of program families in a modular way are Mixin Layers [31]. Mixin Layers can be implemented using C++ templates [31], *P++* [29], *Jak* [5], *Java Layers* [8], *Jiazzi* [23], and *Delegation Layers* [25]. All these approaches leave aside the problem of lacking crosscutting modularity.

The constructor problem in incremental designs was introduced by Smaragdakis et al. [30]. Java Layers solve it by automatic constructor propagation from parent to child classes [8]. Eisenecker et al. utilize static C++ meta-programming [13]. Several approaches solve the extensibility problem, introduced by Findler et al. [14]: Java Layers [8], Jak [5], Jiazzi [23]. Regarding the constructor problem and extensibility problem, FeatureC++ is inspired by these approaches.

Aspects and separation of concerns. [24,19,20,21] discuss the drawbacks of current aspect-oriented languages, in particular no module boundaries, no feature cohesion, etc. FeatureC++ overcomes these problems by combining FOP and AOP concepts. This increases the crosscutting modularity and feature cohesion. Further, this preserves clear module boundaries and allows to scope aspect bindings.

Hyper/J supports multi-dimensional separation of concerns for Java [32]. This approach to software development is more general than that of FeatureC++ because it addresses the evolution of all software artifacts, e.g., documentation, makefiles, etc. However, Hyper/J has a lot of similarities to AHEAD [4]. Since FeatureC++ can be embedded into AHEAD it is an appropriate complement to Hyper/J.

The *Law of Demeter for Concerns (LoDC)* states that concerns should only know other concerns that contribute to its own functionality [18]. Following this principle (1) eases the incremental evolution of software by adding concern by concern and (2) minimizes the number of feature interactions. FeatureC++ follows LoDC and enables a clear encapsulation of concerns. The supported bounding mechanism scopes aspects in order to reduce unpredictable feature interactions.

Classpects combine capabilities of aspects and classes to unify the design of layered module systems [28]. They are related to Aspectual Mixins, whereas classpects unify advices and method bodies (advices can be explicitly invoked), but do not support mixin-based refinements.

AspectJ-like languages can express Mixins too. Using static introductions, several classes (and methods) can be refined. In the face of heterogeneous crosscuts, for each target class a new aspect must be introduced. Otherwise, one aspect declares all introductions. The problem of the first approach is that it does not support feature cohesion. Moreover, the target classes are defined at development time. Therefore, an easy exchange of the target layers is not possible (because class names change which is not the case with Mixins). The second approach merges multiple refinement chains into one aspect. This may destroy the logical structure. Furthermore, our Multi Mixins can be seamlessly integrated into Mixin Layers and support the FOP paradigm. Moreover, they support incremental development by a novel bounding mechanism (see Sec. 4.1).

Aspects, Features, and Collaborations. Mezini et al. show that using AOP as well as FOP standalone lacks crosscutting modularity [24]. They propose *CaesarJ* for Java as a combined approach. Similar to FeatureC++, CaesarJ supports dynamic crosscutting using pointcuts. In contrast to FeatureC++, CaesarJ focuses on aspect reuse and on-demand remodularization. Lieberherr et al. [19] introduce *Aspectual Collaborations* that encapsulate aspects into modules with expected and provided interfaces. The main focus is similar to CaesarJ.

Kendall explores the connection between role modeling and AOP [15]. However, she does not consider the embedding of aspects into collaborations. Furthermore, her approach has several drawbacks regarding cohesive role refinements.

Colyer et al. propose the *principle of dependency alignment*: a set of guidelines for structuring features in modules and aspects with regard to program families [10]. They distinguish between orthogonal and weak-orthogonal features/concerns.

Loughran et al. support the evolution of program families with *Framed Aspects* [22]. They combine the advantages of frames and AOP in order to serve unanticipated requirements. Frames are related to Aspectual Mixins and Aspectual Mixin Layers. Both allow to parameterize aspects at instantiation time.

7 Conclusion

This paper has presented FeatureC++, a novel FOP language extension to C++ that additionally adopts AOP concepts. Besides common FOP concepts it

supports several useful C++-specific extensions, e.g. multiple inheritance, generic programming. After an introduction to FeatureC++, this paper contributes a summary of several weaknesses of FOP in modularizing crosscutting concerns. We have explained how the weaknesses lead to problems in implementing incremental designs. Consequently, we propose three approaches to solve these problems, in particular, Multi Mixins, Aspectual Mixin Layers, and Aspectual Mixins. All these approaches adopt language concepts of AOP. A further highlight is a special bounding mechanism that supports a robust incremental development of program families. All three approaches are completely independent of FeatureC++ and can be applied to other FOP/AOP languages. Currently, we have implemented a prototype of FeatureC++ that supports most of the discussed language features, including Aspectual Mixin Layers. One can download a preliminary version of FeatureC++ at our web site[7]. Our case study has shown that FeatureC++ with its AOP extensions is able to elegantly express dynamic crosscutting, homogeneous crosscuts, non-hierarchy-conform refinements, and to cope with excessive method extensions and interface extensions.

In future work we want to investigate further in the relationship and symbiosis of FOP and AOP. In particular, we are interested in refining and evolving pointcuts and advices, as well as in different bounding mechanisms. Furthermore, we plan to implement and evaluate Multi Mixin and Aspectual Mixins. More complex case studies shall prove our results.

Acknowledgments

We would like to thank Don Batory and Erik Buchmann, as well as the program committee for comments on drafts of this paper.

References

1. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *ASE'04 SEM Workshop*, volume 3437 of *LNCS*. Springer, 2005.
2. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
3. D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Symposium on Software Reusability*, 1995.
4. D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. *ACM SIGSOFT*, 2003.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
6. D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.
7. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *AOSD*, 2002.
8. R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *ICSE*, 2001.

[7] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

9. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *AOSD*, 2004.
10. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.
11. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
13. U. W. Eisenecker, F. Blinn, and K. Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *Workshop on C++ Template Programming*, 2000.
14. R. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *ICFP*, 1998.
15. E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *OOPSLA*, 1999.
16. G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, 1997.
17. R. Laddad. *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning Publication Co., 2003.
18. K. Lieberherr. Controlling the Complexity of Software Designs. In *ICSE*, 2004.
19. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal (Special issue on AOP)*, 46(5), 2003.
20. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.
21. R. E. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *Software Engineering Properties and Languages for Aspect Technologies*, 2005.
22. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *AOSD ACP4IS Workshop*, 2004.
23. S. McDirmid and W. Hsieh. Aspect-Oriented Programming in Jiazzi. In *AOSD*, 2003.
24. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
25. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *ECOOP*, 2002.
26. K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *ECOOP*, 2005.
27. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE TSE*, SE-5(2), 1979.
28. H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *ICSE*, 2005.
29. V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. In *Workshop on Software Reuse*, 1993.
30. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *GCSE*, 2000.
31. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.
32. P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.

Shadow Programming: Reasoning About Programs Using Lexical Join Point Information

Pengcheng Wu and Karl Lieberherr

Northeastern University, Boston, USA
{wupc, lieber}@ccs.neu.edu

Abstract. The expressiveness of AspectJ's dynamic join point model has been shown in many useful applications, while the static join point model (also called *lexical shadows*) has been studied less. We propose a notion of shadow programming that exposes a program's adapted lexical shadow information to compile time language constructs to enable customized static analysis and more expressive join point selection mechanisms. In particular, within the framework of the AspectJ language and compiler, we have designed and implemented two compile time language constructs, called *Statically Executable Advice* and *Pointcut Evaluator* respectively, to show how the lexical shadow information can be used.

1 Introduction

Aspect-oriented programs consist of base programs and a list of aspects. Aspects are composed with base programs using a so called *join point model* [10]. In AspectJ's terminology, *join points* are well-defined points during a program execution, around which extra aspectual code(i.e.,*advice*) can be executed; *pointcut designators* pick out certain join points and expose values at those points. To compile aspect-oriented programs, aspect weavers (for example, the AspectJ compiler) are needed to parse the base programs and aspects, and to match lexical points in the program text against the pointcut designators. The matched lexical points are called *join point shadows* [16,7], and advice code will be injected into those places. At run time, the advice will be executed on *dynamic join points* that are run time instances of the lexical shadows.

In Aspect-Oriented Programming(AOP) [10,5] languages like AspectJ, *dynamic join point models* have been extensively studied and their expressiveness has been shown in many interesting applications, while much less has been done on the lexical shadow side. Shadows have only served as internal implementation constructs for AOP language compilers so far, however, it is also our view that a lot more can be exploited for other purposes as well. The examples include, but are not limited to, static program analysis and to support more expressive join point selection mechanisms.

Following this thought, this paper presents a notion of *shadow programming*, in which shadow information is exposed to aspect programmers who can take advantage of the information to reason about the program's static properties using

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 141–156, 2005.

supported compile time facilities. In particular, to show the feasibility and the usefulness of this notion, two compile time facilities, called *Statically Executable Advice* and *Pointcut Evaluator* respectively, are presented for programmers to write compile time program analyzers and to define sophisticated join point selection algorithms. We have implemented both facilities as extensions to the AspectJ language and the compiler. We believe more compile time facilities can be developed along this direction.

Outline. The rest of the paper is organized as follows. Section 2 discusses the motivations of this work. Section 3 introduces our lexical join point model adapted from AspectJ's internal lexical shadow structures. Section 4 presents the two shadow programming facilities and their use cases. Section 5 briefly describes the implementations. Section 6 compares this work with other related work and Section 7 concludes the paper.

2 Motivations

2.1 Compile Time Program Analyzers

To make programs more understandable, efficient or evolvable, it is important for programmers to follow some programming conventions. For example, an object's clients should never directly access the object's fields, if there are corresponding getters and setters available.

Unfortunately, most programming conventions are not checked by modern programming language compilers that usually just do regular type checking and code generation. Our observation is that the lexical shadow notion of aspect-oriented programming language compilers contains rich static information about program structure, which can serve as excellent ground for nontrivial programming convention checking. As a matter of fact, the AspectJ language has already supported very simple program property compile time checking. For example, to check that in any getter method, there should not be any operation to change an object's states, one can put the following declare error statement in an aspect:

```
aspect Foo {
  declare error : set(* *.*) && withincode(* *.get*(..))
                : "Side effect operations not allowed!";
}
```

When this aspect is compiled with a base program, the AspectJ compiler will try to match the program text against the pointcut designator expression **set(* *.*)&& withincode(* *.get*(..))**, and if any match is detected, the specified error message will be printed out along with the corresponding program text's lexical address (file name and line number), so that we know a violation has occurred. However, this feature is still very primitive in two senses: (1) the conventions it can characterize are limited to those that can be directly expressed using AspectJ's pointcut designators; (2) programmers have no direct access to the lexical shadow information that is important to implement more complex compile time checkers, as we will see later.

In fact, the aforementioned declare error statement fails to report the complete violation set. If a getter method does not directly update a field, instead, it calls another getter method that updates a field, it can only report the violation of the latter method, while treating the former one as a nonviolation. Note that this misbehavior is not due to a programming bug, but due to the fact that AspectJ's declare error mechanism is limited in its expressiveness. Exposing lexical shadow information for programmers to access is essential for realizations of such kind of compile time checkers.

In [12], motivated by another programming convention checker, the Law of Demeter checker, we proposed an extension to AspectJ, called Statically Executable Advice, which allows programmers to define more complex computation using lexical shadow information. That proposal can be realized only if lexical shadow information is accessible to programmers at compile time. Observing that, now we have refined the proposal and implemented it in the AspectJ compiler (version 1.1). In this paper, we discuss its applications in a broader context of reasoning about program properties using exposed shadow information.

2.2 More Expressive Join Point Selection Mechanism

Many aspect-oriented programming languages follow AspectJ's join point selection mechanism, called pointcut designators [10]. In this mechanism, there are static primitive pointcut designators such as call, execution, get, and set to match method or constructor call sites, method or constructor bodies, field read accesses and updates respectively. Programmers can use a simple pattern language to specify the signatures of those lexical points to be selected. Logical connectors || , & , and ! can be used to further refine selections. There are also *dynamic* pointcut designators including cflow,this,target, args and if. They refine join points selection at run time, and we will not cover them in this paper.

The expressiveness of the AspectJ's pointcut designator selection language is still very limited. One of the limitations is that it does not support join point selection based on particular properties of a lexical shadow other than those that can be directly expressed in the simple pattern language syntax. For example, one cannot select a call site call(* *.store()) (call to a method named *store*) such that the target type has a field named id, which is useful in enterprise applications to map an instance of a class back to a persistent store. As a matter of fact, asking whether AspectJ can support selection of join points based on special properties of classes/methods has been one of the most common questions in the AspectJ user community. Just as two additional examples from the AspectJ users mailing list, people were asking how to select the executions of a method in a class but not in its subclasses [8] and how to select the executions of a method in nonanonymous classes [15]. They are either very difficult or impossible to be expressed in AspectJ's pointcut designator language in its current form.

New pointcut selection primitives have been proposed to address this issue. For example, in another AOP system, JBoss AOP [9], a new pointcut expression hasField(..) has been proposed and implemented to help specify the aforemen-

tioned field scenario. But those extensions tend to be specific, and more general solutions are still needed.

Since the properties that a programmer wants to specify about program lexical points can be arbitrary, any general solution suggests programmer accessible lexical shadow information and a mechanism to reason about shadow information. Based on the programmer accessible shadow assumption, we propose a new pointcut designator expression called *Pointcut Evaluator* as a general approach to this problem and it has also been implemented in the framework of the AspectJ compiler (version 1.1).

3 Lexical Join Point Model

3.1 Lexical Shadows in AspectJ Compiler

We briefly introduce what lexical shadows are in the AspectJ language and compiler. According to the AspectJ language model, join points are points in the execution of a program. Join points are also called *dynamic join points*, because they exist only at run time. Each dynamic join point has its corresponding static part in the program text, which is called *lexical shadow* [16,7]. The AspectJ compiler operates on potential lexical shadows and matches those shadows against pointcut designators. For matched shadows, extra code will be injected to call advice at an appropriate time and to construct dynamic join point instances that are accessible to advice code at run time through a keyword variable thisJoinPoint.

Lexical shadows are abstractions of entities in a program, and they contain rich static information about those entities. As an example, a method call shadow contains the name of the method, the static types of the target object and the arguments, and in which method body(which is another shadow) the call is invoked. Currently, there are nine kinds of shadows in the AspectJ language and compiler [7]. The most common ones are method(or constructor) execution [1], method(or constructor) call, field get, and field set.

So far shadows have only served for the compiler's internal implementation's purpose, i.e., for matching programming entities against pointcut designators. Our goal is to make shadow information available to programmer accessible compile time facilities. The shadow class structure of the AspectJ compiler is for its internal use only and thus is unsuitable for programmers to access. We have designed and implemented a new structure based on the compiler's shadow structure information. We call the new structure *lexical join point structure*.

3.2 Lexical Join Point Structure

Fig. 1 is the UML diagram of our abstraction of lexical join points.

[1] The term *execution* may be a little confusing to those who are not very familiar with AspectJ, since it sounds like a run time thing. You can just view it as the body of a method or a constructor.

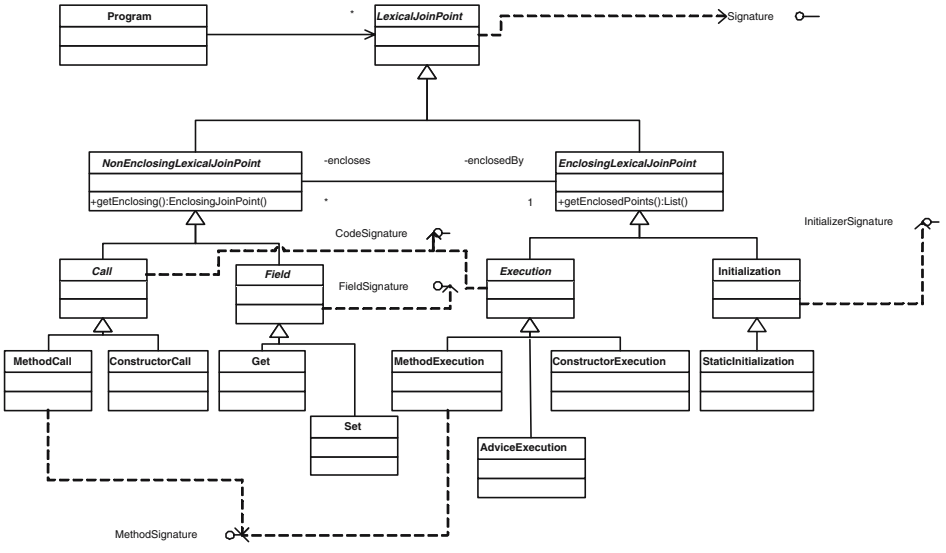


Fig. 1. Lexical Join Points

A program consists of a set of lexical join points, each of which fits into one of the two categories: NonEnclosingLexicalJoinPoint and EnclosingLexicalJoinPoint. An EnclosingLexicalJoinPoint may contain any number of NonEnclosingLexicalJoinPoint objects.

Lexical join points are further classified according to their kinds. The nine concrete classes[2] correspond to the nine kinds of shadows defined in the AspectJ language respectively. Some of the lexical join point classes implement signature interfaces (indicated as circled lines in the figure) so that programmers can access needed information through well-defined APIs. Those signature interfaces are all defined in AspectJ's public reflective API package org.aspectj.lang.reflect, which should be familiar to AspectJ programmers. Letting lexical join point classes implement specialized signature interfaces deliver better API accessibility to programmers, while in the current AspectJ language, programmers only have direct access to the general Signature interface and in programs, they often have to cast it down to more specialized interfaces according to the kind of the current join point.

The signature interfaces lend the programmers the capability to access type hierarchy information as well, which could come from one of two sources. The first source is Java's reflection system where class Class provides entry points to type information about a loaded class. This is a feasible source since AspectJ is using a byte code weaving approach and it is fair to assume that all the base

[2] Abstract class names are in the *italic* font, the circled lines represent interface types and the dashed lines represent implementation relationship between classes and interfaces.

program classes have been in the byte code form before weaving time. The other source is AspectJ compiler's internal type information abstracted as class TypeX, which collects all the class information in the program, either from source text or byte code. Class TypeX has almost the same APIs as Java's reflective Class and thus it is easy to unify them into a general one. To simplify our implementation, we use the reflection system as our source for type hierarchy information. For example, for any lexical join point object, we can ask in which *static* type this lexical join point is defined by calling method

```
Class getDeclaringType()
```

which is declared in interface Signature. In the next section, we will see examples how to make use of type information to reason about programs at compile time.

4 Shadow Programming Facilities

We show how we can take advantage of the exposed lexical join point information in two shadow programming facilities we have designed and implemented in the AspectJ compiler.

4.1 Statically Executable Advice

AspectJ's declare error construct is a useful static checking feature, but it is not expressive enough to check complex program properties. By extending AspectJ's declare mechanism, we can support user-defined complex static checking logic using available lexical join point information. This new construct is called *Statically Executable Advice* that can be declared in an aspect definition using the following syntax.

```
declare advice : pointcut expression : Identifier ;
```

The *pointcut expression* can be any legal AspectJ pointcut expression as long as no cflow, if, args, this and target is used in it[3]. We add this restriction because we want the pointcut expression to be statically resolvable, so that we can apply statically executable advice at compile time. The *Identifier* has to be the name of a user-defined class implementing interface IStaticallyExecutableAdvice as shown in Listing 1.

The idea is that for each of the declare advice declarations, the AspectJ compiler will load the class indicated by *Identifier* and create an instance from it. During the pointcut matching phase, if a shadow matches the specified pointcut expression, the compiler will create a lexical join point object corresponding to the shadow and call the corresponding method on the *Identifier* instance with the new lexical join point object as the argument. For example, if the current lexical shadow is a method call and it matches the pointcut expression, then the

[3] In AspectJ's terminology, a pointcut designator without cflow, if, args, this and target is called a *statically determinable pointcut*.

onCall method will be invoked on the *Identifier* instance with a MethodCall lexical join point created from the shadow as the argument. In addition, the start() method will be invoked before the whole compilation process begins and the finish() method will be invoked after the whole compilation process has finished.

Listing 1. IStaticallyExecutableAdvice.java

```
interface IStaticallyExecutableAdvice {
  void beforeExecution(Execution e);
  void afterExecution(Execution e);
  void beforeInitialization(Initialization ini);
  void afterInitialization(Initialization ini);
  void onCall(Call c);
  void onField(Field f);
  void start();
  void finish();
}
```

Please note that for each of the two direct subclasses of class EnclosingLexicalJoinPoint, namely Execution and Initialization, there are both before and after methods declared for it. But for each of the two direct subclasses of class NonEnclosingLexicalJoinPoint there is only an on method. The semantics is that if an enclosing lexical join point, say e, matches the pointcut expression, then its before method will be executed prior to all of the on methods for the matched nonenclosing lexical join points enclosed in e, and its after method will be executed after all of those on methods. This temporal order is important to simulate the lexical scopes of shadows.

One of the major advantages of using the declare advice construct is that we can make use of AspectJ's pointcut designator selection mechanism to specify where in the program we want to apply the checking logic. This expressiveness is similar to the traversal strategies and Visitor methods in the Demeter system [13] in that pointcut expressions play the roles of traversal strategies and statically executable advice methods play the roles of Visitor methods.

The following use cases exposes the usage and features of the statically executable advice.

Case 1: Side Effect Checking for Getters. As discussed in Section 2.1, AspectJ's declare error mechanism is not expressive enough to statically report whether a getter method may have side effects (change the states of some objects). We need to check the following properties:

1. If a getter method directly has field updating operations, then this method has side effect;
2. If a getter method calls some non-getter methods (whose names do not start with get"), we assume non-getter methods always have side effects and so is this getter method;

3. If a getter method calls other getter methods, then whether that getter method has side effect depends on those methods, and we need to record this dependency information for future processing;
4. After having processed all of the getter methods, then from the methods that have been marked as having side effects, we compute the transitive closure following the reversed dependency relationships and mark those methods as having side effect as well;
5. All of the unmarked getter methods are side effect free methods.

It is easy to see that even this simple program property cannot be easily checked. Fortunately, with our new statically executable advice construct, one can implement this static checker elegantly.

First, we need to specify what shadows we want to check and what is our class that implements the static checker. It is specified using our declare advice construct in an aspect as in Listing 2.

Listing 2. CheckerAspect.java

```
public aspect CheckerAspect {
  declare advice : withincode(* *.get*(..)) : EffectFreeChecker;
}
```

Basically, we need to check every lexical shadow occurring in the body of a getter method, which is specified as pointcut designator expression withincode(* *.get*(..)). Class EffectFreeChecker, which implements interface IStaticallyExecutableAdvice, is the class whose instance does the actual checking. A supporting class MethodNode maintains a global map that maps a getter method's signature to its corresponding MethodNode instance. Each MethodNode instance maintains the side effect dependency relationships of other MethodNode instances on itself. The supporting class MethodNode also maintains a global list of MethodNode instances that have been identified as having side effect.

With this supporting class, we now only need to implement three statically executable advice methods in class EffectFreeChecker (Listing 3).

When the compiler matches a field operation within the body of a getter method, the above onField method is invoked. If the operation happens to be a Set operation (determined by line 3, Listing 3), then that enclosing getter method is directly marked as with side effect.

On the other hand, as shown in the onCall method, if a method call shadow is matched in a getter method, we first make sure this call is also to a getter method (if not, we immediately mark the enclosing getter method "having side effect"), then we establish the side effect dependency relationships for future processing(lines 11 - 12, Listing 3).

Once all of the shadows in the program have been processed, the finish() method is invoked. The sole purpose of the finish method is to call a utility method reachTransitiveClosure() that starts from the nodes in the side effect method node list and marks every MethodNode instance reachable via the transitive closure of dependency relationships as having side effect. After this

Listing 3. EffectFreeChecker.java

```
class EffectFreeChecker implements IStaticallyExecutableAdvice {
  public void onField(Field f) {
    if(f instanceof Set)
      MethodNode.addSideEffectNode(f.getEnclosing().getSignature());
  }
  public void onCall(Call c) {
    if(!c.getName().startsWith("get")) {
      MethodNode.addSideEffectNode(c.getEnclosing().getSignature());
      return;
    }
    MethodNode theCall = MethodNode.getMethodNode(c.getSignature());
    theCall.addDependencyRel(c.getEnclosing().getSignature());
  }
  public void finish(){
    reachTransitiveClosure();
  }
  //other empty methods are skipped
}
```

process finishes, all the getter methods that have side effects will have been marked so.

When the base program and aspect CheckerAspect(Listing 2) are compiled using our extended AspectJ compiler, the getter method side effect checking will be executed automatically during the compilation process.

The implementation of this checker is very succinct due to two reasons: (1) the exposed lexical join point information adapted from shadows is available to the statically executable advice construct for free, while traditional approaches typically require a lot of parsing and abstract syntax tree traversals to get similar information; (2) as pointed out earlier, AspectJ's pointcut designator expression can declaratively instruct the compiler to only run the checker on relevant shadows, which otherwise would require a lot more code.

Case 2: Law of Demeter Checker. The Law of Demeter [14] (LoD) is another example of a programming convention we would like to check.

The class form of the LoD is a variant of the LoD. A class is less coupled with other classes if it follows the class form of the LoD. The essence is to restrict the classes whose methods can be invoked in a class's method. It can be summarized as: in a method of a class, we can only call methods on the following set of classes

- the class itself;
- classes of the class's fields;
- classes of the method's parameters;
- classes whose constructors are invoked in the method body.

It is clear that checking the class form of the LoD only requires statically available information. But we could not implement a sound checker in AspectJ, because using its dynamic join point model we could only do checking on a per-execution basis. With our new statically executable advice construct, we have easily implemented a sound LoD static checker.

Listing 4. LoDCheckerAspect.java

```
public aspect LoDCheckerAspect{
  declare advice : (execution(* *.*(..)) || call(* *.*(..)) || call(*.new(..)))
                   && withincode(* *.*(..))
                 : LoDChecker;
}
```

Listing 5. LoDChecker.java

```
1  public class LoDChecker implements IStaticallyExecutableAdvice {
     HashSet permissibleTypes = new HashSet();
3    List potentialViolations=new ArrayList();
     Class thisClass;
5    public void onCall(Call c) {
       if(c instanceof ConstructorCall) {
7        permissibleTypes.add(c.getDeclaringType());
         return;
9      }
       java.lang.reflect.Field[] fields = thisClass.getDeclaredFields();
11     for(int i=0; i<fields.length; i++) {
         if(c.getDeclaringType() == fields[i].getType())
13         return;
       }
15     if(permissibleTypes.contains(c.getDeclaringType()))
         return;
17     potentialViolations.add(c);
     }
19   public void beforeExecution(Execution e) {
       permissibleTypes.clear();
21     thisClass = e.getDeclaringType();
       permissibleTypes.add(thisClass);
23     Class[] paraTypes = e.getParameterTypes();
       for(int i=0; i<paraTypes.length; i++)
25       permissibleTypes.add(paraTypes[i]);
     }
27   public void afterExecution(Execution e) {
       Iterator it = potentiallyViolations.iterator();
29     while(it.hasNext()) {
         Call c =(Call)it.next();
31       if(!permissibleTypes.contains(c.getDeclaringType()))
           System.err.println("An LoD violation at: " + c.getSourceLocation());
33     }
       potentialViolations.clear();
35   } }
```

Listing 4 uses declare advice to specify where in the program the checks take place and which class implements the checking logic. We need to capture all method call sites residing in a method body to check their target types. We also need to capture constructor calls since they provide permissible types and we want to collect them. The shadows corresponding to execution(* *.*(..)) are the method bodies we are checking. Listing 5 is the implementation of the LoDChecker class that performs all the necessary checking for the class form of the LoD.

The instance of class LoDChecker maintains the permissible types (in a HashSet, Listing 5) in the context of a method body. Within a method body, if the compiler finds any constructor call shadow(Listing 5, lines 6 - 9), its type is added to the set of permissible types for the method; if a method call shadow is found instead, we first check whether the target type is one of the field types or already known permissible types, if not, then this call *may be* a violation call (Listing 5, lines 10 - 17). We cannot determine whether a method call site really violates the LoD until we have processed every shadow in the method body, since there may be other constructor calls coming after that method call, which will bring about more permissible types. This explains why we only report violations in the afterExecution method, right before the process leaves a method body being checked.

Summary. Our experiences with the two static checkers suggest that with the lexical join point information exposed at compile time, one can use statically executable advice to implement static checkers to check nontrivial program properties. Of course, the program properties that can be checked in this construct also depend on the expressiveness of AspectJ's pointcut designators. With more expressive designators added in, we can check even more interesting properties.

4.2 Pointcut Evaluator

As discussed earlier, in practice, there are many requirements for mechanisms to select join points based on properties of shadows, which usually cannot be expressed using the simple pattern matching syntax supported by AspectJ in its current form. The exposed lexical join points provide excellent grounds for more expressive join point selection mechanisms. One such mechanism is a compile time facility called *Pointcut Evaluator* which we propose and have implemented in the AspectJ compiler.

Again, using AspectJ's declare mechanism, in an aspect's definition, one can declare a pointcut evaluator variable which can be referred later in a pointcut designator definition. Here is the syntax.

```
declare evaluator : Identifier
                  : Identifier
                [ : Instantiation Option ] ;
```

The first *Identifier* is the name of an evaluator variable that will refer to an instance, while the second one is the name of a class, from which the instance

referred by the evaluator variable will be instantiated. The class has to implement a simple interface IPointcutEvaluator as shown in Listing 6. The optional *Instantiation Option* controls how the evaluator variable instance is instantiated, and currently, the option can be either perCompile (it is the default option, if no option is specified), or perPCD. Option perCompile indicates that there will be just a singleton instance associated with the evaluator variable during the whole compilation process; while option perPCD indicates that for each pointcut designator definition, there will be a separate instance associated with an evaluator variable used in the designator.

Listing 6. IPointcutEvaluator.java

```
public interface IPointcutEvaluator {
  public boolean eval(LexicalJoinPoint ljp);
}
```

A declared pointcut evaluator variable can be referred in a pointcut designator to refine the join point selection. An evaluator variable can appear anywhere in a pointcut designator where an AspectJ's standard primitive designator is expected. When the AspectJ compiler is matching a potential shadow against a pointcut designator, and if an evaluator variable is used in the designator, then the eval method will be called on the instance referred by the evaluator variable, with a LexicalJoinPoint object constructed from the shadow as the argument. Then the boolean return value from the eval method will be used together with logical connectors and the standard primitive designators to determine the final matching of the shadow. We will see how it is used in the following use case.

Use case: Contract checking for equals/hashCode. Modern software systems often rely on their components to obey some contracts [17] to ensure that systems behave correctly. Usually type systems cannot check contracts and thus they cannot be statically enforced. One such example is the contract between method equals(Object) and method hashCode() in Java's Object class. As documented in the Java API documents, under the entry of class Object, this contract is literally specified as the following statement:

– If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects **must** produce the same integer result.

The importance of this contract is evident from the fact that it has caught significant attentions from both professional programmers [1] and academic researchers [4]. To address this contract, in his well known book *Effective Java* [1], Joshua Bloch makes it a rule that *Always override hashCode when you override equals.*

The rationale of that rule is that if you have overridden the equals method without overriding the hashCode in a class, it is almost certain that the class will fail to obey the aforementioned contract. Following that rule (which is statically checkable) itself, however, will not guarantee that contract will be obeyed.

Instead, we cannot determine it until run time. So to determine whether a class obeys this contract, both compile-time and runtime checking is required.

It is impossible to implement this contract checking in the current AspectJ language, not just because it cannot provide the needed static checking support, but also because its pointcut designator mechanism is not expressive enough for what is needed for the dynamic checking. It is not easy to implement this in other software engineering tools either. But our pointcut evaluator construct provides an elegant solution to this task.

Listing 7 and Listing 8 are the implementation of this contract checker using the pointcut evaluator facility.

Listing 7. ContractCheckingAspect.java

```
aspect ContractCheckingAspect {
  declare evaluator: withHashCode : HashCodeChecker;
  pointcut p(Object b): execution(* *.equals(Object)) && args(b) && withHashCode;
  after(Object b) returning(boolean r): p(b) {
    if(r) { //equals returns true, then test the contract.
      if(thisJoinPoint.getThis().hashCode() != b.hashCode())
        System.err.println("Contract violation!");
    }
  }
}
```

Listing 8. HashCodeChecker.java

```
public class HashCodeChecker implements IPointcutEvaluator {
  public boolean eval(LexicalJoinPoint ljp) {
    //only returns true if the class overrides hashCode method
    Class thisType = ljp.getDeclaringType();
    Method[] methods = thisType.getDeclaredMethods();
    for(int i=0; i<methods.length; i++)
      if(methods[i].getName().equals("hashCode"))
        return true;

    //if it overrides equals(), must also override hashCode()
    if((ljp instanceof MethodExecution) && ljp.getName().equals("equals"))
        System.err.println("Must also override hashCode()!");
    return false;
  }
}
```

In aspect ContractCheckingAspect(Listing 7), a pointcut evaluator variable withHashCode is declared to refine the join point selection of pointcut designator p, which captures executions of equals method only if the target class also overrides the hashCode method. The static determination whether a class overrides hashCode actually is implemented by class HashCodeChecker(Listing 8), from which the instance referred by variable withHashCode is created. Class HashCodeChecker also checks the rule (a class that overrides equals must also

override hashCode) at compile time. The after advice in Listing 7 does the run time checking of the contract

5 Implementations

The implementations of the statically executable advice and pointcut evaluator facilities turned out to be seamless in the Eclipse AspectJ compiler (version 1.1), due to the compiler's extensibility.

There are two important concepts in the architecture of the AspectJ compiler, which are *Shadow* and *Shadow Munger* [7]. Shadow as an abstraction of the program text provides the grounding for our exposed lexical join point model, on which our two compile time facilities are based. Shadow munger is an abstraction of compile time actions when a shadow matches a pointcut designator expression. Two examples of shadow munger are *declare error processor* and *advice weaver*, which respectively prints out a specified error message and weaves the advice into the appropriate places when a shadow in the program is matched. The statically executable advice construct is just implemented as another shadow munger, called AdviceMethodLauncher. It constructs a lexical join point object from the matched shadow and uses it as the argument to call an IStaticallyExecutableAdvice method corresponding to the shadow on the instance created in the declare advice statement.

On the other hand, in a pointcut designator, a pointcut evaluator variable is treated the same as other standard primitive pointcuts, except that its semantics is to invoke the eval method on the instance referred by the evaluator variable with the lexical join point object corresponding to the current shadow as the argument. Its returning value is used to determine the matching of the shadow against the pointcut designator.

6 Related Work

Josh by Chiba and Nakagawa [2] is a new AOP language based on a compile time reflection library called Javassist. In Josh, programmers can have pointcut designators that are implemented as static Java methods using the Javassist API to access static information of the base program, just as we can refine the join point selections in pointcut evaluators using exposed lexical join point information. Our approach has a better integration with AspectJ's well accepted join point model, while Josh users have to program in two models, namely AspectJ-like join point model pointcuts and Javassist's compile time reflection model. Our exposed lexical join point information is obtained almost for free from the AspectJ compiler's internal shadow information, while Josh has to get similar information from a special compile time reflection library.

Eichberg, Mezini and Ostermann [3] use the XQuery language as a join point selection mechanism while the underlying shadow model is an XML representation of class information (translated from class files using a special tool). However, programmers should find our model more accessible, since in our model,

there is no need for special translation from Java programs to their XML representations.

There are also expressive pointcut languages [6,19,11,18] based on logic programming and its unification mechanism. These languages support join point selection on various data models, including the abstract syntax tree, the static type system, execution trace and heap objects. Arbitrary pointcut predicates can be written with regard to the data models to select join points. Aspects written in these pointcut languages tend to be less coupled to syntactic properties of base programs, and thus better aspect reusability can be achieved, just as the case in our shadow programming model. Due to logic programming's declarative nature and its built in unification mechanism, pointcut expressions in those languages are very concise.

SCoPE [20] is an extended AspectJ compiler that optimizes conditional pointcuts (if pointcuts) so that when if pointcuts only refer to statically available information, the SCoPE compiler can evaluate them at compile time and thus there is no runtime overhead associated with them, just like our Pointcut Evaluator facility. The expressiveness of these two features are comparable.

7 Conclusion

We observe that shadow information in AOP languages, particularly in AspectJ, can be also exploited for tasks other than compiler implementations, such as customized compile-time analysis and more expressive join point selection. The notion of shadow programming is proposed to expose the shadow information for usage by compile-time facilities. Concretely in the AspectJ language and compiler, we have designed and implemented the exposed shadow information and API as our lexical join point structure. Two shadow programming facilities, statically executable advice and pointcut evaluator, have been designed and implemented. Use cases for them have been presented exposing their usefulness and feasibility. We believe that more shadow programming facilities can be developed along this line and they will broaden the application domain of AOP languages.

Acknowledgements

We would like to thank Jeffrey Palm and Therapon Skotiniotis for their valuable comments on earlier drafts of this paper. We are also grateful to the anonymous reviewers of GPCE'2005 for their helpful comments.

References

1. Joshua Bloch. *Effective Java Programming Language Guide*. Sun Microsystems Inc., 2001. Pages 25-35.
2. Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111. ACM Press, 2004.

3. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*. Springer, LNCS, 2004.
4. Matthias Felleisen. Functional objects. In *Invited Talk at ECOOP*, 2004. Also available at http://www.ccs.neu.edu/home/matthias/Presentations/ecoop2004.pdf.
5. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis. http://ic-www.arc.n asa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf*, October 2000.
6. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented Software Development*, pages 60–69. ACM Press, 2003.
7. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.
8. Charles N. Harvey III, Jun 2004. http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg02460.html.
9. JBoss Inc. JBoss AOP. http://www.jboss.org.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–353, Budapest, 2001. Springer Verlag.
11. Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable Pattern Implementations Need Generic Aspects. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2004.
12. Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A Case for Statically Executable Advice: checking the Law of Demeter with AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49. ACM Press, 2003.
13. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
14. Karl J. Lieberherr and Ian Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.
15. Marius Marin, Sep 2004. http://dev.eclipse.org/mhonarc/lists/aspectj-users/-msg02944.html.
16. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, pages 46–60. LNCS, 2003.
17. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. Second Edition.
18. Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.
19. Tobias Rho and Günter Kniesel. LogicAJ - Uniform Genericity for Aspect Languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, December 2004.
20. Tomoyhuki Aotani and Hidehiko Masuhara. Compiling Conditonal Pointcuts for User-Level Semantic Pointcuts. In *Software engineering Properties of Languages and Aspect Technologies Workshop*, Chicago, IL, USA, Mar 2005.

Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax

Martin Bravenboer[1], Rob Vermaas[1], Jurgen Vinju[2], and Eelco Visser[1]

[1] Department of Information and Computing Sciences,
Universiteit Utrecht, P.O. Box 80089 3508 TB, Utrecht, The Netherlands
{martin,robv,visser}@cs.uu.nl
[2] Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, NL-1098 SJ, Amsterdam, The Netherlands
jurgen.vinju@cwi.nl

Abstract. In meta programming with concrete object syntax, object-level programs are composed from fragments written in concrete syntax. The use of small program fragments in such quotations and the use of meta-level expressions within these fragments (anti-quotation) often leads to ambiguities. This problem is usually solved through explicit disambiguation, resulting in considerable syntactic overhead. A few systems manage to reduce this overhead by using type information during parsing. Since this is hard to achieve with traditional parsing technology, these systems provide specific combinations of meta and object languages, and their implementations are difficult to reuse. In this paper, we generalize these approaches and present a *language independent* method for introducing concrete object syntax without explicit disambiguation. The method uses scannerless generalized-LR parsing to parse meta programs with embedded object-level fragments, which produces a forest of all possible parses. This forest is reduced to a tree by a disambiguating type checker for the meta language. To validate our method we have developed embeddings of several object languages in Java, including AspectJ and Java itself.

1 Introduction

Meta-level programs analyze, transform, and generate *object-level* programs. It is commonly agreed that such program manipulations are best carried out on a structured representation of the object program in order to achieve compositionality of transformations and to guarantee well-formedness of the resulting program. Furthermore, structured representations support type safety and hygiene more easily. However, the notation for structured representations is usually verbose and rather different from the notations of the language under consideration, rendering it impractical as a syntax for object programs. Using the concrete syntax of the object language as a notation for this structured representation provides the best of both worlds. The meta program can be written using the concise, well-known syntax of the object language, while the underlying representation is still structured.

Syntactically checked concrete object syntax is now available in many meta programming systems. Syntax macro systems such as <bigwig> [6], code generators such

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 157–172, 2005.

as Jak (JTS/AHEAD) [4] and Meta-AspectJ (MAJ) [22], and program transformation systems such as ASF+SDF [14], DMS [5], Stratego/XT [20] and TXL [12] all provide concrete object syntax. Some of these systems are designed for a specific object language, others are configurable for different object languages. In [20] we presented a general architecture for introducing concrete syntax for any object language in any meta language. The approach employs modular syntax definition in SDF and Scannerless Generalized-LR (SGLR) parsing for defining the syntax and parsing the combined meta and object language [19,10].

A remaining problem of concrete object syntax is that the syntax of the combined meta and object languages is usually highly ambiguous if the object language is embedded using a single pair of quotation and anti-quotation symbols. Most systems solve this by using a different quotation and anti-quotation symbol for each non-terminal of the object language, leading to considerable syntactic clutter and requiring the meta programmer to be intimately familiar with the syntactic structure of the object language. Because of the irregularity of the embedding, the set of syntactic categories that can be quoted and unquoted is usually limited. Moreover, in a language with manifest typing that already requires programmers to declare the types of all variables, the disambiguation of quotations feels redundant. For example, consider the following fragment written in Jak (part of the JTS/AHEAD Tool Suite [4]):

```
Stmt s = stm{ if($exp(exp)) {$stm(stm); }; }stm;
```

Here a statement `s` is constructed from an expression `exp` and a statement `stm`. The syntactic categories of the quotation of the entire fragment and the antiquotation of the variables within it are explicitly indicated using identifiers.

Meta-AspectJ (MAJ) [22], an extension of Java for the generation of AspectJ programs, reduces the need for different quotation and anti-quotation symbols by means of a context-sensitive parser, taking variable declarations into account during parsing. For example, in MAJ the Jak fragment above can be written as follows:

```
Stmt s = ‘[ if(#exp) { #stm } ]
```

The syntactic categories of the fragment and the variables are inferred from the explicit declaration of their types in the program. Thus, MAJ requires from the programmer less knowledge of the embedding and the syntactical details of the object language. However, the implementation of MAJ is specific to the embedding of AspectJ in Java, and is not easily reusable for embeddings of other languages, due to a number of limitations. First, the scanner for meta and object language is the same, which precludes embedding of languages with a different lexical syntax. Second, it is not possible to extend the meta language with concrete object syntax for *multiple* languages, since the implementations of context-sensitive parsing do not compose. Finally, the implementation of parsing and type checking is tangled, which leads to complex and hard to maintain code that has limitations that might surprise users. For example, MAJ cannot always handle overloaded methods that are invoked with quoted arguments.

In this paper, we describe an extension of our general architecture for concrete object syntax with type-based disambiguation that allows embeddings with minimal syntactic overhead. The main characteristic of our approach is that ambiguities are preserved by the parser and are solved in a separate phase by an extension of a type checker

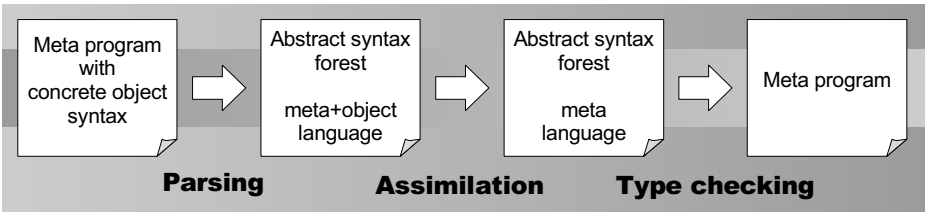


Fig. 1. Architecture of generalized type-based disambiguation

that operates on an abstract syntax forest. This separation of phases is illustrated in Figure 1. As a result, language embedding and assimilation (expansion of embedded object code to the meta language) can remain compositional. Therefore, it is easy to add new object languages and to combine object language embeddings. Since ambiguities are solved *after* assimilation, the implementation of disambiguation for a meta language is *object language independent*. However, we require that the representation of object programs in the meta language is typed and that distinct syntactical categories have a different type in this representation (see Section 4.4 and 5). Disambiguation is achieved by a natural and orthogonal extension of the type system. By separating the issue of disambiguation from the type checker, we can handle ambiguities in complex typing situations for free, hence reducing the number of exceptions and heuristics. Also, the approach is *not restricted to a single meta language*. Disambiguation is implemented as an extension of the type checker of the meta language, so the method is restricted to statically typed meta languages, and not applicable to untyped languages. Furthermore, the method is particularly suitable (and desirable) for languages that use manifest typing (e.g. C, Java, C#). We have no experience with meta languages using type inference.

We proceed as follows. In the next section we recapitulate the embedding and assimilation of an object language in a meta language. In Section 3 we examine the ambiguities caused by such embeddings and previous solutions used for them. In Section 4 we present a generalized type-based disambiguation method for concrete object syntax. In Section 5 we describe our experience with the method in a generic disambiguation implementation for Java as a meta language with embeddings of AspectJ and Java itself. In Section 6 we discuss previous, related, and future work.

2 Meta Programming with Concrete Object Syntax

In this section, we recapitulate the general method for adding support for concrete object syntax to a meta language, which was presented in [20,11]. Introduction of concrete object syntax in a meta language requires (1) embedding the syntax of the object language in the meta language and (2) assimilation of the embedded object code fragments to the meta language, expressed in terms of the underlying structured representation. The generality of the approach is based on syntax definition in the modular syntax definition formalism SDF for defining the embedding and the transformation language Stratego for the assimilation. We illustrate the approach with the introduction of concrete syntax for Java in Java.

```
   module JavaJava
   imports Java-15-Prefixed Java-15
   exports
     context-free syntax
[A]    "|[" Expr "]|"      -> MetaExpr {cons("ToMetaExpr")}
[B]    "#[" MetaExpr "]" -> Expr     {cons("FromMetaExpr")}
```

Fig. 2. Syntax definition for simple embedding of Java in Java

2.1 Embedding

The embedding of an object language in a meta language requires the combination of syntax definitions for both languages. From this combined syntax definition a parser is generated, which is used to parse meta programs that use concrete object syntax. Thus, the embedding of Java in Java is achieved by the module in Figure 2. The module imports the Java syntax twice; once as the meta language and once as the object language. To avoid confusion between the two languages (or language roles in this case), the nonterminals of the meta language are prefixed with '`Meta`' by renaming them in the import declaration.

Next, to actually integrate the meta and object language, the combination of these syntax definitions is extended with productions that determine the possible transitions from the meta language to the object language (quotation) and vice versa (antiquotation). A *quotation* quotes a fragment of an object-level program and embeds it in a meta-level program. Production `A` in Figure 2 defines that an object-level `Expr` between `|[` and `]|` can be used as a meta-level `MetaExpr`. The `cons` annotation in the production declares the constructor to be used in the abstract syntax tree. The following Java statement illustrates the quotation of a Java method call:

```
Expression x = |[ resultSet.getInt(4) ]|
```

The meaning of this statement is Java code for the construction of the abstract syntax tree corresponding to the quoted fragment, as will be further discussed below.

An *anti-quotation* is an escape from a quotation to the meta-level, to splice in pieces of object code computed elsewhere. Production `B` in Figure 2 declares that a `MetaExpr` between `#[` and `]` can be used as an object-level `Expr`. For example, in the following quotation the method argument is an expression *`foreignkey`* that is determined from some domain specification:

```
Expression x = |[ resultSet.getInt(#[ foreignkey ]) ]|;
```

2.2 Assimilation

Assimilation transforms a program with embedded object code to a pure meta-level program by translating the embedded fragments to code in the meta language that constructs the underlying abstract syntax tree representation. For example, in our Java in Java embedding we use the Eclipse JDT Core DOM [15] for representing the object programs. Hence, the Java constructs must be translated to invocations of the methods

in this API. The following Stratego rewrite rules illustrate the assimilation for some Java language constructs. The first rule translates a return statement, the second rule a method invocation. The Stratego rewrite rules use concrete object syntax as well.

```
Assimilate(rec) : |[ return; ]| -> |[ _ast.newReturnStatement() ]|

Assimilate(rec) : |[ e.y(e*) ]| -> |[
    {| MethodInvocation x = _ast.newMethodInvocation();
       x.setName(~e:<AssimilateId(rec)> y);
       x.setExpression(~e:<rec> e);
       bstm* | x |} ]|
  where <newname> "inv" => x
      ; <AssimilateArgs(rec | x)> e* => bstm*
```

In the assimilation rules we use a small extension {| *stmt** | *expr* |} of Java, called an eblock, that allows the inclusion of statements in expressions. The value of an eblock is the expression. In the assimilation rules, the *italic* identifiers (e.g. *e*, *y*, and *e**) indicate meta-level variables, a convention we use in all the code examples. `~e:` denotes an anti-quotation where the result is a Java expression. `<s> p` applies the rewriting *s* to the pattern *p*. *s* `=> p` matches the result of *s* to *p*. `newname` creates a fresh, unique, name, which guarantees hygiene in the assimilation. `AssimilateArgs` is a helper strategy that assimilates a list of expressions to arguments of the method invocation.

As an example, consider the result of assimilating the last example above, which illustrates the advantage of concrete syntax.

```
MethodInvocation inv = _ast.newMethodInvocation();
inv.setName(_ast.newSimpleName("getInt"));
inv.setExpression(_ast.newSimpleName("resultSet"));
List<Expression> args = inv.arguments();
args.add(foreignkey);
Expression x = inv;
```

In the examples of this paper, the assimilation is embedding specific, since the mapping of the object language to an existing API is inherently embedding specific. However, if there is a fixed correspondence between the syntax definition and the API, then the assimilation can be generic. This is typically the case if the API is generated from the syntax definition using an API generator such as ApiGen [8].

3 Ambiguity in Concrete Object Syntax

In this section we discuss how ambiguities can arise when using concrete object syntax. Also, we discuss how these ambiguities are handled in related work.

3.1 Causes of Ambiguity

Lexical State. If a separate lexical analysis phase is used to parse a meta program, then ambiguities will arise if the lexical syntax of the object language is different from the meta language. The set of tokens of both languages cannot just be combined, since

```
[A]  "|[" CompUnit  "]|" -> MetaExpr {cons("ToMetaExpr")}
[B]  "|[" TypeDec   "]|" -> MetaExpr {cons("ToMetaExpr")}
[C]  "|[" BlockStm  "]|" -> MetaExpr {cons("ToMetaExpr")}
[D]  "|[" BlockStm* "]|" -> MetaExpr {cons("ToMetaExpr")}

[E]  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
[F]  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}
```

Fig. 3. Syntax definition for embedding of Java in Java

the tokens of both languages are only allowed in certain contexts of the source file. For example, `pointcut` is a keyword in embedded AspectJ, but should not be in the surrounding Java code.

Quotation. Ambiguous quotations can occur if the same quotation symbols are used for different non-terminals of the object language. If the object code fragment in the quotation can be parsed with both non-terminals, then the quotation itself is ambiguous as well. For example, consider the SDF productions `A` and `B` in Figure 3 that define a quotation for a compilation unit and a type declaration. With these two quotation rules, the fragment `|[ class Foo { } ]|` is ambiguous, since the quoted Java fragment can be parsed as a compilation unit as well as a type declaration. Note that not all quotations are ambiguous: if the object code includes a package declaration or imports, then it cannot be parsed as a type declaration. A similar ambiguity issue occurs if the embedding allows quotation of lists of non-terminals as well as single non-terminals. For example, consider the SDF productions `C` and `D` in Figure 3 for quoting block statements. A quotation containing a single statement is now ambiguous, since it can be parsed using both production rules.

Anti-Quotation. Similar ambiguity problems occur when using the same anti-quotation symbols for different non-terminals of the object language. For example, consider the anti-quotations `E` and `F` in Figure 3 for identifiers and expressions. The anti-quotation in `|[ #[a] + 3 ]|` is ambiguous, since `#[a]` can represent an identifier as well as a complete expression.

3.2 Solutions

Lexical State. Most systems use a separate scanner. The consequence is that the lexical analysis must consider lexical states and will often assume fixed quotation symbols to determine the current state. Alternatively, the scanner can interact with the parser to support a more general determination of the lexical state. Some other systems just take the union of the lexical syntax, hence forbidding reserved keywords of the object language in the meta language. MAJ also reserves several keywords to work around lexical ambiguities (e.g. `pointcut` is a meta keyword) and some of these keywords are not even part of the object language (e.g. `VarDec` and `args`). ASF+SDF and Stratego both use *scannerless parsing* for parsing meta programs. Lexical ambiguities are not an issue in scannerless parsing, since they inherently only occur if a separate scanner is used.

Explicit Typing. Ambiguous quotations and anti-quotations can be solved by requiring explicit disambiguation by using different quotation symbols. For example, JTS uses different quotations for the class example: `prg{...}prg` for compilation units and `cls{...}cls` for class declarations. Stratego uses the same solution, but the disambiguated versions of the quotations are optional: if there is no ambiguity, then the general quotation symbols can be used. For example, `|[ package foo; class Foo {} ]|` is not ambiguous (it is a compilation unit), but a plain class declaration requires an explicit disambiguation, e.g. `comp-unit |[ class Foo {} ]|`.

JTS solves ambiguities between quotations of a single non-terminal and a list of non-terminals in two different ways. First, there are specific quotations for lists, for example `xlst{...}xlst` for the arguments of a method call. Second, some non-terminals only have a single quotation instead of two, where this single quotation always represents a list. In Stratego, list quotations are explicitly disambiguated, e.g. `bstm* |[ x = 5; ]|`.

Context-sensitive Parsing. MAJ uses context-sensitive parsing to solve ambiguous quotations and anti-quotations, by using type information at parse-time to infer the type of the quotation or anti-quotation to be parsed. Concerning list quotations, if `infer` is used, MAJ uses a single element if possible and an array if it must be a list. If the type of the variable is declared, then this type is considered. For example, the quotation in the statement `Stmt[] stms = ‘[ x = 4; ];` will be parsed to the construction of an array instead of a single statement. Hence, explicit disambiguation of the quotation itself is not necessary. Unfortunately, MAJ does not implement full support for the type system of Java and uses common interfaces for conceptually different AST classes to workaround issues in the quotation inference. Section 5 discusses these problems in more detail.

Grammar Specialization. ASF+SDF is a system with first order types. It translates this type system to a context-free grammar, thus parsers can be generated that accept only type correct meta programs. As a result, neither quoting of object fragments, nor anti-quoting of meta variables, nor explicit typing is necessary in ASF+SDF. However, the type system is limited to first order types only. Remaining ambiguities are currently solved by using heuristic disambiguation filters, such as injection count. In [18] a type-based solution for these ambiguities is presented, where grammar generation is no longer necessary.

4 Generalized Type-Based Disambiguation

In summary, quotation and anti-quotation can be used to introduce concrete syntax for object-level program fragments, but need some form of disambiguation. Explicit disambiguation methods introduce syntactic clutter that obscures meta programs. Reduction of this syntactic clutter can be achieved by using type information for disambiguation. While MAJ does a great job at achieving this for the specific embedding of AspectJ in Java, its implementation is hard to generalize to other object languages and to the combination of multiple object languages, because of the poor compositionality of its context-sensitive parsing algorithm.

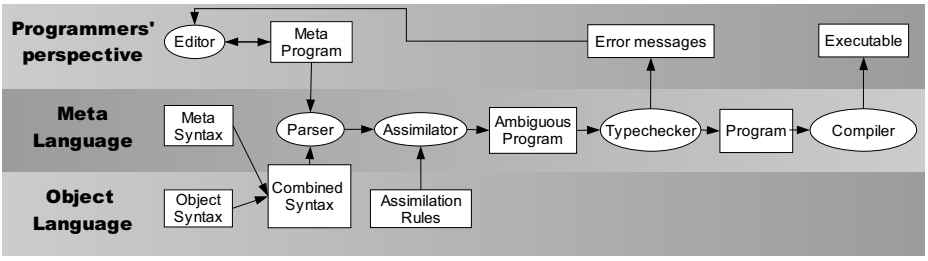


Fig. 4. Architecture of embedding and assimilation framework with type-based disambiguation

In this section, we introduce an alternative approach that generalizes easily to arbitrary object languages. Indeed it is generic in the embedded object language and can easily be transposed to other meta languages, considering the restrictions on the type system, as mentioned in the introduction. We illustrate the method with the embedding of Java in Java, but stress that the architecture and implementation is object language independent. The basic idea of the approach is to perform *type-based disambiguation of an abstract syntax* forest *after assimilation*. The architecture of our method is illustrated in Figure 4. In the rest of this section we describe the elements of the pipeline.

4.1 Syntax Definition and Parsing

The first stage of the pipeline consists of parsing the meta program with a parser generated from the combined syntax definition. This phase preserves all the ambiguities in the meta program, by employing generalized-LR parsing. The result is a parse *forest*, that is, a compact representation of all possible parses of the program. At points where multiple parses are possible the forest contains *ambiguity nodes* consisting of a set of all alternative parse trees, or in fact forests, since ambiguities can be nested. As a technical note, we actually consider *abstract syntax forests*, that is parse forests with irrelevant information such as whitespace, comments, and literals removed. For example, the Java assignment statement `dec = ⟦ class Foo {} ⟧;` is parsed to the following abstract syntax forest in term notation (where we have elided some details of the structure of class declarations having to do with modifiers and such):

```
Assign(ExprName(Id("dec")),
       1> ToMetaExpr(  CompUnit(... ClassDec(... Id("Foo")...) ...)  )
       2> ToMetaExpr(  ClassDec(... Id("Foo") ...)   )
       3> ToMetaExpr([ ClassDec(... Id("Foo") ...) ] ) )
```

In this forest it is clear that the right-hand side of the assignment is ambiguous and has three alternative parses. We use the notation `1>...n>` to indicate the alternatives of an ambiguity node. The three alternatives are a compilation unit containing a class declaration, a class declaration on its own, and a singleton list of a body declaration declaring an inner class (see Section 3.1 for a discussion of ambiguities caused by lists). The `ToMetaExpr` constructor represents a transition from the meta language to the object language (see Figure 3).

4.2 Assimilation

The second stage in the pipeline is assimilation, i.e., the translation of the embedded language fragments to their implementation in the meta language as described in Section 2.2. The only difference is that assimilation now transforms a forest instead of a tree. If the assimilation rules are compositional (i.e. the transformed fragments are small) there is no interference between regular assimilation rules and ambiguities, that is, ambiguities are preserved during assimilation. Thus, after assimilation, the abstract syntax forest only contains meta language constructs and ambiguity nodes. For example, the following code fragment shows the intermediate result after assimilation of the example above to the Eclipse JDT Core DOM (again some details have been elided).

```
1> {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
      TypeDeclaration class_0 = _ast.newTypeDeclaration();
      class_0.setName(_ast.newSimpleName("Foo"));
      ... | cu_0 |}
2> {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
      class_1.setName(_ast.newSimpleName("Foo"));
      ... |class_1 |}
3> {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
      decs_0.add( ... );
      ... | decs_0 |}
```

4.3 Type-Based Disambiguation

In the final stage of processing the meta program, ambiguities are resolved. The disambiguation operates on an abstract syntax forest of the meta language without any traces of the object language. Thus, the disambiguation phase does not have to be aware of quotations and anti-quotations, or of their contents. The disambiguation is implemented as an extension of a type checker for the meta language that analyses the abstract syntax forest and eliminates the alternatives that are not type correct. The algorithm for disambiguation is sketched in Figure 5. From within the type checker the `disambiguate` function is invoked for every node *`node`* in the abstract syntax forest after typing it.

The `disambiguate` function distinguishes three cases, which we discuss in reverse order. If the node *`node`* is not ambiguous it is just returned. If one of the sub-nodes of *`node`* is ambiguous, its alternatives are lifted to the current node by `lift-ambiguity`. Its definition states that if *`n`* is equal to some ambiguity node within a context `c[.]`, the context is distributed over the ambiguity (We give an example of distribution of an assignment shortly). Finally, if the node *`node`* is directly ambiguous or after lifting the ambiguities from its sub-nodes, the `resolve` function is used to resolve the ambiguity.

The `resolve` function takes an ambiguous node and removes from it all alternatives that are not type correct. This may result in an empty set of alternatives (`#`*`node'`* `== 0`), which indicates a type error, a singleton set (`#`*`node'`* `== 1`), which indicates that the ambiguity is solved, or a set with more than one alternative (`#`*`node'`* `> 1`). In the latter case, if the ambiguity involves a statement or declaration no more context information can be used to select the intended alternative, hence it is reported as an ambiguity error. Otherwise, in the case of an expression, the ambiguity is allowed to be lifted into its parent level, where it may be resolved due to context information.

```
disambiguate(node) =
  if node is ambiguous then
    return resolve(node)
  else if node has ambiguous child then
    return resolve(lift-ambiguity(node))
  else return node

resolve(node) =
  node' := remove from node all alternatives which are not type correct
  if #node' == 0 then report type error
  else if #node' == 1 then return node'
  else if #node' >  1 then
    if node' contains a meta statement or declaration then
      report ambiguity error
    else return node'

lift-ambiguity(node) =
   if node == c[1> node1 2> node2 ... j> nodej] then
     return 1> c[node1] 2> c[node2] ... j> c[nodej]
```

Fig. 5. Algorithm for type-based resolution of ambiguities

To illustrate the lifting and elimination of ambiguities, consider the ambiguity between the compilation unit, type declaration and list of body declarations in the assimilated example. If this ambiguous expression occurs in an assignment, i.e.

```
dec = 1> CompUnit 2> TypeDec 3> List<BodyDec>
```

then the ambiguity will be lifted out of the assignment (for brevity, the actual expression has been replaced by its type). This will result in a new ambiguity node with three alternatives for this assignment, i.e.

```
1> dec = CompUnit 2> dec = TypeDec 3> dec = List<BodyDec>
```

Depending on the type of the variable `dec`, two of these assignments will most likely be eliminated. For example, if the variable has type `TypeDec`, then the `CompUnit` and `List<BodyDec>` assignments will be eliminated, since these assignments cannot be typed. Note that this mechanism requires variables to be declared with a reasonably specific type. That is, if the variable `dec` has type `Object` then all the assignments can be typed and an ambiguity error will be reported.

Similarly, ambiguities are lifted out of method invocations: For example,

```
f(1> CompUnit 2> TypeDec 3> List<BodyDec>)
```

is lifted to

```
1> f(CompUnit) 2> f(TypeDec) 3> f(List<BodyDec>)
```

If `f` is just defined for one of these types, then just one of the invocations can be typed. Thus, the other invocations will be eliminated. On the other hand, if `f` is overloaded

or defined for a supertype of two or more of the types, then the ambiguity will be preserved. It might be eliminated later, if the result types of `f` are different. If this is not the case, then an ambiguity will be reported, similar to an ambiguous method invocation in plain Java.

4.4 Explicit Disambiguation

For cases that are inherently ambiguous or just unclear, explicit disambiguation can be used. Most systems introduce special symbols for this purpose, but due to our integration in the type checker one may use *casting* to ensure the type checker that something should a have certain type. The implementation of the explicit disambiguation comes for free, since incorrect casts cannot be type checked. Thus, these alternatives will be eliminated. For example, in our running example a cast to a compilation unit `(CompilationUnit) |[ public class Foo {} ]|` will cause the alternatives to be eliminated. In this way, any object language construct can be disambiguated, not only the ones that the developer of the embedding happens to support.

However, there is a situation where not even casting will help. Our method requires that the underlying structured representation of the object language is typed and that distinct syntactical categories in the object language have a different type in this representation. For example, if the structured representation is a universal data format such as XML or ATerms, then our method will not be able to disambiguate the concrete object syntax, since the different syntactical categories are not represented by different types in the meta language. Fortunately, a sufficiently typed representation is preferable anyway, since it would otherwise be possible to construct invalid abstract syntax trees. Note that similar problems occur in a dynamically typed languages. As mentioned in the introduction, our method is most suitable for statically typed languages.

5 Experience

To exercise the general applicability of our method to the embedding of different object languages, we implemented two large embeddings. Small fragments of the first application have already been presented in several examples: the embedding of Java in Java using assimilation to the Eclipse JDT Core DOM. We call this embedding JavaJava. The second application embeds AspectJ in Java and mimics the object language specific implementation of MAJ. Although AspectJ is a superset of Java, these two applications are quite different, since the embedding of AspectJ assimilates to the MAJ abstract syntax tree. The applications substantiate our claims, but also give some interesting insights in the limitations and the relation to object language specific implementations.

JavaJava. The implementation of JavaJava consists of a syntax definition (small fragment presented in Figure 3) and a set of assimilation rules that translate Java 5.0 abstract syntax tree constructs to the Eclipse JDT Core DOM [15] (examples have been shown in Section 2.2).

The mapping from the syntax definition to the Eclipse DOM is natural, since both are based on the Java Language Specification. Furthermore, the DOM is well-designed

and uses distinct classes to represent distinct syntactical categories. Because of this, our type-based disambiguation works quite well for JavaJava.

An interesting issue is the types of containers used in the DOM. The DOM uses unparameterized standard Java collections, as opposed to arrays, or type specific containers. So although the DOM itself can *represent* parameterized types in an object program, the DOM implementation itself does not *use* parameterized types. Our disambiguating type checker would benefit from parameterized collections, by harvesting the additional type information about the elements of a collection (e.g. `List<Expression>`). Fortunately, parameterized types and unparameterized types can be freely mixed, i.e. we can still use parameterized types in meta programs. However, we prefer a more precisely typed DOM, such that unchecked conversions or explicit casts can be avoided. Note that this shows that a sufficiently typed representation is important for our method.

Meta-AspectJ. We developed the embedding of AspectJ in Java to compare our generalized and staged disambiguation solution to a specific implementation, namely MAJ. For this, we also had to study the behaviour of MAJ in more detail. Our syntactical embedding is based on a modular AspectJ and Java syntax definition in SDF and exactly mimics the syntax of MAJ. The syntactical embedding was very easy to implement using SDF: basically we just have to combine the existing syntax definitions in a new module. The syntax definition also supports the explicit disambiguations of MAJ, but these are not really necessary, since casts can be used in our embedding method. For the underlying structured representation we use the MAJ AST.

We learned that our generalized implementation of disambiguation in a separate type checker has the advantage that it is much easier to implement support for more advanced Java constructs. For example, our implementation fully supports disambiguation of quotations in method invocations by performing complete method overload resolution, which MAJ does not. So, given the following overloaded method declarations:

```
Stmt     foo(CompilationUnit cu) { ... }
JavaExpr foo(ClassDec dec)       { ... }
```

our implementation can disambiguate invocations of the `foo` method that take quoted AspectJ code as an argument:

```
Stmt stmt     = foo(‘[ class MyClass {} ]);
JavaExpr expr = foo(‘[ class MyClass {} ]);
```

On the other hand, MAJ as an object language specific implementation provides some additional, object language specific, functionality that is not available in our generalized implementation. For instance, MAJ supports the conversion of Java (meta-level) values to AspectJ (object-level) expressions. For example, an array can be used as a variable initializer without converting it to the object-level by hand. Unfortunately, this conversion cannot be handled in a generic way, since it is not applicable to all object languages. However, for the specific embedding of Java in Java this could be added to the type checker (see future work).

We have not implemented the `infer` feature of MAJ, which supports inferring the type of a local variable declaration. Hence, the types of all variables should be declared in our implementation. The `infer` feature itself is not hard to implement, but we

would have to introduce heuristics to disambiguate ambiguous expressions, since no type is declared for the variable. MAJ applies such heuristics, for example by choosing a `ClassDec` if the type of the variable is `infer`, even if a `MajCompilationUnit` would also be possible. To work around incorrect choices, similar abstract syntax tree classes implement a common interface. For example, `ClassDec` and `MajCompilationUnit` implement the common interface `CompUnit`. This is a nice example of the problem mentioned in Section 4.4: distinct syntactical categories share a common interface. Thus, the declaration `CompUnit c = ‘[class Foo {}];` will result in an ambiguity error in our approach.

6 Discussion

Previous Work. We use the modular syntax definition formalism SDF [19], with integrated lexical and context-free syntax and declarative syntactical disambiguation constructs. It is implemented using scannerless generalized LR parsing [19,10]. SDF is developed in the context of the ASF+SDF Meta-Environment [14], but is used in several other projects such as ELAN [9]. Our Stratego/XT [21] program transformation system uses SDF for parsing, in particular of meta programs with concrete object syntax [20]. Stratego is not statically type checked. Therefore, it employs quoting with explicit typing, where necessary.

The ASF+SDF Meta-Environment is a meta programming system based on term rewriting. It uses grammar specialization to resolve ambiguities caused by object language fragments. To let the type system of ASF+SDF deal with parametric polymorphism, in [18] a separate disambiguating type checker replaces the grammar generation scheme. This solution instantiates the framework described in this paper for ASF+SDF.

Section 2 describes previous work on hosting arbitrary object languages in any host language [11], which generalizes the approach taken for Stratego [20] to any general purpose programming language. In the examples of [11], Java was used as the host language and we also embedded Java as the object language in Java. However, explicit disambiguation was required and an untyped underlying representation was used. The contribution of generalized type-based disambiguation, as presented in the current paper, is the introduction of a disambiguating type checker to remove the need for explicit typing. Moreover, the implementation is generic in the embedded object language. Thus, we obtain a similar notation as found in ASF+SDF, but can handle more than simple first order type systems, and use no disambiguation heuristics.

Related Work. The subject of embedding the syntax of object languages into host languages has a long history. The following discussion is meant to position our work more precisely. Early work on syntactic embeddings revolves around the concept of syntax macros [17]. They allow a user to dynamically extend a general purpose programming language with syntactic abstractions. These abstractions are defined in programs themselves. Implementations of this idea have been limited to certain subclasses of grammars, like LL(1) and LALR, as an argument of a fixed macro invocation syntax. Thus, these approaches can not be transferred to our setting of hosting arbitrary object languages.

The work of Aasa [1] in ML is strongly related to our setting. By merging the parsing and type checking phases for ML, and using a generalized parsing algorithm, this system can cope with arbitrary context-free object languages. It uses a fixed set of quotation and anti-quotation symbols that allow explicit typing to let the user disambiguate in case the type system can not decide. As opposed to this solution, our approach completely disentangles parsing from type checking, and allows user defined quotation and anti-quotation symbols.

DMS [5] and TXL [12] are specialized meta programming environments similar to ASF+SDF and Stratego/XT. In DMS the user can define AST patterns using concrete syntax, which are quoted and guarded by explicit type declarations. TXL has an intuitive syntax with keywords that limit the scope of object code fragments, instead of quoting symbols that surround every code fragment. Each code fragment, and each first occurrence of a meta variable is explicitly annotated by a type in TXL.

The Jakarta Tool Suite [4] and the Java Syntax Extender [2] are Java based solutions for meta programming and extensible syntax. Our framework, consisting of scannerless generalized-LR parsing and type-based disambiguation, is more general than the parsing techniques used by these systems. JTS uses explicit quotation and explicit typing, which can be avoided with our framework. Maya [3] uses extensible LALR for providing extensible syntax. Multi-dispatch is used to allow multiple implementations of new syntax, where the alternatives have access to the types of the arguments. Unfortunately, a separate scanner and LALR limit the syntactical flexibility. MAJ [22] obtains type-based disambiguation for the embedding AspectJ in Java, using context-sensitive parsing. We contribute by disentangling the parser from the type checker, resulting in an architecture that can handle any context-free object language. Our architecture stays closer to the original Java type system, in order to limit unexpected behavior.

Camlp4 [13] is a preprocessor for OCaml for the implementation of syntax extensions using an extensible top down recursive descent parser. New language constructs are translated to OCaml code by syntax expanders that are associated to the syntax extensions. Camlp4 provides quotations and anti-quotations to allow the generation of OCaml code using concrete syntax. The contents of quotations is passed as a string to a quotation expander, which can then process the string in arbitrary ways. A default quotation expander can be defined, but all other quotations have to be typed explictly. The same holds for syntactically ambiguous anti-quotations. As opposed to Maya, the syntax and quotation expanders can not use type information to decide what code to produce.

The method of disambiguation we use is an instance of a more general language design pattern called “disambiguation filters” [16]. Although there are lightweight methods for filtering ambiguities that are very close to the syntactic level [10], disambiguation filters can generally not be expressed using context-free parsing. For example, any parser for the C language will use an extra symbol table to disambiguate C programs. Either more computational power is merged in parsers, or separate disambiguation filters are implemented on sets of parse forests [7]. We prefer the latter approach, because it untangles parsing from abstract syntax tree processing.

The problem of disambiguating embedded object languages is different from disambiguation issues in type checkers, such as resolution of overloaded methods and

operators. First, disambiguation in type checkers can be done locally, based on the types of the arguments of the expression. Hence, lifting of the ambiguity, an essential part of our algorithm, is not used in such type checkers. Second, in our approach large fragments of the program can be ambiguous and are represented in completely different ways. For typical ambiguities in programming languages, such as overloaded operators, the alternatives can conveniently be expressed in a single tree.

Future Work. We are considering to widen the scope of the framework in two directions. Firstly, we would like to experiment with languages that have other kinds of type systems, such as languages with type inferencing and languages with union types. Secondly, the assimilation of an embedded domain-specific language (beyond object languages) often requires more complex transformations of the meta program and the object fragments. Applying type-based disambiguation after assimilation is a problem in this case. Extending the type checker of the host language with object language specific functionality is one of the options to investigate for this purpose.

7 Conclusion

We have extended an existing generic architecture for implementing concrete object syntax. The application of a disambiguating type checker, that is separate from a generalized parser, is key for providing single quotation and anti-quotation operators without explicit typing. This approach differs from other approaches due to this separation of concerns, which results in object language independence. It can still handle complex configurations such as Java embedded in Java. We have validated our design by means of two different realistic embeddings of object languages into Java, and comparing the results to existing systems for meta programming. The instances of our framework consist of meta programming languages that use manifest typing (i.e. Java), combined with object languages that have a well-typed meta representation (e.g., Eclipse JDT Core DOM). We explicitly do not provide heuristics to automate or infer types, such that the architecture's behavior remains fully declarative and is guaranteed to be compatible with the type system of the meta programming language.

Acknowledgements. At Universiteit Utrecht this research was supported by the NWO/JACQUARD project 638.001.201 TraCE: Transparent Configuration Environments, and at CWI by the Senter ICT-doorbraak project CALCE. We would like to thank Karl Trygve Kalleberg and Eelco Dolstra for providing detailed feedback.

References

1. A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 96–105. ACM Press, 1988.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 31–42. ACM Press, 2001.

3. J. Baker and W.C. Hsieh. Maya: multiple-dispatch syntax extension in java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281. ACM Press, 2002.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse (ICSR'98)*, pages 143–153. IEEE Computer Society, June 1998.
5. I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.
6. C. Brabrand and M.I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'02)*, pages 31–40. ACM Press, 2002.
7. M.G.J. van den Brand, S. Klusener, L. Moonen, and J.J. Vinju. Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In B. Bryant and J. Saraiva, editors, *LDTA'03*, volume 82 of *ENTCS*. Elsevier, 2003.
8. M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, May 2005.
9. M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In *Third International Workshop on Rewriting Logic and Applications*, ENTCS, 2000.
10. M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, April 2002.
11. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM Press, October 2004.
12. J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
13. Daniel de Rauglaudre. Camlp4 reference manual, INRIA, September 2003.
14. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
15. Eclipse Java Development Tools (JDT) website. `http://www.eclipse.org/jdt/`.
16. P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20. Tech. Rep. 126, Università di Milano, 1994.
17. B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.
18. J.J. Vinju. A type driven approach to concrete meta programming. Technical Report SEN-E0507, Centrum voor Wiskunde en Informatica, 2005.
19. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
20. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, October 2002.
21. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.
22. D. Zook, S.S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, volume 3286 of *LNCS*, pages 1–19. Springer, October 2004.

A Versatile Kernel for Multi-language AOP

Éric Tanter[1,⋆] and Jacques Noyé[2]

[1] Center for Web Research, DCC, University of Chile,
Avenida Blanco Encalada 2120, Santiago, Chile
[2] OBASCO project, École des Mines de Nantes – INRIA, LINA,
4, rue Alfred Kastler, Nantes, France
etanter@dcc.uchile.cl, noye@emn.fr

Abstract. Being able to define and use different aspect languages, including domain-specific aspect languages, to cleanly modularize concerns of a software system represents a valuable perspective. However, combining existing tools leads to unpredictable results, and proposals for experimentation with and integration of aspect languages mostly fail to deal with composition satisfactorily and to provide convenient abstractions to implement new aspect languages. This paper exposes the architecture of a *versatile AOP kernel* and its Java implementation, Reflex. On top of basic facilities for behavioral and structural transformation, Reflex provides composition handling, including detection of interactions, and language support via a lightweight plugin architecture. We present these facilities and illustrate composition of aspects written in different aspect languages.

1 Introduction

The existing variety of toolkits and proposals for Aspect-Oriented Programming (AOP) [13] illustrate the fact that the design space of AOP is still under exploration. Low-level toolkits (*e.g.* [8]) can be used to explore the design space and create specific AOP systems, but they require redeveloping an ad hoc software layer to bridge the gap with a proper high-level interface, and they do not address the issue of aspect language design. In this respect, there are proposals of both general-purpose and domain-specific aspect languages. Domain specificity presents many benefits: declarative representation, simpler analysis and reasoning, domain-level error checking, and optimizations [10]. Several domain-specific aspect languages were indeed proposed in the "early" ages of AOP [14, 18, 20], and, after a focus on general-purpose aspect languages, the interest in domain-specific aspect languages has been revived [1, 5, 21, 26].

When several aspects are handled in the same piece of software, it is attractive to be able to *combine* several AO approaches, for instance various domain-specific aspect languages [22]. Yet, combining AO approaches is hardly feasible with today's tools, since the tools are not meant to be compatible with each other: each

⋆ É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 173–188, 2005.

tool eventually affects the base code directly. This tends to jeopardize correctness when different aspects implemented with different tools interact.

Since most approaches rely upon common implementation techniques, we propose to provide a *versatile AOP kernel*, which supports core semantics, through proper structural and behavioral models. Designers of aspect languages can thus experiment more comfortably and rapidly with an AOP kernel as a back-end, focusing on the best ways for programmers to express aspects, may they be domain specific or generic. The crucial role of such a kernel is that of a mediator between different coexisting approaches: *detecting* interactions between aspects and providing expressive means for their *resolution*.

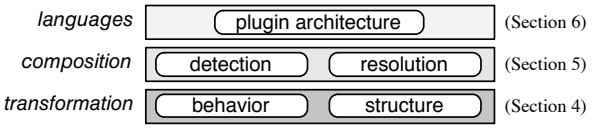


Fig. 1. Architecture of an AOP kernel

This paper illustrates the evolution of Reflex, originally a system for partial behavioral reflection in Java [25], into an AOP kernel. This experiment gives a first picture of what an AOP kernel may look like and of its benefits. Instead of focusing the discussion on a specific closed proposal, it raises, in a practical manner, the issue of determining what the building blocks of AOP are and how they can be combined in a flexible and manageable way. The proposed architecture of an AOP kernel consists of three layers (Fig. 1): a transformation layer in charge of basic weaving, supporting both structural and behavioral modifications of the base program; a composition layer, for detection and resolution of interactions; a language layer, for modular definition of aspect languages.

The following section overviews related work, further highlighting the motivation of this work. Section 3 exposes the running example of this paper. We then present the different layers of our AOP kernel following Fig. 1. Section 4 illustrates the core of Reflex as a reflective Java extension, explaining how aspects are mapped to this transformation layer. Section 5 discusses support for aspect composition. Section 6 describes a plugin architecture for modular aspect language support, explaining how plugins are used to bridge the gap between an aspect language and the core reflective infrastructure. Section 7 concludes.

2 Related Work

We now review several proposals related either to multi-language AOP or to extensible aspect languages.

XAspects [22] is a plugin mechanism for domain-specific aspect languages, based on AspectJ [15]. An aspect language is implemented as a plugin generating AspectJ code, while the global compilation process is managed by the XAspects

compiler. XAspects suffers a number of limitations: the compilation process is particularly heavyweight as it requires two full run of the AspectJ compiler; it is unclear whether controlling the visibility of structural changes made to base code is at all feasible; detection and resolution of aspect interactions is not tackled. More importantly, the XAspects compiler provides plugins with the plain binary representation of a program: no higher-level intermediate abstractions are made available to implementors of aspect languages.

Furthermore, although using AspectJ as both the transformation and composition layer of an AOP kernel (Fig. 1) is interesting because of the direct support for expressing crosscutting abstractions, there are several reasons that limit the validity of AspectJ as an AOP kernel. AspectJ is a mature, production-quality aspect language whose practitioner perspective results in limited versatility. In particular, AspectJ is poorly expressive with respect to aspect composition, as will be discussed later, and does not address *detection* of aspect interactions. We rather concur with Douence *et al.* that automatic detection of aspect interactions should be provided [11].

Brichau *et al.* [4] present an approach to building composable aspect-specific languages with logic metaprogramming. Aspect-specific languages are uniformly defined and composed using the same Prolog-like base language: an aspect language is implemented as a set of logic rules in a logic module. This approach provides means not only to compose aspects written in different aspect languages, but also to actually compose languages themselves. A drawback of this approach is that aspect languages do not really shield the programmer from the inherent power of the logic metaprogramming approach: no aspect-specific syntax is provided, aspects are defined in the same logic framework as languages. Finally, this work does not address detection of aspect interactions.

The Concern Manipulation Environment (CME) developed at IBM [9] is a large-scale project aiming to support aspect-oriented software development at any level (analysis, design, implementation, etc.), with respect to any computing environment (programs in various languages, UML diagrams, etc.). The motivation for developing a flexible infrastructure with advanced building blocks to experiment with various AOSD approaches is definitely shared with our work. However, the wide variety of target formats has a serious impact on the concern assembly language: assembly directives are usually specified open-endedly as strings. We rather aim at a higher-level conceptual model to reason about transformation. Finally, detection of aspect interactions is not considered.

Josh [7] is an open AspectJ-like language, which makes it possible to experiment with new means of describing pointcuts and advices. Due to its inlining-based implementation of aspect advices in base code, Josh lacks convenient support for stateful aspects and per-instance aspects. Also, issues related to aspect composition are not addressed.

Finally, *abc*, the AspectBench Compiler is an extensible framework for experimenting with new language features in AspectJ [2]. The spirit of *abc* is similar to Josh, but since *abc* is a compiler, not a load-time tool, it provides a powerful framework for static analysis. By sticking to AspectJ as the basic language, *abc*

presents the inconvenience that both the complexity of AspectJ and that of the *abc* infrastructure (basically a full compiler infrastructure) may be an overkill for simple extensions. As of today, the proposal does not explicitly address the possibility of mixing *different* aspect languages, and aspect composition is still limited to what AspectJ supports.

This work aims to address the limitations highlighted above: there is a need for a versatile kernel for multi-language AOP providing high-level abstractions to implement new aspect languages, and supporting both detection and resolution of interactions between aspects written in different languages.

3 Running Example

The running example of this paper is a multi-threaded program manipulating a buffer, to which three aspects expressed in different languages are applied. The `Buffer` class defines the `put` and `get` methods of an unsynchronized buffer.

First, the buffer is made thread safe using SOM (Sequential Object Monitors) [5], which makes it possible to code separately the scheduling strategy of the buffer. This strategy is implemented in the `BufferScheduler` (discussed in [5]). A small domain-specific aspect language (DSAL) is used to specify that a buffer instance should be scheduled by an instance of this scheduler, as follows:
`schedule: Buffer with: BufferScheduler;`

The second aspect is implemented using a general-purpose aspect language, AspectJ [15]. It implements an *argument checking* policy: validation behavior can be attached to join point arguments, either producing exceptions in case of invalid arguments, or simply skipping the invalid call. In our example, we use an argument checker aspect for the buffer, `BufferArgChecker`, which skips invocations of `put` with a null parameter.

The last aspect is used to attach a unique identifier (UID) to objects. This basically consists in adding a private field to hold the identifier, properly initialized, as well as a getter method and the associated interface. It is implemented directly with Reflex. This aspect is provided as a library, with a simple entry point for configuration. To apply it to the buffer, a configuration class[1] is used:

```
public class BufferUIDConfig {
  public static void initReflex(){
    UID.applyTo("Buffer");
} }
```

Now the question is: what happens when all three aspects are applied to the same base program, all affecting the same `Buffer` class? Are calls to the `getUID` method synchronized via SOM, although this is not necessary since it is inherently thread safe? Are rejected calls synchronized as well, or can we make sure that only accepted calls to the buffer are synchronized and scheduled?

[1] Configuration classes are the basic mechanism provided to configure Reflex at start up: their `initReflex` methods are called prior to the execution of the application.

Proposal. Our proposal consists in using an AOP kernel on top of which the different aspect languages are implemented, and having this system report on interactions and offer expressive means for the specification of their resolution. With both SOM and AspectJ available on top of Reflex, applying the three aspects above is done as follows:

```
java reflex.Run -som buffer.som -aspectj BufferArgChecker.aj
                -configClass BufferUIDConfig Main
```

When loading the class `Buffer`, Reflex *detects* the interactions and issues warnings, such as:

```
[WARNING] don't know how to compose SOM and BufferArgChecker.
[WARNING] composing arbitrarily (sequence).
```

The programmer is informed of the unspecified interaction between SOM and ArgChecker. The desired semantics here is to avoid scheduling a request if it is to be rejected (this is correct since validating arguments is thread safe). This can be specified by declaring a *composition rule* stating a *nesting* relation between the two aspects. This declaration can be done in a configuration class:

```
public class CompConfig {
  public static void initReflex(){
    API.rules().add(new Wrap("BufferArgChecker", "SOM"));
} }
```

As we will see in Sect. 5, `Wrap` is a *composition operator* that has the same semantics as precedence in AspectJ. The wrapped aspect (SOM) is only invoked if `proceed` is invoked by the wrapper aspect (BufferArgChecker). If BufferArgChecker rejects a call, it returns without calling `proceed`; hence SOM does not apply, meaning that a reification of the call as a request put in a pending queue until scheduled is avoided. Running Reflex with this composition specification is done by adding `CompConfig` to the list of configuration classes.

4 Overview of Reflex

The analysis of AOP features that led us to the proposal of AOP kernels [24] is concerned with asymmetric approaches to AOP, whereby an aspect basically consists of a *cut* and an *action*: a cut determines where an aspect applies, while an action specifies the effect of the aspect. Depending on the aspect language, specification of the *binding* between a cut and an action may not be decoupled: in traditional reflective systems, the binding between a hook in base code and a metaobject is usually standardized and not customizable, while in a language like AspectJ, it is tied to the action (advice definition).

Reflex relies on the notion of an explicit *link* binding a *cut* to an *action*. As a matter of fact, most practical AOP languages, like AspectJ, make it possible to define aspects as modular units comprising more than one pair cut-action.

In Reflex this corresponds to different links, with one action bound to each cut. Furthermore, AspectJ supports higher-order pointcut designators, like `cflow`. In Reflex, the implementation of such an aspect requires an extra link to expose the control flow information. There is therefore an abstraction gap between aspects and links: aspects are typically implemented by several links. This abstraction gap is discussed in more details and illustrated in Sect. 6.

Links are a mid-level abstraction, in between high-level aspects and low-level code transformation. This section overviews and illustrates how such an abstraction is provided and used in Reflex.

4.1 Types of Links

Cuts and actions can be either structural or behavioral. For instance, the UID aspect consists of a selection of structural elements, *i.e.* a *structural cut* (in that case, a set of classes), and a modification of a structural element, *i.e.* a *structural action* (adding several members to a class). Conversely, SOM relies on a selection of behavioral elements, *i.e.* a *behavioral cut* (method invocations), to which a *behavioral action* is associated (reifying calls as requests to be scheduled).

Our Java implementation of the model underlying Reflex is based on bytecode transformation using Javassist [8]. Due to the limitations of the Java standard environment with respect to modifying class definitions, we have to distinguish between two types of links. A *structural link*, termed S-link, binds a structural cut to an action, which can be either structural or behavioral. An S-link is *applied*, *i.e.* its associated action is performed, at load time. A *behavioral link*, called B-link, binds a behavioral cut to an action. A B-link applies at runtime.

We now illustrate structural links (Sect. 4.2) with the implementation of UID, and then show behavioral links (Sect. 4.3) with the implementation of SOM and ArgChecker. Finally, in Sect. 4.4, we discuss how Reflex operates at load time with respect to the different types of links.

4.2 Structural Links

A structural link binds a structural cut to some action (either structural or behavioral). In Reflex, a structural cut is a *class set*, defined intentionally by a *class selector*. For instance, the following class selector defines a cut consisting of the `Buffer` class only:

```
bufferSelector = new ClassSelector(){
    boolean accept(RClass aClass){
      return aClass.getName().equals("Buffer");
    }};
```

A class selector can base its decision on any introspectable characteristics of a reified class object, *down to the constituents of method bodies* (expressions in a method body are reified if needed). The object model of Reflex wraps and extends that of Javassist: `RClass` objects give access to their `RFields`, `RMethods`

and `RConstructors` (all `RMembers`); both methods and fields give access to their bodies as a sequence of `RExpr` objects.

An action bound to a structural cut is implemented in a *load-time metaobject*, instance of a class implementing the `LTMetaobject` interface. For instance, a `UIDAdder` is a metaobject that applies our UID aspect to a given class[2]:

```
public class UIDAdder implements LTMetaobject {
  static final String UID_FIELD = "private long _uid;";
  static final String UID_GET = "public long getUID(){return _uid;}";
  static final RClass UID_INTERFACE =
                       API.getRClass("reflex.lib.uid.UIDObject");

  void handleClass(RClass aRClass) {
    aRClass.addField(MemberFactory.newField(UID_FIELD, aRClass));
    aRClass.addMethod(MemberFactory.newMethod(UID_GET, aRClass));
    aRClass.addInterface(UID_INTERFACE);
} }
```

Since a load-time metaobject is part of the class loading process, it is a singleton created when the link is defined. A structural link is represented by an `SLink` object. For instance, the following excerpt defines a `uidLink`, which binds the previous `bufferSelector` to a `UIAdder` object:

```
SLink uidLink =
    API.links().addSLink(bufferSelector, new UIAdder(), "UID");
```

Configuration of the UID aspect (Sect. 3) is implemented as follows: the `UID` class holds a single S-link whose class selector is progressively updated by calls to `applyTo`.

4.3 Behavioral Links

This section is both a summary and an update of the model presented in [25]. This model is based on a standard model of behavioral reflection, where *hooks* are inserted in a program to delegate control to a *metaobject* at appropriate places. The particularity of our model lies in the possibility to flexibly group hooks into *hooksets*, and in having explicit and configurable *links* binding hooksets to metaobjects. A hookset corresponds to a set of program points, or static cut (*pointcut shadow* in AspectJ terminology [19]), and the metaobject corresponds to the action to be performed at these program points (*advice* in AspectJ terminology). The link is characterized by several *attributes*; for instance an *activation condition* may be attached to the link in order to avoid reification when a dynamically-evaluated condition is false. An exhaustive discussion of the mechanisms provided for specifying the dynamic part of a behavioral cut (*e.g.* residues) can be found in [23].

[2] The UID code could have been defined in a real class rather than with plain strings.

Defining a B-link. The BufferArgChecker aspect is defined in AspectJ[3]:

```
public aspect BufferArgChecker {
  pointcut checked(): execution(Buffer.put(..));
  around(): checked() { /* check args and possibly skip execution */ }
}
```

This aspect is translated into the following Reflex API calls by the AspectJ plugin (Sect. 6):

```
(1) Hookset putBuffer = new PrimitiveHookset(
      MsgReceive.class, new NameCS("Buffer"), new NameOS("put"));

(2) BLink buffCheck = API.links().addBLink("BufferArgChecker",
      putBuffer, new MODefinition.Class(BufferArgChecker.class));

(3) buffCheck.setControl(Control.AROUND);
(4) buffCheck.setScope(Scope.GLOBAL);
```

First, the cut of the aspect, *i.e.* executions of `put` on a `Buffer` is defined as a hookset `(1)`. A *primitive* hookset is defined by first giving an *operation class*, *e.g.* `MsgReceive`. An operation class represents a kind of operation we are interested in: this corresponds to a join point kind in AspectJ. The set of operations in Reflex is open, meaning the core of Reflex does not support any operation by itself, and can be extended [25]. The definition of the hookset then requires a class selector, which we already presented in the previous section (`NameCS` is a utility that selects classes based on their names); and an *operation selector*, which is a predicate selecting *operation occurrences* (`NameOS` is a utility also doing name-based selection) in program text. Primitive hooksets can be composed in order to obtain more complex hooksets.

The B-link is then defined by associating this hookset to the appropriate metaobject `(2)`. Metaobjects can be obtained either as new instances of a class of metaobjects, or from a factory. `BufferArgChecker` is a metaobject class implementing the desired validation behavior. At this stage the link is defined and operational. Still, we specify some of its attributes: the *control* attribute is set to *around* `(3)`, and since a single instance of the metaobject suffices, the *scope* of the link is set to *global* `(4)`.

As we have just seen, a B-link is represented at load time by a `BLink` object. Because of our implementation approach, a B-link is *set up* at load time and *applied* at runtime. An `RTLink` object represents a B-link during execution: it makes it possible to access and change metaobjects and activation conditions associated to a link at the appropriate level (object, class, or link). An `RTLink` object hence provides a link-specific runtime API for localized metaprogramming. There is a one-to-one relation between a `BLink` and an `RTLink`[4].

[3] For the sake of clarity, the advice accesses arguments via the `thisJoinPoint` object rather than via context exposure. Context exposure is briefly mentioned later.

[4] Due to implementation restrictions, the causal connection between both representations is not fully established: runtime changes to the definition of a link will not affect already-loaded classes (some changes are prohibited to avoid inconsistencies).

SOM is implemented similarly: the hookset for SOM consists of the entry points of the public methods of the classes to synchronize. The metaobject (scheduler), is instance-specific, hence the link has scope *object*. Furthermore, since a SOM scheduler needs to act both before and after a method invocation, the control is set to *before-after*.

Context exposure. The particularity of the implementation of SOM is that it makes use of the facilities of Reflex to specify the protocol between a cut and an associated action. This protocol is implemented as a metaobject protocol (MOP). If no custom protocol is specified, the default MOP for a given operation is used [25]. This is usually not efficient because it means reifying information that may not be needed at the metalevel.

In SOM, the scheduler gets control *before* method invocations via invocation of its `enter` method, which receives the name of the invoked method and its arguments, and *after* via its `exit` method, which does not take any parameter. Both `enter` and `exit` are defined in the `som.Scheduler` base class. This specialized MOP is specified as follows:

```
somLink.setMOCall(Control.BEFORE, "som.Scheduler", "enter",
                  new Parameter[]{ nameParam, argsParam });

somLink.setMOCall(Control.AFTER, "som.Scheduler", "exit");
```

The description of parameter generation (such as `nameParam` and `argsParam`) is open and relies on the extended Java language supported by the Javassist compiler, which is both expressive and efficient [8].

Apart from making it possible to program metaobjects without using overly generic protocols, MOP specialization represents a great source of performance improvement. The good performance and scalability of SOM, demonstrated in [5], was obtained thanks to this mechanism. A specialized MOP can be specified at the global level of operations like in traditional reflective systems where the reification of an operation occurrence is standardized and common. But it can also be specified more locally, at the link level, and even at the hookset level. This makes it possible to specialize context exposure at a fine-grained level.

4.4 Process Overview

In order not to modify the standard Java execution environment, behavioral links are *set up* at load time: during the *B-link setup* phase (BLS), hooks, along with necessary infrastructure, are installed in base code at the places indicated by the hookset definitions. Conversely, structural links are *applied* at load time. Since they can influence B-link setup, for instance by inserting a method whose execution is subject to a behavioral cut, the *S-link application* phase (SLA) is carried out before the BLS phase.

This two-phase process is illustrated in Fig. 2. Both phases follow a similar scheme: when a class is loaded, a *selection step* (a diamond in Fig. 2) determines the set of links that potentially apply. In SLA, links that select the loaded class

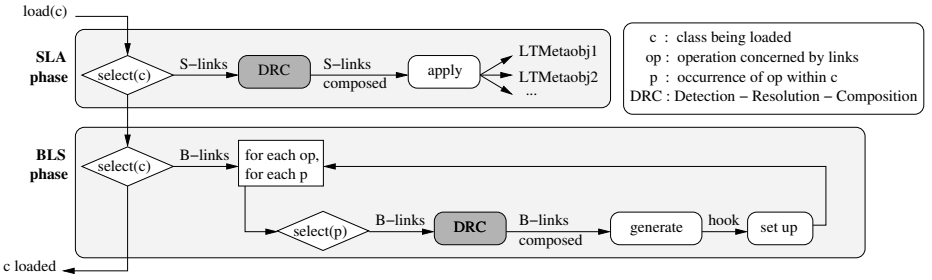


Fig. 2. Reflex operates in two phases at load time; (1) S-link application (SLA); (2) B-link setup (BLS)

are determined, while in BLS, selection goes down to operation occurrences in the class definition. If more than one link potentially apply, a detection-resolution-composition step (DRC in Fig. 2) occurs. Resolution is driven by user specifications; the kernel reports any unresolved interaction. Then links are appropriately composed. DRC is presented in Sect. 5. Finally, S-links are applied (in SLA), and B-links are set up (in BLS) after generating hook code.

5 Link Composition

Aspect composition is a challenging and multi-faceted issue, which is inherently impossible to resolve automatically. Five dimensions related to aspect composition have been identified in the literature, although we are not aware of any proposal addressing them all:

- *implicit cut*: an aspect that should apply whenever another applies [4, 11];
- *mutual exclusion*: an aspect that should not be applied whenever another applies [4, 11, 16];
- *aspects of aspects*: an aspect that applies *onto* another aspect [11];
- *visibility of aspectual changes*: when an aspect performs structural changes, their visibility to other aspects should be controllable [6];
- *ordering and nesting of aspects*: when several aspects apply at the same program point, their order of application must be specified [4, 11, 27].

AspectJ does not provide any support for mutual exclusion and visibility of aspectual changes, and is limited in terms of aspects of aspects and ordering/nesting of aspects. Conversely, Reflex provides initial support for these five dimensions of aspect composition. We hereby only briefly discuss implicit cut, aspects of aspects and visibility of aspectual changes, and pay more attention to mutual exclusion and ordering/nesting of aspects (details can be found in [23]).

An implicit cut is obtained by defining a link whose cut is shared with another link, and aspects of aspects are obtained by links whose cut affects the action (metaobject) of another link.

During the transformation process presented in Sect. 4.4, both the application of S-links and the setup of B-links effectively introspect and modify code

raising the issue of whether these modifications should be visible to others. In Reflex, a general-purpose *collaboration protocol* makes it possible to selectively expose or see changes made by other links. By default, Reflex ensures that structural changes made to a class *are not visible* to other links when they introspect the class. This avoids *unwanted* conflation of extended and non-extended functionalities, as discussed in the meta-helix architecture [6]. For our example, the default behavior of Reflex ensures that SOM does not *see* the `getUID` method added by the UID aspect, and hence this method is not subject to scheduling.

5.1 Interaction Detection

Brichau*et al.* [4], as well as AspectJ, only address means to *specify* composition, while Klaeren *et al.* [16] focus on means to *detect* interactions. But, as argued by Douence *et al.*, both detection and resolution of aspect interactions are crucial [11]. Thus we consider them as fundamental features of an AOP kernel.

Our approach follows a detection-resolution-composition (DRC) scheme [11]. The kernel ensures that interactions are detected, and notifies an *interaction listener* upon underspecification. The default interaction listener simply issues warning as shown in Sect. 3, but it is possible to use other listeners, *e.g.* for on-the-fly resolution. The kernel provides expressive and extensible means to specify the resolution of aspect interactions; from such specifications, it composes links appropriately.

An aspect interaction occurs when several aspects affect the same program point (execution or structure). This work is limited to a *static* approximation of aspect interactions. Hence we may detect spurious interactions, *i.e.* that do not occur at runtime. In the process illustrated in Fig. 2, selection steps determine the subset of links that (potentially) apply. If more than one link applies, then there is an interaction. For S-links, there is an interaction when a class being loaded belongs to more than one class set; for B-links, there is an interaction when an operation occurrence in program text belongs to more than one hookset.

In order to support mutual exclusion between aspects, Reflex provides *link interaction selectors*. An interaction selector can be attached to a link, and will be queried whenever the link is involved in an interaction, in order to determine whether it actually applies or not, depending on the other links present in the interaction. Resolving an interaction is hence carried out in two steps: *1)* selecting, within the current interaction, the subset of links that should be applied, and *2)* ordering and nesting the links of the subset. In the following section, we explain how this is supported in the case of B-links. The case of S-links is simpler due to the fact that S-links do not have a control attribute; nesting does not make sense.

5.2 Ordering and Nesting

The interaction between two before-after aspects can be resolved in two ways: either one always applies prior to the other (both before and after), or one "surrounds" the other [4, 11].

These alternatives can be expressed using composition operators, *seq* and *wrap*, dealing with sequencing and wrapping. Note that AspectJ only supports wrapping. Considering aspects that can act *around* an execution point (like ArgumentChecker in the example), the notion of aspect *nesting* as in AspectJ appears: a nested advice is only executed if its parent around advice invokes `proceed`. Around advices cannot be simply sequenced in AspectJ: they always imply nesting, and hence their execution always depends on the upper-level around advice [27].

In Reflex, link composition *rules* are specified using composition *operators*. The rule $seq(l_1, l_2)$ uses the *seq* operator to state that l_1 must be applied before l_2, both before and after the considered operation occurrence. The rule $wrap(l_1, l_2)$ means that l_2 must be applied within l_1, as clarified hereafter.

Kernel operators. User composition operators are defined in terms of lower-level kernel operators not dealing with links but with *link elements*. A link element is a pair $(link, control)$, where *control* is one of the control attributes: for instance, b_1 (resp. a_1) is the link element of l_1 for ***b**efore* (resp. ***a**fter*) control. There are two kernel operators, *ord* and *nest* which express respectively ordering and nesting of link elements. *nest* only applies to *around* link elements: the rule $nest(r, e)$ means that the application of the *a**r**ound* element r nests that of the link element e. The place of the nesting is defined by the occurrences of `proceed` within r. Sequencing and wrapping can hence be defined as follows:

$$seq(l_1, l_2) = ord(b_1, b_2), ord(r_1, r_2), ord(a_1, a_2)$$
$$wrap(l_1, l_2) = ord(b_1, b_2), ord(a_2, a_1), nest(r_1, b_1), nest(r_1, r_2), nest(r_1, a_2)$$

Composition operators. Reflex makes it possible to define a handful of user operators for composition on top of the kernel operators. For instance, `Seq` and `Wrap` are binary operators that implement the *seq* and *wrap* operators as defined above:

```
class Wrap extends CompositionOperator {
  void expand(Link l1, Link l2){
    ord(b(l1), b(l2)); ord(a(l2), a(l1));
    nest(r(l1), b(l2)); nest(r(l1), r(l2)); nest(r(l1), a(l2));
} }
```

The methods `b` (before), `r` (around), `a` (after), `ord`, and `nest` are provided by the `CompositionOperator` abstract class. The way user operators are defined in terms of kernel operators is specified in the `expand` method.

Higher-level composition operators can also express mutual exclusion between links. For instance, in Event-based AOP, binary operators like *fst* (resp. *snd*) are proposed, expressing that if the left child applies, then the right child does not apply (resp. applies) [12]. *fst* can be implemented by specializing `Seq`, using a link interaction selector stating that `l2` does not apply if `l1` does.

At the kernel level no language support is provided to define rules conveniently: they need to be manually instantiated, node by node (recall the example

in Sect. 3). Language support for Reflex configuration (discussed in Sect. 6) can be used to define languages dedicated to composition, or to define languages that include syntactic support for composition. For instance, the notion of precedence of the Reflex version of AspectJ is implemented with `Wrap`.

5.3 Hook Generation

When detecting link interactions, the composition algorithm of Reflex generates a hook skeleton based on the specified composition rules. During this generation Reflex issues warnings whenever composition is under-specified. Users are free to ignore them and let Reflex arbitrarily compose the non-specified parts. The hook skeleton is then used for driving the hook generation process. In order to support nesting of aspects with `proceed`, Reflex adopts a strategy similar to that of AspectJ, based on the generation of closures.

6 Plugin Architecture for Open Language Support

A versatile AOP kernel provides means to modularly define aspect languages, either general-purpose or domain-specific, so that programmers can implement aspects at the level of abstraction that most suits their needs. In Reflex, an aspect language is implemented by a translator to kernel configuration, called a *plugin*. A plugin takes as input an aspect program written in a given language and outputs, either on-line or off-line, the adequate Reflex configuration: links, metaobject classes, selectors, etc., together with calls to the kernel API. The SOM DSAL and (a subset of) AspectJ are the two first aspect languages we have developed for the Reflex AOP kernel[5].

Bridging the Abstraction Gap. A Reflex plugin is typically expected to bridge the abstraction gap between the aspect level and the kernel level. At the kernel level, the main conceptual handle is the notion of *links*. Though making it possible to abstract from low-level details, links are lower-level abstractions than *aspects*. As a result, an aspect is typically implemented by several links.

This abstraction gap can be observed in different scenarios, for instance considering AspectJ support. First, an AspectJ aspect definition may include several pointcuts and advices, plus inter-type declarations; each will be implemented by (at least) one link. Second, the implementation of aspects with higher-order pointcuts requires several links. For instance, if the ArgumentChecker aspect is extended with a control flow restriction so that nested calls to checked calls are not checked (using `!cflowbelow(p)`), two B-links will be used: the *advice link* for binding the validation behavior, and the *cflow link* for exposing control flow information of the nested pointcut `p`. The cflow link is a before-after link using a simple counter (increased on before, decreased on after). The restriction that

[5] All the code (including plugins and the running example) can be obtained from: `http://reflex.dcc.uchile.cl`.

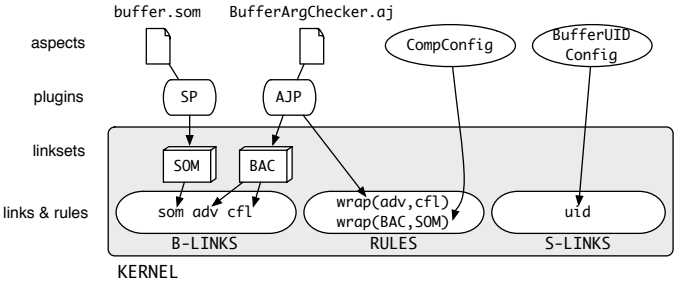


Fig. 3. The running example with the different aspects and their mapping in the Reflex AOP kernel

the around advice only applies when not below the control flow of an already-checked call is implemented by adding an activation condition to the advice link that checks the value of the counter.

The major issue with this abstraction gap is related to composition. Composition of links related to the same pointcut-advice should be addressed: in the case of control flow above, depending on the order in which the two links are composed, one either obtains the semantics of the `cflow` pointcut designator (first cflow link, then advice link), or that of `cflowbelow` (first advice link, then cflow link).

Links related to the same aspect may also need to be composed. For this issue, AspectJ adopts a syntactic rule whereby advice precedence is defined by the order of definitions in the aspect file. We believe this (implicit) syntactic rule is error-prone (just imagine moving code around). Our approach rather makes such a composition issue explicit and offers more flexible means for its resolution.

When composing aspects that may be written in different languages (implemented by different plugins), the aspect programmer does not care about links. However, all the composition mechanisms of Reflex (interaction notification, composition rules, resolution and generation) work with links, not aspects. In order to support traceability of a link back to its associated aspect-level entity, we introduce *linksets* as a means to group a set of links that are part of the same higher-level conceptual entity.

A linkset is therefore the counterpart, in the kernel world, of an entity in the aspect world. The mapping is defined by the plugin. Reflex accepts linksets in composition rules: they stand for all their links. This semantics is similar to that of AspectJ, where an aspect stands for all its advices in a precedence relation.

Illustration. Fig. 3 illustrates the overall architecture of our running example. The application of SOM to `Buffer` is expressed using the SOM DSAL, implemented by the SOM Plugin (`SP`): it results in the definition of one B-link (`som`), embedded in a linkset (`SOM`). The BufferArgChecker (`BAC`) aspect is expressed in AspectJ, implemented by the AspectJ Plugin (`AJP`): it results in the definition of one linkset `BAC`, encapsulating two B-links, one for the advice (`adv`) and one for the cflow (`cfl`) links, and in the definition of a composition rule `wrap(adv, cfl)`

to ensure the `cflowbelow` semantics. The UID aspect is expressed in the configuration class `BufferUIDConfig`, directly adding an S-link (`uid`). And finally, composition between SOM and BufferArgChecker is done in the configuration class `CompConfig`, declaring a composition rule `wrap(BAC, SOM)`.

7 Conclusion

We have proposed an architecture for versatile kernels for multi-language AOP: basic facilities for behavioral and structural transformation, composition handling and language support. The Reflex kernel relies on a reflective model that provides mid-level abstractions to designers of aspect languages: links are a simple abstraction for both transformation and composition. We have exposed the major features of composition support in the Reflex kernel: automatic detection of interactions between aspects and expressive, extensible means for their explicit resolution. A plugin architecture makes it possible to modularly define aspect languages, bridging the abstraction gap between links and aspects. We have illustrated the resolution of interactions between aspects defined in different aspect languages. Future work includes experimenting with more aspect languages in complex scenarios in order to study the scalability of our approach and refine our initial treatment of aspect composition.

Acknowledgments. We thank Leonardo Rodríguez for his work on the AspectJ plugin, and Guillaume Pothier and the anonymous reviewers for their comments.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. of ASE 2003*, pages 196–204, Montral, Canada, March 2003. IEEE CS Press.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. Technical Report abc-2004-1, The abc Group, September 2004.

[3] D. Batory, C. Consel, and W. Taha, editors. *Proc. of GPCE 2002*, volume 2487 of *LNCS*, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[4] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [3], pages 110–127.

[5] D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In M. Odersky, editor, *Proc. of ECOOP 2004*, number 3086 in LNCS, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[6] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proc. of ISOTAS 1996*, volume 1049 of *LNCS*, pages 157–172. Springer-Verlag, 1996.

[7] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In Lieberherr [17], pages 102–111.

[8] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In F. Pfenning and Y. Smaragdakis, editors, *Proc. of GPCE 2003*, volume 2830 of *LNCS*, pages 364–376, Erfurt, Germany, September 2003. Springer-Verlag.
[9] The Concern Manipulation Environment website, 2002.
[10] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
[11] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [3], pages 173–188.
[12] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [17], pages 141–150.
[13] R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
[14] J. Irwin, J.-M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *LNCS*. Springer-Verlag, 1997.
[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. of ECOOP 2001*, number 2072 in LNCS, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
[16] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proc. of GCSE 2000*, volume 2177 of *LNCS*, pages 57–69. Springer-Verlag, 2000.
[17] K. Lieberherr, editor. *Proc. of AOSD 2004*, Lancaster, UK, March 2004. ACM.
[18] C.V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
[19] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proc. of CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer-Verlag, 2003.
[20] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997.
[21] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [17].
[22] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 DDD Track*, October 2003.
[23] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Univ. of Nantes and Univ. of Chile, November 2004.
[24] É. Tanter and J. Noyé. Motivation and requirements for a versatile AOP kernel. In *EIWAS 2004*, Berlin, Germany, September 2004.
[25] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In R. Crocker and G. L. Steele, Jr., editors, *Proc. of OOPSLA 2003*, pages 27–46. ACM, October 2003.
[26] M. Wand. Understanding aspects. In *Proc. of ICFP 2003*. ACM, 2003.
[27] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

Semi-inversion of Guarded Equations

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark
torbenm@diku.dk

Abstract. An *inverse* of a program is a program that takes the output of the original program and produces its input. A *semi-inverse* of a program is a program that takes some of the input and some of the output of the original program and produces the remaining input and output. Inversion is, hence, a special case of semi-inversion.

We propose a method for inverting and semi-inverting programs written as guarded equations. The semi-inversion process is divided into four phases: Translation of equations into a relational form, refining operators, determining evaluation order for each equation of the semi-inverted functions and translation of semi-inverted functions back to the original syntax. In cases where the method fails to semi-invert a program, it can suggest which additional parts of the programs input or output are needed to make it work.

1 Introduction

Many programming problems involve making two programs that are each others semi-inverses, for example encryption and decryption of text, where the encryption program takes a clear text and a key and produces a cypher text while the decryption program takes a cypher text and key and produces the clear text. It is usually up to the programmer to ensure that the two programs are, indeed, semi-inverses and to maintain this property when the programs are modified.

Ideally, the programmer should only write one of the two programs and then let the computer derive the other. While this is not realistic for all cases (a public-key encryption function is, for example, deliberately made difficult to semi-invert), it can be useful to have systems that works only some of the time, as long as "some of the time" isn't "nearly never".

In general, semi-inversion means taking a program and producing a new program that as input takes part of the input and part of the output of the original program and as output produces the rest of the input and output of original program.

Not all semi-inverses of programs are well-defined: The provided parts of input and output may not be sufficient to uniquely determine the remaining parts. But it is always possible to extend an ill-defined semi-inverse to a well-defined semi-inverse by providing additional parts of the input or output. In the extreme, one can provide all of the input (and possibly some of the output) of the original program as input to the semi-inverse. This makes the output of the semi-inverse uniquely determined by its input, though there might be cases where the semi-inverse is a partial function. For example, it is possible to define a semi-inverse that takes *all* of the input and output

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 189–204, 2005.

of the original program. This semi-inverse will produce an empty output, but is defined only on input/output pairs where the output is what the original program would produce for the given input.

Our approach will ask for extra inputs to semi-inverses when it is unable to find a well-defined semi-inverse. Even semi-inverses that are mathematically well defined may require extra inputs by the method, as it may not be able to find a program that implements the mathematical semi-inverse.

2 Formalism for Semi-inverses

We start by recalling the familiar definition of inverse functions:

A partial function g is the inverse of an injective partial function f if for all x in f's domain we have that $f(x) = y \Rightarrow g(y) = x$. g is only defined on the range of f.

The equation for semi-inverses is not as simple, as we need to talk about parts of inputs and outputs.

Projections [10,9] have been used in theory about partial evaluation to talk about parts of the input. They could also be used here, but we prefer less heavy machinery, so we introduce *extractions*.

Extractions. An *extraction* for a domain α is a total and surjective function from α to a domain β. Intuitively, an extraction finds some information about its input and injects this into an output domain β in such a way that there are no redundancies.

A simple extraction is the function that takes the first component of a pair, but there are also more complex extractions, such as any total predicate (in which case the output domain is the set of boolean values). We will, in practice, use only simple extractions that "throw away" parts of tuples, including the identity function and the function that throws away all input, i.e., whose range is the one-element domain (called "`unit`" in SML and "()" in Haskell).

We call a pair of extractions $p : \alpha \to \beta$ and $q : \alpha \to \gamma$ *a division of* α if the function $(p,q) : \alpha \to \beta \times \gamma$ defined by $(p,q)(x) = (p(x),q(x))$ is bijective. Intuitively, this means that no part of (or information about) x is thrown away by both p and q and no part of (or information about) x is retained by both p and q.

Note that the bijection requirement is on the type of the arguments and results of the function, it is not required that the division is bijective on the set of valid argument/result pairs of the function. Indeed, this would generally not be possible, as the result is functionally dependent on the argument.

The semi-inverse equation. Given a partial function $f : \alpha \to \beta$ and a division (p,q) of $\alpha \times \beta$, where $p : \alpha \times \beta \to \gamma_1$ and $q : \alpha \times \beta \to \gamma_2$, we say that a partial function $g : \gamma_1 \to \gamma_2$ is a semi-inverse of f with respect to (p,q) if $f(x) = y \Rightarrow g(p(x,y)) = q(x,y)$.

Example: If $f(x,y) = x + y$ and $p((x,y),z) = (x,z),\ q((x,y),z) = y$ then the function g defined by $g(x,z) = z - x$ is a semi-inverse of f with respect to (p,q), as $x + y = z \Rightarrow y = z - x$.

Not all divisions define valid semi-inverses for a function. For example, if p and q were reversed in the example above, then g would not be well-defined (there would be no unique result).

$$
\begin{array}{lll}
\textbf{fid} & = & \textit{function identifiers} \\
\textbf{vid} & = & \textit{variable identifiers} \\
\textbf{cid} & = & \textit{constructor identifiers} \\
\textbf{oid} & = & \textit{operator identifiers} \\
\textbf{pid} & = & \textit{predicate identifiers} \\
\textbf{num} & = & \textit{numbers}
\end{array}
$$

$$\textit{Program} \rightarrow \textit{Function}^{+}$$

$$\textit{Function} \rightarrow \textit{Equation}^{+}$$

$$\textit{Equation} \rightarrow \textbf{fid}\ \textit{Pattern} \mid \textit{Guard} = \textit{Exp}$$

$$
\begin{array}{lll}
\textit{Pattern} & \rightarrow & \textbf{num} \\
 & \mid & \textbf{vid} \\
 & \mid & (\textit{Pattern}, \ldots, \textit{Pattern}) \\
 & \mid & \textbf{cid}\ \textit{Pattern}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Guard} & \rightarrow & \varepsilon \\
 & \mid & \textbf{pid}\ \textit{Exp} \\
 & \mid & \textit{Guard}\ \&\&\ \textit{Guard}
\end{array}
$$

$$
\begin{array}{lll}
\textit{Exp} & \rightarrow & \textbf{num} \\
 & \mid & \textbf{vid} \\
 & \mid & (\textit{Exp}, \ldots, \textit{Exp}) \\
 & \mid & \textbf{cid}\ \textit{Exp} \\
 & \mid & \textbf{fid}\ \textit{Exp} \\
 & \mid & \textbf{oid}\ \textit{Exp} \\
 & \mid & \texttt{let}\ \textit{Pattern} = \textit{Exp}\ \texttt{in}\ \textit{Exp}
\end{array}
$$

Fig. 1. Syntax

3 Language

The programming language that we use in this paper is a first-order functional language where each function is given as a set of equations using pattern-matching and guards. The equations of a function are unordered and their domains must, hence, be visibly disjoint through their patterns and guards, i.e., for any given input at most one equation will have matching pattern and guard. We allow partial functions, so it is allowed that some inputs have no matching equation. If at any point during execution no matching equation for a call can be found, the result of the entire computation in undefined.

Patterns *can* have repeated variables, which means that the matching values must be equal. A let-expression does *not* introduce a new scope, so if a variable defined in a let-pattern is defined previously in the same equation, the two definitions must define equal values, just like repeated variables in a pattern. This is different from the usual semantics of let-expressions in functional languages, but it is convenient for the inversion method, as repeated uses of a variable in the original program may become repeated definitions of it in a semi-inverted program. Translation to and from traditional first-order functional languages is not difficult.

In an equation, all variables occurring in the guards must also occur in the pattern on the left-hand side of the equation. Variables occurring in the expression must be defined in the pattern on the left-hand side of the equation or in an enclosing let-expression.

Values can be numbers, (possibly empty) tuples of values or elements of recursive datatypes in the style of ML or Haskell. The syntax is reminiscent of Haskell (in spite of the semantic differences) and can be seen in figure 1. Note that the "|" in the rule for *Equation* is part of the syntax of a guarded equation and not a grammar symbol denoting choice of several production. We have not specified the set of operators or predicates, though the latter must at least include an equality test. We also allow partial functions that return the empty tuple as predicates. These are considered to succeed if the call returns a value and fail if the function call is undefined. Such functional predicates may be constructed from "normal" functions during semi-inversion.

The syntax shows constructors, operators and predicates as prefix operators operating on one argument, which may be a tuple. When writing programs in the language, we will, for readability, sometimes use infix notation for these.

Example. A sample program is shown below.

```
i2p (0, 0) | true = nil
i2p (n, i) | n>0 = insert (i2p (n-1,i `div` n), n, i `mod` n)

insert (xs, n, 0) | true = n : xs
insert (x:xs, n, i) | i>0 && x/=n = x : insert (xs, n, i-1)
```

The function `i2p` takes two numbers n and i and produces a list of the numbers $1 \ldots n$ permuted with the permutation with index i, where $0 \leq i < n!$ (using one of several possible enumerations of permutations).

The guard $\texttt{x} \neq \texttt{n}$ in the second equation for `insert` is not strictly required to make the equations disjoint, but it turns out we will need the bit of invariant it represents to make a nontrivial semi-inversion. We will discuss this issue later.

4 Semi-inversion of a Program

A semi-inverted program will consist of a set of semi-inverses of the original functions, possibly having none or several different semi-inverses of some of the original functions using different divisions.

The user specifies which semi-inverses he desires by specifying a function and division for each. These specifications forms the initial set of *desired semi-inverses*.

We may, in the course of transformation, find that we need to call other semi-inverted functions. We will, then, add their specifications to the list of desired semi-inverses and attempt to make definitions of these.

We may, also, find that we are not able to make a definition of a desired semi-inverse, as we can not uniquely define its output in terms of its input. In this case, we add the specification to a list of *invalid semi-inverses* and start over. When we want to add a semi-inverse f' to the list of desired semi-inverses, we must first check the list of invalid semi-inverses. If the specification of f' is found there, we must go back and

see if we can use another (not invalidated) semi-inverse instead. If we can not go back (i.e., if f' is in the initial, user specified, top-level list of semi-inverses), semi-inversion has failed, but the information gathered by the method may suggest alternative semi-inverses that may be useful in spite of not being exactly what the user desired.

To summarize, we need to implement the following:

1. A procedure for determining if a specification of a semi-inverse is invalid, given a list of semi-inverses already found to be invalid.
2. A procedure for finding out which other semi-inverses are needed to define a desired semi-inverse.
3. A procedure to construct the definition of a valid semi-inverse.
4. A procedure for suggesting which extra inputs should be added to an invalid semi-inverse in an attempt to extend it to a valid semi-inverse.

Given these, we can keep updating the lists of desired and invalid semi-inverses until all desired semi-inverses are definable in terms of predefined operators and other desired semi-inverses and and none of them are in the list of invalid semi-inverses. We will, eventually, reach such a state as we can, in the worst case, add all of the inputs of the original functions as extra inputs to the semi-inverses, which makes them trivially valid.

Example. We will semi-invert `i2p` from figure 3 with a division (p,q), where $p(n,i,xs) = (n,xs)$ and $q(n,i,xs) = i$, i.e., making a function `i2p'` that takes a number n and permutation xs of the numbers $1 \ldots n$ and returns i, where i is the index of the permutation. We initialize the list of desired semi-inverses to hold one semi-inverse of `i2p` with the division (p,q) as defined above.

4.1 Desequentialisation

The remaining parts of the transformation are simplified if we don't have nested expressions or patterns and if the order of evaluation is less explicit, so we first translate each equation to an unordered set of relations between tuples of variables. We call this process *desequentialisation*.

Figure 2 shows the syntax of the relational form and figure 3 shows the desequentialisation of a single equation. I translates an equation into an equation in the relational form. I_p translates a pattern into a set of relations and I_e translates an expression into a set of relations. I_v is an auxiliary function that is used when a singleton variable is required. If necessary, it adds an extra relation that binds a non-variable to a new variable. The guards are left unchanged by this transformation. Note that constructed syntax is shown in `typewriter font` to distinguish from meta-level syntax used to define the translation function (which is shown as "normal" text). Function calls are translated into calls of relations between input and output variables (using the function name as name for the relation). Guards are already relational, so they are just added to to the set of relations.

Note the similarity between the relations for patterns and structure-building expressions: It is not *a priori* given which side defines the other. The same will be true for some of the other relations: They can be read either as defining the left-side variables

$$
\begin{array}{lll}
\textbf{fid} & = & \textit{function identifiers} \\
\textbf{vid} & = & \textit{variable identifiers} \\
\textbf{cid} & = & \textit{constructor identifiers} \\
\textbf{oid} & = & \textit{operator identifiers} \\
\textbf{pid} & = & \textit{predicate identifiers} \\
\textbf{num} & = & \textit{numbers}
\end{array}
$$

$$\textit{Program} \rightarrow \textit{Function}^{+}$$

$$\textit{Function} \rightarrow \textit{Equation}^{+}$$

$$\textit{Equation} \rightarrow \textbf{fid}\ \textit{Vars}\ \texttt{where}\ \textit{Relation}^{*}$$

$$
\begin{array}{lll}
\textit{Vars} & \rightarrow & \textbf{vid} \\
 & \mid & (\textbf{vid}, \ldots, \textbf{vid})
\end{array}
$$

$$
\begin{array}{lll}
\textit{Relation} & \rightarrow & \textbf{vid} = \textbf{num} \\
 & \mid & \textit{Vars} = \textit{Vars} \\
 & \mid & \textbf{vid} = \textbf{cid}\ \textit{Vars} \\
 & \mid & \textbf{fid}\ \textit{Vars} \\
 & \mid & \textbf{pid}\ \textit{Vars} \\
 & \mid & \textit{Vars} = \textbf{oid}\ \textit{Vars}
\end{array}
$$

Fig. 2. Syntax of relational form

in terms of the right-side variables or *vice versa*. They can even be read "sideways", determining any subset of the variables in terms of the rest (like semi-inverses). This undirected view of the relations will be the essence of the semi-inversion method.

Example. The example program from section 3 is shown below in relational form.

```
i2p (x1,x2,x3) where
  {x1 = 0, x2 = 0, x3 = nil}
i2p (n,i,x1) where
  {n>0, insert (x2,n,x3,x1), i2p (x4,x5,x2), x4 = sub1 n,
   x5 = div (i,n), x3 = mod (i,n)}

insert (xs,n,x1,x2) where
  {x1 = 0, x2 = : (n,xs)}
insert (x1,n,i,x2) where
  {x1 = : (x,xs), x2 = : (x,x3), i>0, x/=n, insert (xs,n,x4,x3), x4 = sub1 i}
```

Note that `n-1` has been translated as `sub1 n` instead of using a binary subtraction operator. While this is not strictly necessary, it will make the example a bit less cumbersome.

4.2 Refining Operators

In order to make semi-inversion of operators possible more often, we can refine the operators in different ways:

$$
\begin{array}{lll}
f & \in \textbf{fid} = \textit{function identifiers} \\
x, y, x_i & \in \textbf{vid} = \textit{variable identifiers} \\
c & \in \textbf{cid} = \textit{constructor identifiers} \\
o & \in \textbf{oid} = \textit{operator identifiers} \\
k & \in \textbf{num} = \textit{numbers} \\
\\
I & : \textit{Equation} \to \textit{Equation}' \\
I[f\ P \mid G\ =\ E] & = f\ (V_i \cup V_o)\ \texttt{where}\ (R_i \cup R_o \cup R_g) \\
 & \quad \text{where } (R_i, V_i) = I_p[P],\ \ (R_o, V_o) = I_e[E],\ \ R_g = I_g[G] \\
\\
I_p & : \textit{Pattern} \to \textit{RelationSet} \times \textit{Vars} \\
I_p[(P_1, \ldots, P_n)] & = (R_1 \cup \cdots \cup R_n,\ (x_1, \ldots, x_n)) \\
 & \quad \text{where } (R_1, x_1) = I_v(I_p[P_1]),\ \cdots,\ (R_n, x_n) = I_v(I_p[P_n]) \\
I_p[k] & = (\{x = k\}, x) \quad \text{where } x \text{ is a new variable} \\
I_p[y] & = (\{\}, y) \\
I_p[c\ P] & = (R \cup \{x = c\ V\}, x) \\
 & \quad \text{where } (R, V) = I_p[P] \ \text{ and } \ x \text{ is a new variable} \\
\\
I_g & : \textit{Guard}^* \to \textit{RelationSet} \\
I_g[\varepsilon] & = \{\} \\
I_g[p\ E] & = R \cup \{p\ V\} \\
 & \quad \text{where } (R, V) = I_e[E] \ \text{ and } \ x \text{ is a new variable} \\
I_g[G_1\ \&\&\ G_2] & = I_g[G_1] \cup I_g[G_2] \\
\\
I_e & : \textit{Exp} \to \textit{RelationSet} \times \textit{Vars} \\
I_e[k] & = (\{x = k\}, x) \quad \text{where } x \text{ is a new variable} \\
I_e[y] & = (\{\}, y) \\
I_e[(E_1, \ldots, E_n)] & = (R_1 \cup \cdots \cup R_n,\ (x_1, \ldots, x_n)) \\
 & \quad \text{where } (R_1, x_1) = I_v(I_e[E_1]),\ \cdots,\ (R_n, x_n) = I_v(I_e[E_n]) \\
I_e[c\ E] & = (R \cup \{x = c\ V\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \ \text{ and } \ x \text{ is a new variable} \\
I_e[f\ E] & = (R \cup \{f\ (V \cup x)\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \ \text{ and } \ x \text{ is a new variable} \\
I_e[o\ E] & = (R \cup \{x = o\ V\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \ \text{ and } \ x \text{ is a new variable} \\
I_e[\texttt{let}\ P = E_1\ \texttt{in}\ E_2] & = (R_0 \cup R_1 \cup R_2 \cup \{V_0 = V_1\}, V_2) \\
 & \quad \text{where } (R_0, V_0) = I_p[P],\ \ (R_1, V_1) = I_e[E_1],\ \ (R_2, V_2) = I_e[E_2] \\
\\
I_v & : \textit{RelationSet} \times \textit{Vars} \to \textit{RelationSet} \times \textbf{vid} \\
I_v[(R, V)] & = (R, V) \quad \text{if } V \in \textbf{vid} \\
I_v[(R, V)] & = (R \cup \{x = V\}, x) \quad \text{if } V \notin \textbf{vid} \\
 & \quad \text{where } x \text{ is a new variable}
\end{array}
$$

Fig. 3. Translation to relational form

- The guard of an equation may restrict the range of inputs to an operator enough that it makes semi-inversion possible more often. For example, if the guard ensures x is even, we can refine the relation $z = x\ \texttt{div}\ 2$ to $x = 2 * y$, where any of the two variables can define the other.
- Related operators with the same inputs may be combined into one operator that is easier to semi-invert. For example, $z = \texttt{div}\ (x, y)$ and $v = \texttt{mod}\ (x, y)$ can be com-

bined into one relation $(z,v) = \texttt{divmod}\ (x,y)$, using an operator `divmod`, that returns both fraction and remainder after division by y. Given this, we can from z, v and y find x, which can't be found from any one of the two relations individually even when all of z, v and y are known.
- Relations of the form $(x_1,\ldots,x_n) = (y_1,\ldots,y_n)$ are translated into separate relations $x_1 = y_1,\ldots,x_n = y_n$.

We may combine several of these approaches, for example by joining two related operators and then use the guard to restrict the range.

Example. The operators used in the program are `sub1`, `div` and `mod`. `sub1` requires no refinement as any of the two variables defines the other. `div` and `mod`, on the other hand, can be combined as they both take the same arguments. This means we replace the relations `x5 = div (i,n), x3 = mod (i,n)` with `(x5,x3) = divmod (i,n)`.

4.3 Resequentialisation

When semi-inverting an relational equation $f\ V_f$ `where` R with respect to a division (p,q), where p and q are functions from the tuple V_f to the input and output tuples of the semi-inverted equation, we need to check if a thusly defined semi-inverse is valid, i.e., if the value of $q(V_f)$ is uniquely defined from the value of $p(V_f)$ through the relations R. Hence, we need to determine which variables can be computed from which others, starting from $p(V_f)$. We do this by *resequentialisation*, which orders relations by data dependency. We keep a list of known variables K and a list S of relations from R that depend only on the variables in K. At the end, the relations in S will be in a valid evaluation order.

$K = p(V_f)$ (treated as a set of variables)
$S = \varepsilon$
`while there is a relation` r `in` R `that is determined by` K
  $S := S, r;$
  $R := R \setminus \{r\}$
  $K := K \cup Vars(r)$

A relation r is determined by K if all variables in the relation can be defined through variables that are already contained in K. For structural relations (tuples and constructors), this means that all variables on any one side of $=$ must be defined. Predicates normally need all variables in order to be defined, but equality predicates can be resolved if any one side is defined.

Arithmetic operators are treated as relations between input and output. For example $z = x + y$ is a relation between three variables.

Semi-inverting an operator is possible when all variables can be uniquely determined from those that are known. For the $z = x + y$ example, this is true when any two of the variables are known. For all standard operators, we list the subsets of inputs and outputs that can make the remaining defined, each with the name of the corresponding semi-inverse operator. For example, the operator $+$ will be described by the list

$[z = +(x,y),\ x = -(z,y),\ y = -(z,x)]$. What this means is that the first relation in the list can be replaced by any of the other relations.

For function calls, we don't know *a priori* which semi-inverses of the function are valid. We will, hence, assume that any non-empty subset of the variables in the relation is enough to define the remaining variables, *unless* this subset corresponds to a semi-inverse in the list of invalid semi-inverses. If we later need to invalidate the semi-inverse, we must redo the resequentialisation.

If, at the end, K contains all variables in the relation set (or even just in $q(V_f)$), we know that we can evaluate $q(V_f)$ from $p(V_f)$, so we can semi-invert the current equation for f with respect to (p,q), assuming the called semi-inverses are all valid. We add the called semi-inverses to the list of desired semi-inverses, so we will check their validity later.

If, on the other hand, at the end of resequentialisation there are variables from $q(V_f)$ that are not contained in K, the semi-inverse is not valid. So we add it to the list of invalid semi-inverses and start over.

Example. We recall that we want to semi-invert `i2p` with the first argument and the result known. In the relational form in section 4.1, this means the first and last parameter.

Starting with the first equation for `i2p`, we initialise K to $\{$`x1, x3`$\}$, so we can immediately add `x1 = 0` and `x3 = nil` to S. As one side of the equality `x2 = 0` is known, we can add this too and add `x2` to K, which concludes resequentialisation of this equation with S equal to

```
x1 = 0, x3 = nil, x2 = 0
```

The second equation starts with $K = \{$`n, x1`$\}$. We can immediately add `n>0` to S. The list of valid semi-inverses of the `sub1` operator is `[y = sub1 x, x = add1 y]`, so we can add `x4 = sub1 n` to S and `x4` to K. At this point, we can only add semi-inverses of functions to the list. We choose `insert`, where we know two of the four parameters. Hence, we add `insert` with known second and last parameter to the list of desired semi-inverses, add `insert (x2,n,x3,x1)` to S and `x2, x3` to K. We can then add `i2p (x4,x5,x2)` to S and `x5` to K. We already have this semi-inverse in the list of desired semi-inverses, so we continue.

We replaced `x5 = div (i,n)` and `x3 = mod (i,n)` by `(x5,x3) = divmod(i,n)`, where the semi-inverses of `divmod` are described by `[(z,v) = divmod (x,y), x = MLA (z,y,v)]` with `MLA (z,y,v)` meaning `z*y+v`, provided `0`$\leq$`v`$<$`y` (and undefined otherwise). Hence, we can add `(x5,x3) = divmod(i,n)` to S and i to K. This concludes the resequentialisation of the second equation for `i2p` with S equal to

```
n>0, x4 = sub1 n, insert (x2,n,x3,x1), i2p (x4,x5,x2), (x5,x3) = divmod(i,n)
```

We added a semi-inverse of `insert` with second and fourth parameters known to the list of desired semi-inverses, so we need to resequentialise the equations for that. Skipping details, we get the sequences

```
x2 = : (n,xs), x1 = 0
x2 = : (x,x3), x/=n, insert (xs,n,x4,x3), x1 = : (x,xs), x4 = sub1 i, i>0
```

for the two equations. We have now resequentialised all desired semi-inverses.

4.4 Translation Back into Guarded Equation Form

When we have resequentialised all desired semi-inverses, we translate each equation from relational form back into "normal" syntax.

Resequentialisation gives us a list of relations that defines a possible order of evaluation. Together with a division, we wish to translate this into an equation in the original syntax, *i.e.*, into something of the form $f'\ P \mid G' = E$ where P is a pattern for $p(V_f)$, E is an expression that defines the value of $q(V_f)$ in terms of the variables in P, and the guard G' is a guard using these same variables.

We start with the patterns. P will be created from $p(V_f)$ by finding a relation where a variable in $p(V_f)$ is related to a structure (tuple or constructor application), substituting this structure for the variable in the pattern, and then repeating this process for the variables in the substructure.

$$
\begin{array}{ll}
x, y, x_i & \in \textbf{vid} = \textit{variable identifiers} \\
c & \in \textbf{cid} = \textit{constructor identifiers} \\
 & \\
M_p & : \mathit{Relation}^* \to \mathit{Vars} \to \mathit{Pattern} \times \mathit{Relation}^* \\
M_p[S](x_1, \ldots, x_n) & = ((p_1, \ldots, p_n), S_n) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x_1, \ldots, (p_n, S_n) = M_p[S_{n-1}]x_n) \\
M_p[\{x = k\}, S]x & = (k, S) \\
M_p[\{x = (x_1, \ldots, x_n)\}, S]x & = M_p[S](x_1, \ldots, x_n) \\
M_p[\{x = c\ x_1\}, S]x & = (c\ (p_1), S_1)) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x_1) \\
M_p[\{x = y\}, S]x & = M_p[S]y \\
M_p[r, S]x & = (p_1, (r, S_1)) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x \\
M_p[\varepsilon]x & = (x, \varepsilon)
\end{array}
$$

Fig. 4. From relations to pattern

Figure 4 shows how a pattern and a reduced list of relations is made from a variable or tuple of variables and a relation list. The rules are applied in precedence from top to bottom. If no other rule applies, the last is used, so the variable itself is used as pattern. Note that this procedure may result in a variable occurring several times in a pattern, but this is O.K., as we allow nonlinear patterns.

Relations in S that only contain variables from the pattern can be considered as tests and will be part of the new guard, so we remove such relations from the set and build an expression from the remaining relations. Figure 5 shows how the new guard and the reduced list of relations are constructed by M_g and $M_{\overline{g}}$, respectively, from the relation list S and the set V of variables in the pattern p returned by M_p.

Figure 6 shows how we build an expression given a list of relations R and a tuple of desired variables V_{out} (where $V_{out} = q(V_f)$). To choose the correct semi-inverses for operators and function calls, we maintain a list of known variables. This is initialised to be the set of variables contained in the pattern returned by M_p.

All intermediate results are bound in let-expressions, but (for readability) some of these can later be unfolded, e.g., when the bound variable is used only once.

$$
\begin{array}{lll}
M_g, M_{\overline{g}} & : \mathit{Relation}^* \to \mathit{Vars} \to \mathit{Relation}^* & \\
M_g[\varepsilon]V & = \mathit{true} & \\
M_g[r, S]V & = r \;\&\&\; M_g[S]V & \text{if the variables in } r \text{ are contained in } V \\
M_g[r, S]V & = M_g[S]V & \text{if the variables in } r \text{ are not all contained in } V \\
 & & \\
M_{\overline{g}}[\varepsilon]V & = \varepsilon & \\
M_{\overline{g}}[r, S]V & = r, M_{\overline{g}}[S]V & \text{if the variables in } r \text{ are not all contained in } V \\
M_{\overline{g}}[r, S]V & = M_{\overline{g}}[S]V & \text{if the variables in } r \text{ are contained in } V
\end{array}
$$

Fig. 5. From relations to guard and relations

Combining all of the above, we get that if we have a function `f`, a list of relations S, a set of input variables V_{in} and a set of output variables V_{out}, we construct an equation `f` $p \mid g = e$, where

$$
\begin{array}{ll}
(p, S_1) & = M_p[S]V_{in} \\
V & = \text{the set of variables in } p \\
g & = M_g[S_1]V \\
S_2 & = M_{\overline{g}}[S_1]V \\
e & = M_e[S_2]V
\end{array}
$$

Example. If we apply the translation function to the sequences found in section 4.3, we get the equations shown below.

```
i2p' (0,nil) | true = 0
i2p' (n,x1) | n>0 =
  let x4 = sub1 n in
    let (x2,x3) = insert' (n,x1) in
      let x5 = i2p' (x4,x2) in
         let i = MLA (x5,n,x3) in i

insert' (n,n:xs) | true = (xs, 0)
insert' (n,x:x3) | x/=n =
  let (xs,x4) = insert' (n,x3) in
    let x1 = (x:xs) in
      let i = add1 x4 in
        let true = i>0 in (x1, i)
```

We can unfold some of the let-expressions, replace prefix operators with infix operators and eliminate the redundant test to get a more readable result as shown here:

```
i2p' (0,nil) | true = 0
i2p' (n,x1) | n>0 =
    let (x2,x3) = insert' (n,x1) in i2p' (n-1,x2) * n + x3
```

$$
\begin{array}{ll}
x, y, x_i & \in \textbf{vid} = \textit{variable identifiers} \\
c & \in \textbf{cid} = \textit{constructor identifiers} \\
f & \in \textbf{fid} = \textit{function identifiers} \\
o & \in \textbf{oid} = \textit{operator identifiers} \\
o & \in \textbf{pid} = \textit{predicate identifiers}
\end{array}
$$

$$
\begin{array}{ll}
M_e & : \mathit{Relation}^* \to \textbf{vid}^* \to \mathit{Exp} \\
M_e[S]V & = V_{out} \quad \text{if } V_{out} \subseteq V \\
M_e[x = k,\, S]V & = \texttt{let}\ x = k\ \texttt{in}\ M_e[S](\{x\} \cup V) \\
M_e[x = y,\, S]V & = \texttt{let}\ x = y\ \texttt{in}\ M_e[S](\{x\} \cup V) \quad \text{if } y \in V \\
M_e[x = y,\, S]V & = \texttt{let}\ y = x\ \texttt{in}\ M_e[S](\{y\} \cup V) \quad \text{if } x \in V \\
M_e[x = (y_1, \ldots, y_n),\, S]V & = \texttt{let}\ x = (y_1, \ldots, y_n)\ \texttt{in}\ M_e[S](\{x\} \cup V) \\
 & \qquad \text{if } \{y_1, \ldots, y_n\} \subseteq V \\
M_e[x = (y_1, \ldots, y_n),\, S]V & = \texttt{let}\ (y_1, \ldots, y_n) = x\ \texttt{in}\ M_e[S](\{y_1, \ldots, y_n\} \cup V) \\
 & \qquad \text{if } x \in V \\
M_e[x = c\ Y,\, S]V & = \texttt{let}\ x = c\ Y\ \texttt{in}\ M_e[S](\{x\} \cup V) \quad \text{if } Y \subseteq V \\
M_e[x = c\ Y,\, S]V & = \texttt{let}\ c\ Y = x\ \texttt{in}\ M_e[S](Y \cup V) \quad \text{if } x \in V \\
M_e[Y_2 = o\ Y_1,\, S]V & = \texttt{let}\ Z_2 = o'\ Z_1\ \texttt{in}\ M_e[S](Z_2 \cup V) \\
 & \qquad \text{where } Z_1 \subseteq (Y_1 \cup Y_2) \cap V,\ \ Z_2 = (Y_1 \cup Y_2) \setminus Z_1 \\
 & \qquad \text{and } o' \text{ is a semi-inverse of } o \text{ with inputs corresponding to } Z_1 \\
M_e[f\ Y,\, S]V & = \texttt{let}\ Z_2 = f'\ Z_1\ \texttt{in}\ M_e[S](Z_2 \cup V) \\
 & \qquad \text{where } Z_1 \subseteq Y \cap V,\ \ Z_2 = Y \setminus Z_1 \\
 & \qquad \text{and } f' \text{ is a valid semi-inverse of } f \text{ with inputs corresponding to } Z_1 \\
M_e[p\ (y_1, \ldots, y_n),\, S]V & = \texttt{let}\ \texttt{true} = p\ (y_1, \ldots, y_n)\ \texttt{in}\ M_e[S]V
\end{array}
$$

Fig. 6. From relations to expression

```
insert' (n,n:xs) | true = (xs, 0)
insert' (n,x:x3) | x/=n =
  let (xs,x4) = insert' (n,x3) in (x:xs, x4+1)
```

4.5 Joining Equations

The language demands that the equations of a function must have disjoint domains through their patterns and guards. There is no guarantee that this will be true of the semi-inverted equations, even if it was true for the original equations.

If two or more equations of a semi-inverted function have overlapping domains, we can do several things:

- See if we can refine the guards by considering the domains of operators: If an operator is applied to variables in the pattern and the operation is not defined on all possible values, we can add a guard that restrict the variables to the defined domain of the operator.
- If a predicate is part of the expression, it and all required let-bindings of variables used in the predicate can be copied into the guard.
- Invalidate the semi-inverse.

The first of these options is preferable, but it only rarely works.

The second option may cause duplicated work, so we will use this only if the duplicated code doesn't involve function calls.

Invalidating the semi-inverse is simple (we already have a mechanism for that) and can always be done, but requires that we rerun part of the transformation, see below, so we use it as a last resort.

Note that the presence of overlapping domains may be undecidable if the guards are nontrivial. Hence, we may sometimes reject equations where there is no overlap because we are unable to see this. Alternatively, we can restrict guards to a form that makes disjointedness decidable. This would require the extraction of guards from relations to only extract guards in this form.

4.6 Adding Extra Arguments to Make Semi-inverses Valid

In the simplest case, the user specifies a number of desired semi-inverses, we desequentialise, resequentialise and find that the specified semi-inverses are valid, possibly in the process adding a few more required semi-inverses to the set and validating these. We then translate back, and if the domains of the equations are disjoint, we are done.

If, however, resequentialisation or joining of equations find that a desired semi-invariant is not valid, we add it to the list of invalid semi-inverses. We then rerun resequentialisation for all semi-inverses in the desired list and do the subsequent back-translation and joining of equations, repeating all of this as needed. If we find that we are still unable to define the top-level user-specified semi-inverses, we need to have extra inputs added to these in order to make them valid.

In the extreme, we can move all remaining parts of the input of the original function from the output of the semi-inverse to its input, in which case the semi-inverse will surely be valid: All intermediate variables can be computed from the original input, and the patterns and guards used to distinguish the original equations will also distinguish the equations of the semi-inverse.

But we will usually want to add as little of the original input as possible as extra input to the semi-inverse. Fortunately, when we discover that a semi-inverse in invalid, we will often have information that is useful in deciding which part of the input to add:

- If resequentialisation fails to sequentialise all relations in the relation set, we can look at the variables not in K. If one of these represents a part of the original input and is enough to make other variables defined (possibly through an as yet not validated semi-inverse function), this is an obvious choice for additional input to the semi-inverse.
- If we find that the equations for a semi-inverse do not have disjoint domains, a part of the input that would make them disjoint is an obvious additional parameter. This may be a variable that is used in the guards of the original equations or a variable that correspond to a part of the input where patterns made the original equations disjoint.

Example. The equations of `i2p'` are disjoint as `n` is 0 in the first equation and required to be greater than 0 in the second. The equations of `insert'` are disjoint as the first

element of the list is equal to `n` in the first equation and required to be different from `n` in the second. Note that this exploits the "unneeded" guard `x`$\neq$`n` from the original definition of `insert`. The guard represents part of a non-trivial invariant of `i2p`: The permuted lists do not have repeated elements.

The relations between the original and semi-inverted functions are:

$$\begin{array}{ll} \texttt{i2p}(n,i) = p & \Leftrightarrow \texttt{i2p'}(n,p) = i \\ \texttt{insert}(xs,n,i) = ys & \Leftrightarrow \texttt{insert'}(n,ys) = (xs,i) \end{array}$$

5 Conclusion

The main new contribution of this paper is working on semi-inverses, where most previous work has focused on true inverses. This is enabled by a number of smaller contributions: Using a relational intermediate form with no *a priori* order of evaluation and a resequentialisation analysis to discover a valid order of evaluation for a semi-inverse and determining if one such exist. Additionally, the method, when it fails, can provide guidance to the user for finding extra information that can lead to useful semi-inverses as alternatives to those initially specified.

Our definition of extractions is also, we believe, new. It is used similarly to the way *projections* [10,9] have been used for describing incomplete input in partial evaluation, but it is a better fit to strict languages as these don't normally work on the partially undefined values that projections yield.

Due to space constraints, the example did not show backtracking on invalidated semi-inverses. If true inversion of the program from figure 3 is attempted, the method will find that the desired semi-inverse of `insert` is not valid (due to overlapping equations), so backtracking is made to the top level, where extra inputs are requested. Adding `n` as the extra input gives the semi-inverse shown above.

The example required an invariant of the input to the semi-inverse to be specified in order to work. This invariant was a property of the output of the original program, but became a property of the input to the semi-inverse. As such it is to be expected that tests are required in the semi-inverse to verify that the input actually has this property.

Ideally, a semi-inversion method should discover such invariants, but it is unrealistic to expect it to always do so, as discovery of nontrivial invariants is uncomputable. As a consequence, it may sometimes be necessary to provide such invariants as extra information to the semi-inversion process. In the example, the invariant concerns one of the original function parameters, so adding a guard to the original program was easy. If the invariant concerns variables that do not occur in a pattern, it may be necessary to add a "partial identity function", i.e., a function that returns its input as result but is only defined on values that obey the invariant. Turchin [14] call such partial identity functions "filters". Adding such redundant guards or filters is conceptually similar to using *binding time improvements* [6,3,1] to improve the result of partial evaluation.

5.1 Related Work

Prolog [13] and similar languages have long had the ability to run programs backward or partly backward, so each program is its own inverse and semi-inverse. Prolog relies

on backtracking and may fail to terminate when not run with sufficient variables instantiated. We avoid both backtracking and nontermination by requiring extra inputs to be added when there isn't sufficient information to uniquely choose an equation, but in doing so may fail to produce a semi-inverse program in some cases where Prolog is able to find solutions with a similar subset of inputs specified. The relational form used as an intermediate step in our transformation also has a resemblance to Prolog. Inversion of relational programs is investigated in [8] and [11], but the relational form is quite different, as the programs are variable-free many-to-many relations constructed from relational combinators, which makes inversion relatively simple.

Other work on inverting programs [4,2,5] also avoid backtracking by requiring deterministic choice in the inverted programs. These methods can, however, only make true inverses and simply give up if they can't find a non-backtracking inverse program. In the main, these methods work by direct inversion of all data and control flow, so they are not easily extended to semi-inversion, where data and control flow is partly forwards and partly backwards.

We have been able to find one work, [12], that mentions the possibility of transforming a program to a semi-inverse (there called a partial inverse) and shows an example of semi-inverting multiplication on unary numbers to division. The method can introduce backtracking (and does so in the example). Our method can semi-inverse multiplication of unary numbers to non-backtracking divison.[1]

5.2 Future Work

A prototype implementation of an early version of the semi-inversion method presented here has been implemented as a student project. It is able to do the example semi-inversion shown in this paper, but lacks backtracking on invalid semi-inverts.

Some of the transformation steps used in this paper, such as the order in which variables are defined in the semi-inverted programs, are under-specified. Good heuristics for these steps need to be found.

The method for obtaining disjoint equations after semi-inversion is fairly crude, and it is plausible that the more advanced methods used in [4] could be applied. However, semi-inversion can often do with less powerful methods than complete inversion, partly because some of the original inputs may be retained so guards of the original program can be reused and partly because one might be able to find another semi-inverse to use instead of the one that failed, an option not available for full inversion.

Larger, more realistic, examples need to be examined, such as deriving a decryption function from an encryption function as mentioned in the introduction. We can not expect the method to be amenable to encryption methods based on prime numbers, as their invertibility often rely on nontrivial number theoretic properties, but it is possible that the method can work with cyphers that work by manipulating bitstrings. This needs to be determined, however.

[1] This requires constraining one argument of the multiplier to be non-zero, as division by a possibly zero value will cause overlapping equations in the semi-inverse.

References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10. ACM Press, 1992.
2. Edsger W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, LNCS 69, pages 54–57. Springer-Verlag, 1978.
3. Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.
4. Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming. Proceedings*, LNCS 2998, pages 291–306. Springer-Verlag, 2004.
5. David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
6. Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In [7], pages 83–100, 1991.
7. Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*. Springer-Verlag, August 1991.
8. Ed Knapen. *Relational programming, program inversion and the derivation of parsing algorithms*. Master's thesis, Eindhoven University of Technology, 1993.
9. J. Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.
10. John Launchbury. Projections for specialisation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.
11. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In Dexter Kozen, editor, *Mathematics of Program Construction. Proceedings*, LNCS 3125, pages 289–313. Springer-Verlag, 2004.
12. A. Y. Romanenko. The generation of inverse functions in Refal. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 427–444. North-Holland, 1988.
13. Leon Sterling and Ehud Shapiro. *The Art of Prolog. Second Edition*. MIT Press, Cambridge, Massachusetts, 1994.
14. Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, LNCS 85, pages 645–657. Springer-Verlag, 1980.

A Generative Programming Approach to Interactive Information Retrieval: Insights and Experiences

Saverio Perugini[1] and Naren Ramakrishnan[2]

[1] Department of Computer Science,
University of Dayton, OH 45469-2160 USA
saverio@udayton.edu
http://homepages.udayton.edu/~perugisa
[2] Department of Computer Science,
Virginia Tech, VA 24061-0106 USA
naren@cs.vt.edu
http://people.cs.vt.edu/~ramakris
http://oot.cps.udayton.edu

Abstract. We describe the application of generative programming to a problem in interactive information retrieval. The particular interactive information retrieval problem we study is the support for 'out of turn interaction' with a website – how a user can communicate input to a website when the site is not soliciting such information on the current page, but will do so on a subsequent page. Our solution approach makes generous use of program transformations (partial evaluation, currying, and slicing) to delay the site's current solicitation for input until after the user's out-of-turn input is processed. We illustrate how studying out-of-turn interaction through a generative lens leads to several valuable insights: (i) the concept of a web dialog, (ii) an improved understanding of web taxonomies, and (iii) new web interaction techniques and interfaces. These notions allow us to cast the design of interactive (and responsive) websites in terms of the underlying dialog structure and, further, suggest a simple implementation strategy with a clean separation of concerns. We also highlight new research directions opened up by the generative programming approach to interactive information retrieval such as the idea of web interaction axioms.

1 Introduction

Generative programming has been typically been applied to problems at the crossroads of programming languages and software engineering such as modularizing cross-cutting concerns, synthesizing programs from formal specifications, and automatically generating program documentation. We describe here a novel application of generative programming to a problem in interactive information retrieval [1].

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 205–220, 2005.

1.1 Motivating Example

Everybody has experienced the frustration in interacting with automated information systems where the system does not let the user progress through the dialog without answering a currently posed question. For instance,

1 **System:** Welcome to the automated flight reservation system.
2 **System:** Please say the date on which you wish to travel.
3 **Sallie:** I'd like to fly from New York to Brussels next week.
4 **System:** Sorry, I didn't understand. Please specify a date.
5 **Sallie:** If you can tell me available dates, I can choose.
6 **System:** Please say the date on which you wish to travel.
7 **Sallie:** [Hangs up]

The mental mismatch between Sallie's conception of the task and the system's design is manifest in the above interaction. The system is expecting a date in Line 3 whereas Sallie specifies her choice of source and destination cities. Even though this information is going to be relevant further into the interaction, the system insists on specifying date before going further. Similar inconveniences happen while interacting with websites. A site presents hardwired choices of hyperlinks to pursue and even though the user's input is pertinent and probably solicited deeper in the site, there is no way for the user to circumvent the given navigation structure.

Our solution to the above situations, where the user cannot answer a currently posed question, but does have some other information pertinent to the task at hand, is to provide a capability for *out-of-turn* interaction. For instance, we would provide a capability for the user to speak something into the browser, and in this way supply out-of-turn input. Such 'unsolicited reporting' has been recognized [2] as a simple form of *mixed-initiative interaction*, a dialog management strategy where the two participants take turns exchanging the initiative. Using out-of-turn interaction, the user is empowered to complete an information-finding task in the manner that best suits her conception. Moreover, we show that out-of-turn interaction, irrespective of when it happens, can be supported uniformly by a generative programming approach. A website that currently provides a hardwired choice of completion options can be automatically converted into one that supports out-of-turn interaction!

The idea behind our approach is quite simple: we liken out-of-turn interaction to non-sequential evaluation of a computer program, e.g., partial evaluation. We model an information seeking interaction as a computer program so that user inputs correspond to values for program variables (ref. Fig. 1, left). When the user provides input in the order in which they are requested, we are sequentially evaluating the program, i.e., interpreting it. In a web hierarchy, this would correspond to plain browsing. When the user provides out-of-turn input, we jump ahead to nested program segments that involve that input and simplify them out via partial evaluation. By employing sequences of such interpretations and partial evaluations, we can support complex interactions that involve both responsive as well as out-of-turn inputs.

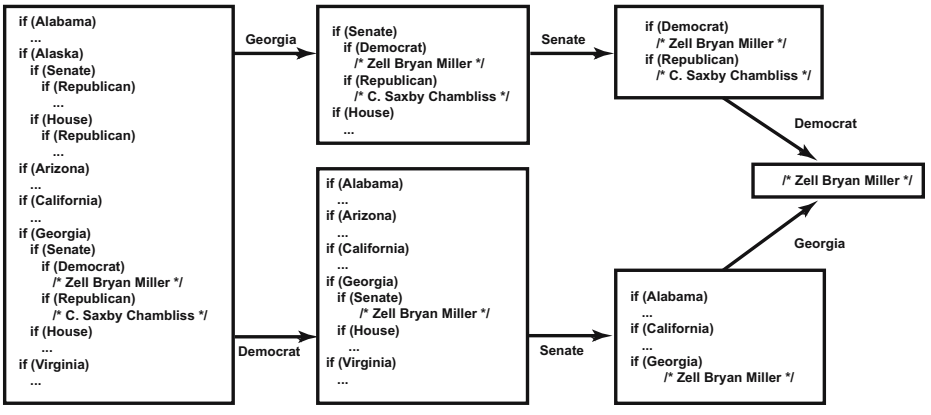


Fig. 1. Staging web interactions using program transformations. The top series of transformations mimic an in-turn (i.e., browsing) interaction sequence with the user specifying (Georgia: Senate: Democrat), in that order (ref. Fig. 2, left). The bottom series of transformations correspond to an out-of-turn interaction sequence where the user specifies (Democrat: Senator: Georgia), in that order (ref. Fig. 2, right). Notice that we can stage both interaction sequences here with the *same* program transformation! *All* programs shown here are *partial evaluations* of the starting program (left).

1.2 Implementing Out-of-Turn Interaction Generatively

Now, since a given program can be transformed in numerous ways, the designer need only write the program in one way but the use of program transformations enables us to realize all possible interaction sequences. Further, since partial evaluation subsumes interpretation, there is no need to distinguish between an in-turn or out-of-turn input. This enables a simple implementation strategy with a clean separation of concerns. An input, supplied using any of a variety of user interfaces, is communicated to a server where it is used to partially evaluate a program. The resulting program is rendered as a website and presented back to the user. Fig. 1 depicts these ideas using Project Vote Smart (PVS; `www.vote-smart.org`), a website which indexes the webpages of the US Congressional Officials and asks a user to make a selection for state, branch of Congress, and party, *in that order*, to access an official's page. Fig. 2, left and right, illustrates how the sequences staged by the top and bottom series of program transformations in Fig. 1, respectively, might be rendered on the web.

Our generative approach makes enabling out-of-turn interaction in an existing website a fairly mechanical process. The approach requires four components: a representation, transformer, out-of-turn interaction interface, and generator. A representation of the site's hyperlink structure, such as that in Fig. 1 (left), can be extracted from the original site and stored in its server from which it will be transformed to stage user interaction. Such a representation can be easily gen-

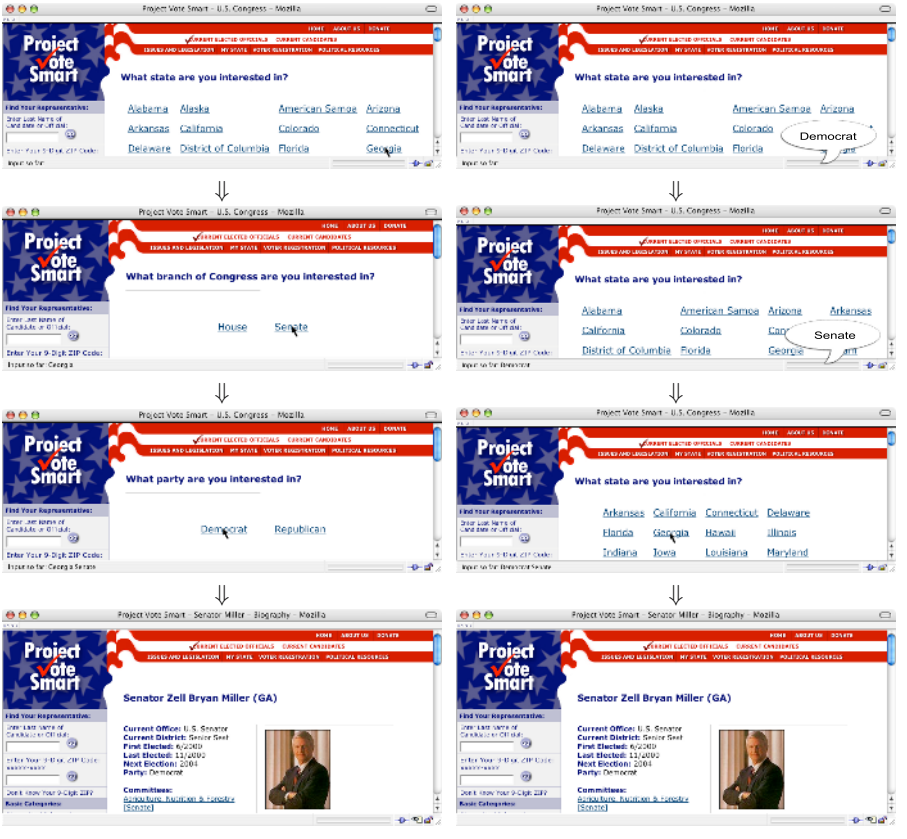


Fig. 2. Retrieving the webpage of the Senator Miller in PVS. (left) In-turn interaction sequence: the user specifies values for relevant politician attributes by progressively clicking on the presented hyperlinks (Georgia: Senate: Democrat), in that order. (right) Out-of-turn interaction sequence: user specifies (Democrat: Senate: Georgia), in that order, using out-of-turn interaction via voice.

erated from a depth-first traversal of the site using either an off-the-shelf web crawler or web scripting languages (e.g., Python) to build a customized bot. The out-of-turn interaction interface captures and communicates the user's out-of-turn input (i.e., a string) to a web server. We have built an out-of-turn interaction toolbar interface, called *Extempore*, using XUL (XML User Interface Language). *Extempore* is embedded into a traditional web browser as a plug-in [3]. We also have implemented a voice interface using SALT (Speech Application Language Tags) to capture out-of-turn speech utterances (illustrated in Fig. 2, right). A server-side program or web service transforms the representation given a set of (in-turn or out-of-turn) user input terms (communicated via a hyperlink click or the out-of-turn interface). Lastly, the generator produces a webpage containing

hyperlink labels corresponding to the variables at the topmost level of nesting in the representation.

Initially we simply run the representation through the generator to create the top-level page of the site. The resulting webpage is aesthetically identical to the original site's homepage except that each hyperlink now represents a request to invoke the transformation operator on the representation wrt the hyperlink's label rather than a request for a static page. Once this initialization is complete, a communicate-transform-generate loop responds to each user interaction (hyperlink click or out-of-turn input). The user *communicate*s an input using the available interaction interfaces (hyperlinks or the out-of-turn interface). This input is used to *transform* the representation. Then the *generator* produces the resulting page from the new representation. Notice that there is no longer a need to store and retrieve any static pages. The current page is always generated dynamically from the the topmost level of nesting of the mostly recently transformed representation. The malleability of the representation stages the interaction and provides the illusion of a website containing a hardwired hierarchy of hyperlinks. When the representation reduces to the modeling of a single `page`, the user is redirected to that webpage. Note also that the representation is the only site-specific component in our framework.

1.3 Outline

Our research began by exploring the use of partial evaluation to transform representations of websites, for realizing out-of-turn interaction [3]. One of the first lessons we learned was that program transformers have a novel use (hitherto unexplored) as *stagers*, i.e., devices to mediate and manage interaction between two entities. In this sense, a partial evaluator is not just a pre-processor before a compiler, it is an active participant in an interaction loop between the human and the information system. This led us to investigate other program transformers (e.g., currying) and study their staging properties. We are now able to develop complex (web) dialogs as compositions of these primitive stagers [4], especially those involving mixed-initiative interaction. This paper begins by presenting these notions. Studying dialog simplification in this context then leads us to an improved understanding of web taxonomies. Next, we present new web interaction techniques and associated interfaces that exploit properties of taxonomies and which allow us to support complex dialogs. We conclude by introducing the idea of web interaction axioms and their potential role in interactive systems.

2 Related Research

Concepts from generative programming (partial evaluation [5], currying [6], program slicing [7], and continuations [8]), have been traditionally studied and employed in systems such as compilers and debuggers. While there are established and effective models for classical information retrieval (e.g., vector-space [9]), models for solutions to interactive information retrieval (IR) problems are in

their infancy. Generative programming suggests helpful metaphors for developing such models. However, generative programming is under-explored in the interactive IR community.

Belkin *et al.* introduced the idea of an 'interaction script' [10] which can be thought of as a program for interaction, though expressed in English rather than program codes and is only intended to be sequentially evaluated. Slicing [11], and source-to-source rewrite rules [12] have been used to restructure web applications. Graunke *et al.* [13] describe an approach to automatically restructure batch programs for interactive use on the World Wide Web. An important issue addressed is maintaining state across web interactions which use the stateless HTTP protocol. Their approach involves first-class continuations from programming languages [8], e.g., via the `call/cc` (call-with-current-continuation) facility provided by Scheme. Since first-class continuations can be saved and resumed, they are an ideal construct for saving and restoring state between user interactions over the web. Using a similar idea based on continuations, Queinnec [14] developed a model for a web server intended to address state maintenance problems caused by connections terminated prematurely, pressing the 'back button,' and window cloning. Lastly, Quan *et al.* [15] explore the idea of using continuations and currying to postpone, save, and resume interactions with intrusive dialog boxes, including partially-filled ones, in traditional application software, such word processors and e-mail clients. The common theme of these efforts, including our research, is the appeal to concepts from programming languages to achieve a rich and expressive form of a human-computer interaction. Our work differs from all these efforts in its focus on out-of-turn interactions (and dialogs involving them). We believe that the generative programming approach presented here suggests useful metaphors for developing interactive information systems and also lends insights into representations for complex dialogs.

3 Web Dialogs

In studying the nature of dialogs supported in our framework, we started to think of a program transformation in terms of the number of interaction sequences it is capable of staging, which we refer to as the transformer's *interaction paradigm*. For example, our use of partial evaluation in PVS is capable of staging interaction sequences representing all permutations of state, branch, party or, in other words, 3,240 ($= 540 \times 3!$) sequences. PVS has 540 paths from its root to each leaf corresponding to the 540 members of the US Congress. In general, a partial evaluator can support $m \times n!$ sequences assuming that each of the m paths through the site has a consistent dialog length of n. Studying program transformers via the number of sequences they support revealed that partial evaluation could stage $m \times n!$ sequences in a given site, but no less. In other words, while partial evaluation can support all orders of supplying inputs, it cannot *enforce* an order. This property prevented us from, e.g., staging dialogs involving state, branch, party, where the party information *must* be supplied second. This 'all or nothing' nature of partial evaluation arises because, without factoring a program

into multiple units, there is no way to prevent expressions containing particular remaining variables from being simplified by a partial evaluator.

This compelled us to develop a dialog notation, where the specific inputs (e.g., Alabama, Senate, Democrat) are abstracted into their categories (e.g., state, branch, party) and used as dialog slots in the context of a program transformer. The syntax of our notation uses the abbreviation of a program transformation (e.g., PE for partial evaluator) over a sequence of such slots. For example, we represent a dialog where values for state, branch, party may be communicated in any order as $\frac{PE}{\text{state branch party}}$. Such an expression succinctly compacts a set of interaction sequences; contrast this with the programs in Fig. 1 where the inputs are woven into the dialog structure. Next we incorporated additional program transformers in order to achieve a finer level of control over the number of interaction sequences stagable. For example, a *currier* stages a different number of sequences than a partial evaluator, i.e., $\frac{C}{\text{state branch party}} \neq \frac{PE}{\text{state branch party}}$. The former only permits the user to supply a *prefix* of the remaining dialog options at any point in the interaction, whereas the latter makes no such restriction. Next we can begin to nest program transformers on top of each other to create complex dialogs (i.e., dialogs composed of smaller dialogs, or subdialogs). For instance, $\frac{PE}{\frac{PE}{a\ b}\ \frac{PE}{c\ d}}$ precludes sequences such as $\prec c\ a\ b\ d \succ$. This notation provided a concise way to specify complex dialogs (i.e., much more compact than enumerating each individual interaction sequence to be supported). In addition, using a small set of reduction rules [4], which indicate how any dialog (described in this notation) should be simplified each time a user supplies an input, we are able to stage a variety of web dialogs in our generative framework.

4 An Improved Understanding of Web Taxonomies

Dialog simplification in the staging transformations framework can be viewed as pruning branches of a website based on user input. This led us to investigate functional dependencies in information hierarchies, a concept which implicitly captures what should remain and what should be pruned out when a user supplies input.

4.1 Functional Dependencies on the Web

Intuitively, an FD of the form $x \rightarrow y$ exists in a website when *all* paths (from the root to a leaf) through the site containing x also contain y. Notice that $x \rightarrow y$ does not necessarily mean that $y \rightarrow x$. In the generative approach, since representations change dynamically after every interaction, the set of FDs satisfied by a site also changes dynamically. As some paths through the site are pruned out by partial evaluation, new dependencies emerge. For instance, communicating 'Senate' to PVS out-of-turn at the top level, causes the 'Virginia $\rightarrow$ Republican' FD to emerge. This FD is not present in the original site because not all politicians in Virginia are Republicans (but the Senators are). In the untransformed PVS site, there are 129 FDs!

The set of FDs a site satisfies can be mined from a relational representation of the paths through the site, where each tuple in the relation corresponds to a path, using standard algorithms from association rule mining [16]. We added a mining component to our generative framework to discover FDs. Since FDs must be re-computed at every step, we would like to optimize the process of mining them. While non-intuitive, it happens to be helpful to postpone computing the *current* set of FDs until *after* the user's input has been processed. In other words, rather than discovering that the 'Virginia → Republican' FD (among many others) exists after the user supplies 'Senate', only compute that FD if and when the knowledge of its existence is required (e.g., if the users supplies 'Virginia' next!). This lazy discovery not only prevents us from computing FDs that are irrelevant to the task at hand, but also results in a more efficient and simple mining procedure. For example, if the user does supply 'Virginia' next, we need only observe that the only party option remaining is 'Republican' to conclude that the 'Virginia → Republican' FD held *before* the input 'Virginia' was processed – a procedure more efficient than examining all pairs of terms (which co-occur) *a priori* as candidates for potential FDs.

FDs serve multiple uses in the generative framework. First, and most obviously, they suggest a simple strategy to perform *input expansion*. For instance, if a user communicates 'Washington, DC' at the top level of PVS we can safely expand this input to 'Washington, DC House Democrat', without changing the semantics of the request, because the 'Washington, DC → {House, Democrat}' dependency exists. Second, the exploitation of FDs results in cleaner representations, by consolidating series' of nested conditionals without else clauses, thereby relieving the user from having to click through several pages, each with only one link.

Further, we can generalize the notion of FDs to say that an FD of the form $x \rightarrow y$ exists in a website when at least $t\%$ of the paths through the site containing x also contain y. Using FDs of this sort for input expansion makes interactive IR *approximate*. Approximate interactive IR is important as it enables a host of new and compelling queries and suggests novel user interfaces. We shall have more to say about these two items in section 5 when we discuss new interaction techniques. In summary, we have illustrated how our generative approach led to the concept of a web FD and a new way to conduct query expansion on the web which together led to approximate interactive IR on the web.

4.2 Levelwise and Non-levelwise Taxonomies

Thus far, we have focused on out-of-turn inputs in a levelwise taxonomy, where each input (e.g., Virginia) addresses a distinct information category (e.g., state). With very minor modifications, we can extend our approach to work with non-levelwise taxonomies – those where no such organization exists. For instance, consider the web directory in Fig. 3 (left). Notice that in this website, unlike PVS, each level does not correspond to an information category (e.g., state or party). For instance, a hyperlink labeled 'soccer' resides at levels two and three.

To capture input expansion in non-levelwise sites, we use the concept of a *negative* FD: $x \rightarrow \neg\, y$, that holds when *none* of the paths through the site

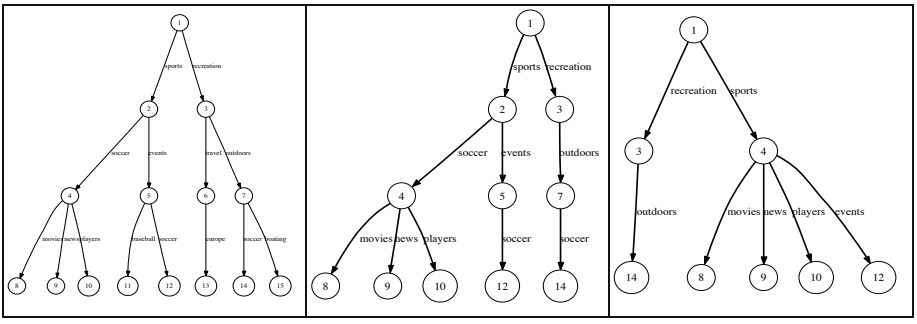


Fig. 3. (left) Hypothetical hierarchical web directory with characteristics similar to those in Yahoo!. (right and center) Customized versions of (left) wrt 'soccer'.

containing x also contain y. Some intuitive negative FDs in PVS are, 'Democrat $\rightarrow$ $\neg$ Republican', 'House $\rightarrow \neg$ Senate', and 'Virginia $\rightarrow \neg$ Ohio'. Notice that any negative FD $x \rightarrow \neg\ y$ implies $y \rightarrow \neg\ x$. When the user communicates 'Senate' out-of-turn, we can partially evaluate wrt to `Senate=true` and `House=false`. The reader will notice that negative FDs in a levelwise site involve only the labels of hyperlinks at the same level. However this is not the case in non-levelwise sites, thus providing the motivation for negative FDs. For instance, the following are some negative FDs that hold in the site illustrated in Fig. 3 (left): 'sports $\rightarrow \neg$ {boating, europe, outdoors, recreation, travel}' and 'soccer $\rightarrow \neg$ {baseball, boating, europe, travel}'. Thus, when a user says 'soccer' out-of-turn in Fig. 3 (left), we can partially evaluate the program in Fig. 4 (left) wrt `soccer=true` and `baseball`, `boating`, `europe`, and `travel` set to `false` which yields the program in Fig. 4 (right) which models Fig. 3 (right).

Notice that there are no salient structural properties of websites that ultimately influence the characteristics of their FDs (e.g., only FDs, only negative FDs, or a mixture of the two). The type of each FD currently satisfied by a site is dependent only on the current relationships between the co-occurrence of terms (labeling hyperlinks) on paths through the site (term y co-occurs on all/no paths containing term x). There are ways alternate to path containment in which terms can co-occur (e.g., two terms co-occur if the distinct paths which contain them lead to the same leaf vertex, i.e., the terms are used to index the same page) and using these criteria lead to additional types of FDs [17]. Further, notice the t threshold in our generalized notion of an FD defines the type of FD (t=0 specifies a negative FD and t=100 indicates an FD) for a particular co-occurrence criteria.

The distinction between levelwise and non-levelwise sites encouraged us to study the properties of web hierarchies to discern which program transformations are applicable to certain types of hierarchies. This analysis led to our development of a partial order of graph-theoretic classes of hierarchical hypermedia [17] which formally characterize websites by the relationships among the

```
 1 if (sports)
 2   if (soccer)
 3     if (movies)
 4       page = 8;
 5     if (news)
 6       page = 9;
 7     if (players)
 8       page = 10;
 9   if (events)
10     if (baseball);
11       page = 11;
12     if (soccer);
13       page = 12;
14 if (recreation)
15   if (travel)
16     if (europe)
17       page = 13;
18   if (outdoors)
19     if (soccer)
20       page = 14;
21     if (boating)
22       page = 15;
```

```
if (sports)
  if (soccer)
    if (movies)
      page = 8;
    if (news)
      page = 9;
    if (players)
      page = 10;
  if (events)

    if (soccer)
      page = 12;
if (recreation)

  if (outdoors)
    if (soccer)
      page = 14;
```

```
if (sports)

    if (movies)
      page = 8;
    if (news)
      page = 9;
    if (players)
      page = 10;
  if (events)

      page = 12;
if (recreation)

  if (outdoors)

      page = 14;
```

Fig. 4. (left) Programmatic representation of the website modeled by Fig. 3 (left). (center) Representation of the site in Fig. 3 (center) resulting from slicing (left) wrt `music`. (right) Representation of site in Fig. 3 (right) resulting from partially evaluating (center) wrt only `soccer=true`.

terms modeling the site's hyperlink labels. This ordering helps connect our work to the hypermedia and interactive visualization community who have developed a similar taxonomy [18]. We refer the reader to [17] for a discussion of the formal details of the classes, detecting them, and proofs of their properties.

4.3 A General Program Transformation: Program Slicing

Even though we were able to generalize the support for out-of-turn interaction to non-levelwise sites, we still wanted to develop a general purpose program transformation technique than simply applying a combination of FDs and partial evaluation. One reason for this is that often only a subset of the terms in the consequent of a negative FD employed are necessary for partial evaluation. For example, partially evaluating the program in Fig. 4 (left) wrt `sports=true` and `recreation=false` results in the same program as would a partial evaluation wrt to `sports=true` and `boating`, `europe`, `outdoors`, `recreation`, and `travel` set to `false`. This encouraged us to study non-semantic-persevering transformations (ref. Table 1), such as program slicing [7], to generalize our approach to different forms of hierarchical hypermedia in a *single* framework. The idea involves slicing a program to retain only those sequences annotated with the user's out-of-turn input [19]. Program

Table 1. Comparison of partial evaluation and program slicing along a syntax- vs. semantic-preserving dichotomy

	Syntax-preserving	**Semantic-preserving**
Partial evaluation	×	√
Program slicing	√	×

slicing [7] is a theoretical operation used to extract statements that affect (or are affected by) the computation of a variable of interest at a point of interest from a program. There are several varieties of slicing; backward and forward slicing are the two most relevant for our purposes. Slicing has been predominately applied to problems in software engineering such as debugging and reverse engineering [7]. However, it has been applied applied to web application development [11]. Our use of it here helps relate it to interactive IR.

When the user says 'soccer' out-of-turn in Fig. 3 (left) we forward slice the program in Fig. 4 (left) wrt the `soccer` variable at all program points. This leads to the (`page` assignment) statements at lines 4, 6, 8, 13, and 20 from which we backward slice the program. The result is a representation of the site containing only paths involving hyperlinks labeled 'soccer' leading to leaf webpages containing information about soccer (ref. Fig. 4, center). Finally, we partially evaluate the program wrt the variable modeling the user's input ('soccer') statically set to `true` thereby removing all expressions involving it, since it has now been supplied. This results in a program modeling the new site (ref. Fig. 4, right) from which an actual site can be recreated. This program transformation technique generalizes out-of-turn interaction to all of the classes of hierarchical hypermedia that we identified.

4.4 A Duality in Uses of Program Slicing

The instructive nature of our use of generative programming suggested that an attempt to compute web FDs via program transformation might reveal more insight. We developed a technique which uses program slicing to mine web FDs from a programmatic representation of a website. We refer the reader to [17] for the details of the program transformation technique and rather focus on its implications here. We use partial evaluation and program slicing as pruning operators. There is a tradeoff between these two program transformations in the context interactive IR. Specifically, one can think of program slicing as a transformation for

1. directly pruning a website (as illustrated above in Fig. 4), *or*
2. extracting information (i.e., FDs) about *what* to prune from a site and then using this information with partial evaluation to conduct the same site pruning as in (1).

Studying this duality reveals that there might be simpler or more effective methods for realizing out-of-turn interaction with instances of specialized classes in our taxonomy, akin to that illustrated in section 4.2 for levelwise sites.

Towards a Taxonomy of Program Transformations for Interactive IR Thus far, we have seen that our generative approach using only partial evaluation works for only levelwise sites, the most specific class in our partial order. In addition, we have illustrated an alternate program transformation technique, based on slicing, to realize out-of-turn interaction in all of the classes of hierarchical websites we identified. This suggests that additional specialized transformation techniques might exist for classes in our lattice between the most general and most specific class. Developing a mapping between classes of hierarchical hypermedia and generative techniques for interacting with them moves us closer to a generative programming model for interactive IR.

This section has described many insights and made several connections. To recap, we have seen that

1. our simplification rules, involving program transformers, led to the idea of a web FD,
2. web FDs led to a new way to conduct query expansion on the web and, ultimately, approximate interactive IR,
3. the application of partial evaluation as a pruning operator led to classes of hierarchical hypermedia,
4. supporting out-of-turn interaction with instances of the classes led to two new generative approaches: one involving a combination of FDs and partial evaluation and another involving program slicing which generalizes out-of-turn interaction to each class,
5. the two new generative approaches and a method to mine FDs with slicing led to a duality in uses of program slicing, and
6. we are optimistic that this duality will lead to a taxonomy of program transformations for interactive IR.

Overall, this generative programming thread resulted in an improved understanding of web taxonomies and new research issues.

4.5 New Research Issues for Web Taxonomies

Many large web taxonomies, such as Yahoo! and the Open Directory Project at `dmoz.org`, are modeled as a DAG (Directed Acyclic Graph), owing to the presence of symbolic links. A symbolic link is a special type of hyperlink which makes a directed connection from a webpage along one path through a website to a page along another path. One obvious use of symbolic links is multiclassification. For example, information about music is classified under both the arts and computers categories in Fig. 3 (left). Rather than classify information under more than one category, a designer might classify it under only one category, but include a symbolic link from one category to another (e.g., from the arts sub-tree to the computers sub-tree, or vice versa) to give the user the illusion that the item is classified in both categories. Representing a website modeled as a DAG using a program is challenging and requires the use of unconditional branches (e.g., `gotos`) or functions to factor common branches. This viewpoint

leads us to associate a symbolic link with a kludge for out-of-turn interaction. We hypothesize that designers include symbolic links to address the fact that users do not have facilities to interact out-of-turn. Testing this hypothesis suggests that we should mine the uses of symbolic links on the web to identify the typical contexts in they are employed. Replacing symbolic links with new interaction techniques which more naturally support users' information-seeking activities relieves the designer from having to anticipate where and how to include symbolic links in a taxonomy to accommodate users with diverse informational goals.

Lastly, notice that we might generalize our definition of an FD even further to capture and expose the various relationships that might exist between the terms that label hyperlinks in websites. For instance, rather than saying that the $x \rightarrow y$ FD holds if $t\%$ of the paths through the site involving x also involve y, we might say the $x \rightarrow y$ FD holds if $M\ (x,\ y) \geqslant t$, where M is a term similarity metric from IR (e.g., cosine or Jaccard's [20]) and t is a threshold. Thus, another research issue that this thread revealed involves experimentation to identify the metrics and threshold values that are appropriate for a given (class of website, information-seeking goal) pair to be supported.

5 New Web Interaction Techniques and Interfaces

5.1 User Interfaces for Approximate Interactive IR

Approximate retrieval in a web taxonomy, introduced above, is important because, by exposing term relationships (similarities), it can help a user assimilate the underlying domain by dependency exploration. It also can reveal hidden aspects of the domain. For example, a user that communicates 'Senate Senior' and observes that it expands to 'Senate Senior Democrat' might conclude that the Senior leadership in the Senate is largely Democratic, an inference difficult to make simply by browsing the site. Such approximate interactive IR suggests that we might expand a query in *real-time* for the user or permit the user to set the expansion threshold (from no expansion to as much expansion as possible) and *dynamically* observe the links that are removed or added as the user, e.g., moves a slider bar UI widget to dynamically adjust the threshold.

5.2 Dialog Continuations

Our footing on the generative landscape suggested investigating addition techniques from programming languages employed in program transformations, such as continuations. We then used continuations to design a new web interaction primitive, *dialog continuations*, intended to address the destructive nature of program transformations on interaction. To support procedural tasks, we must allow for new subdialogs to be dynamically invoked at the behest of the user, who also determines any partial input that might be applicable at that point. To achieve this functionality, we explicitly manipulate dialog continuations, borrowing an important notion from the programming languages literature [8]. A continuation indicates a 'promise to do something' and summarizes the amount

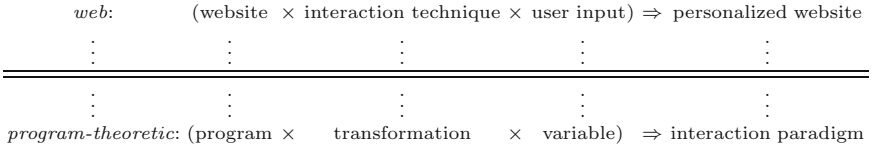


Fig. 5. The connection between the web and program-theoretic domains

of work remaining at a point of execution in a program. While all languages employ continuations internally, only some (e.g., Scheme) allow the user to explicit manipulate and reason about them. To cascade one dialog onto another, we essentially replace the current continuation with a fresh dialog that has been partially evaluated with the user's chosen input, and jump to this dialog. This allows the user to both abandon a given line of conversation (since the requisite information has been obtained) and find themselves in the middle of another line of inquiry.

We have implemented a real-time query expansion interface and various dialog continuation interfaces for users to interactively explore the PVS data. They are available for demonstration from our project webpage at `http://oot.cps.udayton.edu`.

6 Discussion

We have described the insight gained by the virtue of a generative programming lens for interactive IR. The central theme of our generative approach is to pose interactive information retrieval as the application of a program transformations to a programmatic representation of a website based on partial user input (ref. Fig 5, bottom). The creativity in our work (ref. Fig. 5) arises from relating concepts in the web domain (e.g., sites, interactions) to notions in the program-theoretic domain (e.g., programs, transformations). An additional opportunity for creativity involves varying the (program, transformation) pair to achieve a desired form of interaction. The predominate form of interaction discussed in this article is out-of-turn interaction.

The generative techniques showcased here can be implemented with many software tools or programming languages. Our implementation employs PHP for the transformation, generative, and mining components and XUL, SALT, and JavaScript for the user interaction interfaces. Our generative approach has not only been instructive, but also has led to a simple implementation strategy. We implemented the entire framework using less then 1000 lines of code, where the constituent components each occupy approximately equal amounts of code and are cleanly factored. The framework contains no code specific to a targeted website. The representation is the only component which contains site-specific information and is supported as plug-in basis (or simply stored on the original website's server). For further implementation details, including caching and sessioning, see [4]. We have applied these techniques and our framework to several websites: (i) GAMS (Guide to Available Mathematical Software) at

gams.nist.gov, (ii) Project Vote Smart at vote-smart.org, (iii) CITIDEL (Computing and Information Technology Interactive Digital Educational Library) at citidel.org [21], (iv) the Open Directory Project at dmoz.org, and (v) the Online Virginia Tech Timetables of Classes accessible through vt.edu. We refer the reader to [3,4,17] for the details of these case studies.

In summary, the insight exposed by our use of generative programming has (i) helped us connect our work to other communities, (ii) driven the development of new concepts, and (iii) led to new research issues. In particular, we connected our work to the discourse analysis (dialog) and hypermedia/visualization communities. The concept of an FD on the web, while simple, suggested a new way to do query expansion on the web and led to approximate retrieval in large web taxonomies. In addition, borrowing notions like continuations from programming languages led to new, richer, and more conversational, ways of interacting with websites. We intend to continue to investigate the use of generative programming for interactive IR. For example, we might use a language's support for *reflection* to permit the user to dynamically query the program for the choices that are still unspecified as a way of enquiring 'what may I say at this point in the dialog?'

Another compelling line of future work entails investigating what the axiomatic semantics of a program modeling a website imply about the forms of interaction supported by the site. In other words, what are the web interaction analogs to the axiomatic semantics of a program modeling web interaction? An example of a simple (and obvious) interaction axiom which can be inferred from the program is 'no customer shall reach the thank you page without first paying for the items in their shopping cart.' We are optimistic that this work will help us automatically reason about interacting with a site from program axioms. We anticipate such automated reasoning to become more important with the growth of initiatives advocating for more automation, such as the *semantic web* [22] which aims to lift the communication paradigm of the web from human-to-computer to computer-to-computer. For this reason, we believe that the generative approach espoused here is especially timely. The long-term goal of this work is to use the insight detailed here to develop general, but automated, models for the design of interactive (and responsive) websites.

References

1. Marchionini, G.: Information Seeking in Electronic Environments. Cambridge Series on Human-Computer Interaction. Cambridge University Press (1997)
2. Allen, J.F., Guinn, C.I., Horvitz, E.: Mixed-Initiative Interaction. IEEE Intelligent Systems Vol. 14 (1999) pp. 14–23
3. Perugini, S., Ramakrishnan, N.: Personalizing Web Sites with Mixed-Initiative Interaction. IEEE IT Professional Vol. 5 (2003) pp. 9–15
4. Narayan, M., Williams, C., Perugini, S., Ramakrishnan, N.: Staging Transformations for Multimodal Web Interaction Management. In: Proceedings of the Thirteenth ACM International World Wide Web Conference (WWW), New York, NY (2004) pp. 212–223
5. Jones, N.D.: An Introduction to Partial Evaluation. ACM Computing Surveys Vol. 28 (1996) pp. 480–503

6. Ullman, J.: Elements of ML Programming. Second edn. Prentice-Hall (1997)
7. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages Vol. 3 (1995) pp. 121–189
8. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. Second edn. MIT Press (2001)
9. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley (1999)
10. Belkin, N.J., Cool, C., Stein, A., Thiel, U.: Cases, Scripts, and Information-Seeking Strategies: On the Design of Interactive Information Retrieval Systems. Expert Systems with Applications Vol. 9 (1995) pp. 379–395
11. Ricca, F., Tonella, P.: Web Application Slicing. In: Proceedings of the International Conference on Software Maintenance (ICSM), Florence, Italy (2001) pp. 148–157
12. Ricca, F., Tonella, P., Baxter, I.D.: Restructuring Web Applications via Transformation Rules. In: Proceedings of the First International Workshop on Source Code Analysis and Manipulation (SCAM), Florence, Italy (2001) pp. 150–160
13. Graunke, P., Findler, R., Krishnamurthi, S., Felleisen, M.: Automatically Restructuring Programs for the Web. In: Proceedings of the Sixteenth IEEE International Conference on Automated Software Engineering (ASE), San Diego, CA (2001) pp. 211–222
14. Queinnec, C.: The Influence of Browsers on Evaluators or, Continuations to Program Web Servers. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP), Montreal, Canada (2000) pp. 23–33
15. Quan, D., Huynh, D., Karger, D.R., Miller, R.: User Interface Continuations. In: Proceedings of the Sixteenth Annual ACM Symposium on User Interface Software and Technology (UIST), Vancouver, Canada (2003) pp. 145–148
16. Agrawal, R., Imielinski, T., Swami, A.N.: Mining Association Rules between Sets of Items in Large Databases. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Washington, DC (1993) pp. 207–216
17. Perugini, S.: Program Transformations for Information Personalization. Ph.D. dissertation, Department of Computer Science, Virginia Tech (2004) Available in the Virginia Tech ETD collection at http://scholar.lib.vt.edu/theses/available/etd-06252004-162449/. US Copyright Office Registration Number TX 6-040-012.
18. McGuffin, M.J., m. c. schraefel: A Comparison of Hyperstructures: Zzstructures, mSpaces, and Polyarchies. In: Proceedings of the Fifteenth ACM Conference on Hypertext and Hypermedia (HT), Santa Cruz, CA (2004) pp. 153–162
19. Perugini, S., Ramakrishnan, N.: Personalization by Program Slicing. Journal of Object Technology Vol. 4 (2005) pp. 5–11 Special issue: *Sixth ACM GPCE Young Researchers Workshop*, Vancouver, Canada, October 2004.
20. Srehl, A., Ghosh, J., Mooney, R.: Impact of Similiarity Measures on Web-page Clustering. In: Proceedings of the AAAI Worshop of Artificial Intelligence for Web Search, Austin, TX (2000) pp. 58–64
21. Perugini, S., McDevitt, K., Richardson, R., Pérez-Quiñones, M.A., Shen, R., Ramakrishnan, N., Williams, C., Fox, E.A.: Enhancing Usability in CITIDEL: Multimodal, Multilingual, and Interactive Visualization Interfaces. In: Proceedings of the Fourth ACM/IEEE Joint Conference on Digital Libraries (JCDL), Tucson, AZ (2004) pp. 315–324
22. Despeyroux, T.: Practical Semantic Analysis of Web sites and Documents. In: Proceedings of the Thirteenth ACM International World Wide Web Conference (WWW), New York, NY (2004) pp. 685–693

Optimizing Marshalling by Run-Time Program Generation*

Barış Aktemur[1], Joel Jones[2], Samuel Kamin[1], and Lars Clausen[3]

[1] University of Illinois at Urbana-Champaign, USA
{aktemur, kamin}@cs.uiuc.edu
[2] University of Alabama, USA
jones@cs.ua.edu
[3] State's Library, Aarhus, Denmark
lc@statsbiblioteket.dk

Abstract. Saving the internal data of an application in an external form is called *marshalling*. A generic marshaller is difficult to optimize because the format of the data that will be marshalled is unknown at the time the marshaller is implemented. On the other hand, efficient marshallers can be written for specific kinds of data. In this paper we use run-time program generation (RTPG) to produce specialized marshallers. We use Jumbo, a Java compiler supporting programmer-specified RTPG. We show that RTPG is easily employable. Speedups in order of magnitude can be achieved in some cases. We study the case where the data consist of a large number of objects of a single class and the case where there are objects of many classes. In the latter case, "just-in-time" heuristics allow us to limit RTPG costs and gain considerable speedups.

1 Introduction

Marshalling is the term used for saving the internal data of an application in an external form. Once marshalled, objects can be passed to other applications. Java RMI (remote method invocation) and CORBA are examples of systems which marshal data for transmission to remote machines. Another term for marshalling is *serialization*. The reverse process is called *unmarshalling*.

Serialization generally involves writing large amounts of data, and so is often a performance bottleneck. (According to [1], Java serialization accounts for 25–65% of a remote method invocation.) For any particular type of data, it can be heavily optimized. However, optimizing a general-purpose marshaller is difficult because the format of the data to be marshalled is not known at compile-time. Such marshallers are guided by a description of the data that becomes available only at run-time; it is provided either by the client of the marshalling code, or, as in the case we consider here, by the language's reflection mechanism.

Run-time program generation (RTPG) is the use of programs that write other programs at run-time. RTPG can produce efficiencies by taking advantage

* Partial support for this work was received from NSF under grant CCR-0306221.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 221–236, 2005.

of information not known at compile time. Since this is precisely the situation we have just described, marshalling is a natural application for RTPG.

Our research group has developed Jumbo [2,3], a compiler for Java that incorporates an easy-to-use run-time program generation mechanism. Jumbo is distinguished by its implementation strategy [4] and by its consequent generality: virtually any Java program can be generated at run-time. This makes Jumbo particularly easy to learn and use. Thus, we believe it can make the writing of run-time program generators a routine matter [5,6].

In this paper, we demonstrate the practicality of RTPG by applying Jumbo to the problem of optimizing marshalling in Java [7,8,9]. Interestingly, the implementers of the generic marshaller in Sun's standard Java library (`java.io.ObjectOutputStream`) thought its efficiency so important that critical parts of their implementation are written in C++. This takes this class beyond the routine and has the specific drawback that the code cannot be verified. It also renders the implementation unsuitable as a starting point for our experiment. Hence, we start from a pure Java implementation of serialization, provided by Kaffe [10]. We demonstrate that run-time program generation is easy to employ — requiring no more skill than ordinary programming — and can deliver very substantial speedups relative to the pure Java code. (We were not specifically attempting to catch up with Sun's implementation, but we have done so in some cases; we discuss this in section 7.)

On standard marshalling benchmarks — mainly long arrays or lists — a straightforward use of RTPG speeds up the Kaffe implementation by an order of magnitude. Cases involving many classes are more difficult because the cost of the run-time program generation itself cannot be so readily amortized; an adaptive method similar to that used in just-in-time compilers can be employed.

Our contributions in this paper are:

- Demonstrating that, with Jumbo, obtaining generative code based on non-generative code is straightforward.
- Showing significant speedups for marshalling with RTPG.
- Showing that adaptive methods can be applied to reduce the cost of run-time compilation.

The paper is structured as follows. In section 2, we discuss marshalling in Java in more detail and give some ideas about where RTPG might help. Section 3 introduces Jumbo, and section 4 shows how Jumbo can be used to implement the suggestions made in section 2. Section 5 gives performance comparisons between Kaffe and Jumbo for the benchmark cases — large, homogeneous and near-homogeneous collections, and heterogeneous collections. In section 6, we discuss usage of "just in time" program generation to reduce the cost of run-time compilation for heterogeneous data. In section 7, we briefly return to Sun's implementation that uses native code and ask two questions: Can our safe, run-time-generated code compete with Sun's implementation, and can we use RTPG to produce further optimizations of that code? Finally, section 8 reviews related work and section 9 presents our conclusions.

2 Marshalling in `Java`

Java provides a simple API for serialization. A Java programmer doesn't need to write any serialization code, but must simply declare her classes to implement the empty interface `java.io.Serializable`. If a class implements this interface, an instance can be marshalled by passing it to `java.io.ObjectOutputStream`'s (OOS) `writeObject()` method.

Sun provides a specification of serialization [11], and an implementation. However, that implementation uses native methods, written in C/C++, to gain efficiency. Therefore, it is not appropriate for our experiment. An implementation in pure Java[1] is provided by Kaffe [10]; we start our study there.

Throughout this paper we refer to Sun's and Kaffe's implementation as Sun OOS and Kaffe OOS, respectively. The implementation for marshalling which uses RTPG is referred as Jumbo OOS. (Actually there are two versions of Jumbo OOS, but it will be clear from the context to which one we're referring.) When it doesn't matter which OOS we're referring to, we just say OOS.

We now explain Java serialization in detail, to highlight the places that can be optimized by RTPG. The serialization format is roughly as follows: For each object, first write a descriptor for its class and then write the object's fields; primitive fields are written directly, and object fields are written recursively using the same format. To prevent outputting multiple copies of class descriptors or objects – and to avoid infinite loops – each class and object is assigned an id number, or *handle*; every class and object written is stored in a hashtable the first time it is seen, and only its handle is output on subsequent sightings. The pseudo code below outlines Kaffe OOS's `writeObject()` method.

```
writeObject(obj) { //method in Kaffe OOS
  if obj is null {
    writeNull
  } else if obj was already written { // look up the object in the hashtable
    write object handle
  } else if obj is an instance of Class or String {
    write obj according to the specification for that particular case
  } else if the object is an Array {
    for each element i in obj
      writeObject(i) //a recursive call
  } else{ // first write class description of object
    if class of obj was already written
      write class handle
    else
      writeObject(class of obj) //recursive call to serialize class descriptor
    // then write content of object
    if obj is Serializable {
      for each classDescriptor in the class hierarchy of obj
        for each field in the classDescriptor
```

[1] Actually there is one call to a native method, to test whether a class has a static initializer. This test is not available in the reflection API [10].

```
        if the field is primitive
          writePrimitive(field)
        else
          writeObject(field) // recursive call
    } else { throw Exception("obj is not serializable") }
  }
}
```

To summarize, each object is passed through a set of checks: Is the object null? Was it already written to the stream? Is it an array? Was its class descriptor already written? Is it `Serializable`? Finally, for each class descriptor in the inheritance hierarchy of the object, we find the fields of that class. For each field, if it is primitive, we write the actual value directly to the stream. Otherwise, we marshal it by making a recursive call. Note the use of reflection in the above, using class descriptors to discover the fields of the class.

We can optimize the serialization of objects of any class by generating a marshaller specific to it when we first see an instance of that class. After the specialized marshaller is generated, it can be used to serialize subsequent instances. With this alteration, the general marshalling procedure becomes:

```
writeObject(obj) { //method in Jumbo OOS
  if obj is null
    writeNull
  else if obj was already written // look up the object in the hashtable
    write object handle
  else{
    // look for specialized marshaller in the hashtable
    marshaller = getMarshallerFor(class of obj)
    if marshaller is not null // marshaller is found
      marshaller.write(obj)
    else if obj is an instance of Class or String
      // ... as above
      if obj is Serializable {
        // generate specialized marshaller and put it into hashtable
        marshaller = ProgGen.generateMarshallerFor(obj)
        storeMarshaller(marshaller)
        // ... as above
      }
  }
}
```

The bold faced lines in the code above show when to look for a specialized marshaller and when to generate it. As a technical point, the reader will note that a specialized marshaller is not used for marshalling right after it is generated. This is because the generated code writes only the handle of the class, but the class descriptor needs to be written the first time an object of the class is seen.

We now introduce Jumbo. Readers familiar with it can skip the next section. In section 4, we continue the present discussion by showing how to go from the Kaffe code for serialization to the Jumbo code.

3 Jumbo

Jumbo [3,5] is a staged compilation system for Java, allowing run-time program generation. It provides a high degree of programmer control, source level specification, and binary-level operation.

In Jumbo, the programmer specifies code to be generated at run-time by placing it within special quotation brackets: `$<` and `>$`. From the programmer's point of view, these brackets behave very much like ordinary string quotes, but the values represented are of type `Code`, not string, and ordinary string operations cannot be applied. For this to work, the enclosed piece of program cannot be arbitrary, but must be a parsable fragment. The effect of this restriction is that these fragments can be partially compiled, with the result that no external compiler has to be invoked at run-time to generate code. Since many computers have a Java run-time, but no compiler, this is an important practical feature.

A quoted Java fragment can have *holes* that will get filled with `Code` values not known at code-writing time. The syntax for holes is backquote (`‘`) followed by a syntax category, followed by a Java expression of type `Code` in parentheses. Consider

```
public Code infiniteLoopGen(Code body){
  return $< while(true){
              ‘Stmt(body)
            } >$;
}
```

The call `infiniteLoopGen($< if(i == 3) break; i++; >$);` would give us `Code` equivalent to:

```
while(true){
  if(i == 3)
    break;
  i++;
}
```

This code can now be used in a context where `i` is defined.

For expressions of primitive type, there is a second kind of anti-quotation, one which evaluates the expression at program-generation time and then inserts the value into the generated code as a constant. For example, `‘Int(x)` means that x is an `int` variable and its *current* value is to be inserted into the enclosing `Code` (this is called *lifting* [12]).

`Code` is the main class in the Jumbo implementation. A `Code` value represents the *partially* compiled version of a program fragment and is represented as a method. Its argument is the information about the usage context of that fragment that is needed to fully compile the fragment; its result is the virtual machine code thus calculated. Because it is a method, this program fragment is represented as virtual machine code, rather than as source or as a syntax tree.

Detailed information on Jumbo is available in [3,5,2]. Jumbo can be obtained at `loome.cs.uiuc.edu/Jumbo/index.php`.

4 Jumbo Code for Marshalling

In section 2, we showed how to make use of program generation in Jumbo OOS. In this section we discuss how to write the specialized marshaller generator using Jumbo. We have implemented a class, called `ProgGen`, which produces the marshallers. Before we explain `ProgGen`, let's look at the specialized marshaller that would be produced for the following class, representing a linked-list node:

```
public class Node implements Serializable{
  int data;
  Node next;
}
```

Its generated marshaller would be:

```
public class NodeMarshaller implements Marshaller {
  JumboObjectOutputStream oos;
  Field[][] fields;
  int handle;

  public void init(JumboObjectOutputStream oos,
                   Class clazz, int handle) {
    this.oos = oos;
    this.handle = handle;
    ... // initialize fields[][] here - omitted
  }

  public void write(DataOutput stream, Object obj) {
    // Write the OBJECT tag and class handle to the stream
    // These magic numbers are defined in Sun's specification.
    stream.writeByte(115);
    stream.writeByte(113);
    stream.writeInt(handle);
    // write the 'data' field
    stream.writeInt(fields[0][0].getInt(obj));
    // send the 'next' field to Jumbo OOS to have it serialized
    oos.writeObject(fields[0][1].get(obj));
  }
}
```

Note that Jumbo generates byte code – *not* source code. We have given source code for readability: the byte code generated is just what would be produced by a Java compiler if presented with this source code.

When compared with the original OOS, the specialized marshaller is much simpler. The `next` field of `Node` will also be serialized via the specialized marshaller (provided that its run-time type is `Node`). The marshalling process will end when `next` is a null pointer or an already serialized object.

`ProgGen` is obtained by a fairly straightforward massaging of the Kaffe OOS. Basically, `ProgGen` and Kaffe OOS have code in one-to-one correspondence. However, `ProgGen` does not write data into a stream like Kaffe OOS does. Instead, it

forms `Code` which does that job. To illustrate, let's examine the `writeFields()` method of Kaffe OOS. This is the method that actually writes the fields of an object.

```
private void writeFields(Object obj, ObjectStreamClass osc){
  ObjectStreamField[] fields = osc.fields;
  String field_name;
  Class type;
  for (int i = 0; i < fields.length; i++){
    field_name = fields[i].getName();
    type = fields[i].getType();
    if (type == Boolean.TYPE)
      realOutput.writeBoolean(
        getBooleanField(obj, osc.forClass(), field_name));
    else if ... // check for other primitive types
    else        // non-primitive
      writeObject(getObjectField(obj, osc.forClass(),
                  field_name, fields[i].getTypeString ()));
  }
}
```

This method first gets all the fields in a class descriptor. Then, by using each field's descriptor, it fetches the value of the field from the object. This is done in `get`*X*`Field()` of OOS, which uses the `getField()` method below (exception-handling is omitted here for clarity):

```
private int getIntField (Object obj, Class klass, String fname) {
  Field f = getField(klass, fname);
  return f.getInt(obj);
}

Field getField (Class klass, String name) {
  final Field f = klass.getDeclaredField(name);
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      f.setAccessible(true);
      return null;
    }});
  return f;
}
```

This work is done for each object, even if another object of that class was already written. We shouldn't have to find the field specifiers and field types each time. Instead we can generate code with these values built in:

```
private Code writeFields(ObjectStreamClass desc, int hier) {
  ObjectStreamField[] fieldDecls = desc.fields;
  Code c = $< ; >$;
  for (int i = 0; i < fieldDecls.length; i++){
    Class type = fieldDecls[i].getType();
```

```
    if (type == Boolean.TYPE)
      c = $< `Stmt(c)
             stream.writeBoolean(
               fields[`Int(hier)][`Int(i)].getBoolean(obj));
          >$;
    else if ... // other primitive types
    else // non-primitive type. write the field via Jumbo OOS
      c = $< `Stmt(c)
             oos.writeObject(
               fields[`Int(hier)][`Int(i)].get(obj));
          >$;
  }
  return c;
}
```

In the code above, `fields[][]` holds the field specifiers. The first index corresponds to the position of the class descriptor in the hierarchy, and the second index corresponds to the position of the field in that class descriptor. Note that the method requires `hier` as an argument. It doesn't need the `Object obj` parameter anymore, in contrast to the implementation of `writeFields` in Kaffe OOS. The code shows that if the field is non-primitive, it is passed to the Jumbo OOS to be written. In fact, we keep a one-element cache in the specialized marshaller associated with each non-primitive field; if the run-time type of the field is the same as the one in cache, we call the associated specialized marshaller without passing the object to Jumbo OOS. This saves us from the hashtable lookup that would occur in Jumbo OOS. If there is a cache miss, we pass the object to Jumbo OOS, it does a hashtable lookup, writes the object and then we update the cache. We do not give this code because of space limitations.

After we have the methods that return `Code` to serialize an object, we need to generate the `init` method[2], which will set up the data in the generated marshaller. In particular, this method is where the class handle is assigned to a data member of the serializer and where the `fields[][]` matrix is set. Note that this happens only once per generated serializer. This initializer method is constructed using code pieces from Kaffe OOS. Therefore writing this method is again straightforward, and to save space we don't provide the source code here.

The generated marshallers implement an interface called `Marshaller`, which defines the methods `init` and `write`. Interfaces, or abstract classes, are normally required in Java when ordinary code is to call generated code [3,5,2].

5 Performance

When using RTPG, the cost of run-time program generation must be taken into account. For this cost to pay off, we need to use the generated program a lot; that

[2] Java doesn't provide the ability to pass arguments to the constructors of dynamically loaded classes, so the class can only have a zero-argument constructor [3,13]. Thus we define a normal method, `init`, and call it right after the object is created via the zero-argument constructor.

is, we need to marshal a large data set. Still, the running time of the generated code — excluding compile time — is a useful quantity to know, because it gives the upper limit of speed-up (to which the actual speed-up will converge, over time). In this section, we give the performance of specialized marshallers, both including and excluding the cost of run-time compilation.

The performance of marshalling code is highly dependent upon the properties of the data being marshalled. Furthermore, it is not clear what should count as a "realistic" workload for marshalling. Large data sets — which are the ones we most care about, since these will be the most time-consuming to marshal — are likely to consist of large numbers of a few kinds of objects; this would be characteristic of video or audio streams, for example. On the other hand, most data in Java consists of objects of many different types. From the point of view of run-time program generation, these two scenarios have very different performance characteristics. Accordingly, we show benchmarks of both kinds. Specifically, we start by marshalling large, homogeneous collections of a class called `Dummy`, which has several fields. Then we test a linked-list class, and a class similar to `Dummy`, but with fields which can contain either of two types of objects (one a subclass of the other). After showing benchmarks for these homogeneous and near-homogeneous collections, we discuss a non-homogeneous data set, containing objects of 66 different classes.

Table 1. Performance table for `Dummy` class. Crossover point is 250 objects

Object Count	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	Kaffe Jumbo	Kaffe Jumbo+comp.
1000	30000	6.6	26.9	152.9	23.1	5.68
10000	300000	121.1	140.6	1545.0	12.75	10.99
20000	600000	257.8	277.0	3121.4	12.1	11.27

These benchmarks are run as follows: All the tests are executed on a Linux Debian, AMD Athlon XP 1700+ machine with 900MB memory. The timings are in milliseconds. We use HotSpot as the Java Virtual Machine, which is the most popular JVM[3]. When running a test, we first marshal a substantial number of objects to give the virtual machine time to *warm up*. During this time, the JVM loads classes and performs just-in-time optimization. Our experience has shown that this approach gives more consistent results. After warming up the JVM, we begin the test. We create a certain number of serializable objects, then pass the objects to the OOS's and measure the time spent. We call this *a benchmark*. After a benchmark is done, we discard the objects and OOS's —together with the hashtables they contain— and run another benchmark with a different number of objects. Thus, each benchmark begins with the Jumbo API and OOS's loaded

[3] Performance measurements with IBM's JVM [14] actually show significantly better speedups for Jumbo, but space prevents us from including these timings.

and optimized, the specialized marshallers not generated. In the tables below, each row represents a benchmark.

5.1 Homogeneous and Near-Homogeneous Data

Table 1 gives the results for marshalling objects of the `Dummy` class:

```
public class Dummy implements Serializable {
    Simple simple1;
    Simple simple2;
    int id;
}

public class Simple implements Serializable {
    int id;
}
```

The Jumbo OOS column does not include the run-time compilation cost, but Jumbo+compilation does. We have shown timings for marshalling 1000 to 20000 objects. The "Bytes written" column gives the size of the data written to the output stream. Jumbo OOS is at least 12 times faster than Kaffe OOS, when run-time generation cost is not included.[4]

Table 2. Performance table for linked-lists of `Dummy` objects. Each list has fifty nodes.

Number of lists	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	Kaffe Jumbo	Kaffe Jumbo+comp.
10	19363	6.7	48.9	145.0	21.42	2.96
50	84479	45.1	71.9	723.9	16.04	10.06
100	186877	107.7	131.3	1496.6	13.88	11.39
150	246075	115.8	135.4	2145.9	18.52	15.84
200	352161	144.6	174.4	2896.0	20.02	16.60

In our next test, we marshal linked-lists of Dummy nodes (same as `Node` class, but with data of type `Dummy`). Each linked list has 50 nodes. Jumbo OOS is up to 20 times faster than Kaffe OOS in this test. (See Table 2.)

Inheritance affects the cost of marshalling because it requires that we test the type of each field and not simply call the marshaller for the declared type of the field[5]. In the previous benchmarks, we did not marshal any objects whose classes had subclasses; thus, the actual type of every marshalled object was the

[4] The crossover points we give were determined by direct observation, not by interpolation from the presented data. We have omitted the timings for smaller data sets for lack of space.

[5] Remember that to eliminate some hashtable lookups, we associate a one-element cache with each field. See Section 4.

same as its declared type, and, in particular, the one-element cache always held the right class. For the next benchmark (Table 3), we marshal `Dummy` objects, but allow the fields of type `Simple` to contain either a `Simple` or a `SimpleChild` object, determined randomly. The `SimpleChild` class is shown below.

```
public class SimpleChild extends Simple{
    int otherId;
}
```

Table 3. Performance table for `Dummy` objects, allowing the fields to be either `Simple` or `SimpleChild`. Crossover point is 280 objects.

Object Count	Bytes written	Jumbo OOS	Jumbo + comp.	Kaffe OOS	Kaffe / Jumbo	Kaffe / Jumbo+comp.
1000	30136	9.9	55.3	154.7	15.55	2.80
10000	334320	136.3	167.2	1637.0	12.0	9.79
20000	641548	285.1	303.9	3237.3	11.35	10.65

5.2 Non-homogeneous Data

Data commonly consist of many objects of a variety of classes. This has a significant effect on the performance of our code because it implies a lot more classes being generated and therefore a lot more program generation time. In this section we examine the behaviour of Jumbo OOS on such data.

Table 4. Performance table for heterogeneous data. The objects come from a total of 66 classes.

Object Count	Bytes written	Jumbo OOS	Jumbo + comp.	Kaffe OOS	Kaffe / Jumbo	Kaffe / Jumbo+comp.
13210	128140	76.5	1504.1	1830.0	23.92	1.22
39630	372578	239.5	1690.9	5486.0	22.9	3.24
66050	617016	368.2	1837.9	9248.0	25.11	5.03
92470	861454	524.3	1899.5	12789.2	24.39	6.73
118890	1105892	657.4	2065.5	16499.7	25.09	7.99

For this purpose, we serialize `Code` objects. `Code` is the type of partially-compiled program fragments, as described earlier. In total, the `Code` objects indirectly touch 13210 objects, from 66 classes; 127 kilobytes were written to the stream. The timings are given in Table 4. We start by marshalling just one `Code` object, and increment by two on each row (i.e. marshal the object two more times than on the previous row). In this test, Jumbo OOS is faster than Kaffe OOS by approximately 25 times, when the cost of program generation is not counted. However, when code generation time is counted, the improvement relative to the Kaffe OOS goes down to about 1.22 in the worst case. The speed-up will approach 25 as the size of the data set increases, but it only achieves an eight-fold increase on the largest data set we tried.

The generated code shows much less speed-up than for the homogeneous case. Recall that the crossover point when marshalling `Dummy` objects was about 250 objects; now it is about 10500 objects. The problem, of course, is that we are generating code for many classes that have a small number of instances. We discuss this issue in the next section.

6 Just-in-time Program Generation

When marshalling heterogeneous data like `Code`, many classes are represented by only a few objects, and the cost of generating the marshalling code for those classes is not repaid. Our analysis of the test with heterogeneous data showed that only 14 out of the 66 classes allocated more than 250 objects. (Recall that, for `Dummy` objects, the crossover point was 250 objects.) Clearly, the remaining 52 classes will create a significant drag on the overall marshalling process.

To test the hypothesis that avoiding code generation for classes with few objects will yield better results, we ran a set of tests using varying *threshold* values: For each threshhold value, we generated code only for those classes which produce at least that many objects in the benchmark. This depends upon our having counted the number of objects for each class beforehand, so this does not represent a viable implementation strategy; we are only attempting to prove our hypothesis. We see (Table 5) that at a threshold value of 100, the generated code produces nearly a four times speedup over Kaffe OOS (compared to 1.22 fold speedup when all marshallers are generated). Note that, even at the optimal threshold value of 100, the speedup we can obtain in this situation is much less than we did with the simpler, homogeneous collections, because (1) the cost of run-time compilation is great due to the large number of classes and (2) many objects are marshalled by non-generated code.

Table 5. Performance comparison when threshold value is used. Marshallers are generated only for classes known to have more than threshold number of instances.

Threshold	Object Count	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
20	13210	659.5	1861.2	2.82
60	13210	527.6	1871.3	3.54
100	13210	482.9	1872.0	3.87
140	13210	515.6	1869.9	3.62
180	13210	551.7	1855.4	3.36
300	13210	592.8	1871.0	3.15
400	13210	706.5	1868.1	2.64

In this experiment, the number of instances of each class was known prior to marshalling. What shall we do when we don't know that? The situation is similar to JIT compilation [15]. HotSpot keeps track of method calls and when a method is called a certain number of times, it is optimized.

Following this idea, our second version of the marshaller counts the number of objects marshalled. Once it has reached the threshold value, it generates specialized code and uses that for subsequent objects of the class. Note that this version will be slower than the previous one, because all objects marshalled prior to reaching the threshold value are marshalled by non-generated code. The results are shown in table 6. Here, we don't reach the previous speedup factor, but instead reach 3.16 (again with a threshold value of 100).

Table 6. Marshalling 13210 objects, with different threshold values

Threshold	Object Count	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
20	13210	920.3	1851.6	2.01
60	13210	669.5	1851.0	2.76
100	13210	585.6	1854.4	3.16
140	13210	594.8	1847.9	3.1
180	13210	606.9	1832.7	3.01
300	13210	629.8	1851.0	2.93
400	13210	732.5	1852.9	2.52

Our final version of the marshaller uses the "just-in-time" idea with a threshold value of 100. We ask our last question: Does this version extract a significant penalty when marshalling *homogeneous* data? Table 7 shows the timings for this version of the marshaller, when marshalling collections of `Dummy` objects. This table is comparable to Table 1, and it shows that the JIT approach has almost no effect on performance for large homogeneous data sets.

Table 7. Performance when marshalling `Dummy` objects with threshold value of 100

Object Count	Bytes written	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
1000	30000	37.0	151.7	4.09
10000	300000	149.9	1592.3	10.61
20000	600000	289.7	3167.4	10.93

It should be noted that if we have the opportunity to do off-line program generation, using specialized marshallers is the obvious decision, because we wouldn't have the run-time compilation cost. In this case, we would generate the specialized marshallers once before run-time and then at run-time we'd get the benefit of using them. Unfortunately off-line compilation is not always possible.

7 Sun's ObjectOutputStream

The aim of this paper is to show that RTPG using Jumbo is an easy and effective way to achieve higher performance. In this, we have reached the end of our

exposition. However, there are some loose ends to tie up. In particular, the reader may wonder how our code stacks up against the marshalling code that is delivered with HotSpot, which, as we have mentioned, uses unsafe, native code. (To be more specific, it uses the `sun.misc.Unsafe` class to access arbitrary memory addresses.) Another natural question is whether the kind of program generation we have done can be applied *to* the HotSpot code.

Table 8. Performance of Jumbo OOS vs. Sun OOS. Marshalling `Dummy` objects, program generation cost included, threshold value 100, incorporating lightweight hashtable

Object Count	Bytes written	Jumbo + compilation	Sun OOS	Jumbo+compilation / Sun
1000	30000	45.1	11.1	4.06
10000	300000	123.3	85.1	1.44
20000	600000	201.5	187.5	1.07

In table 8, we show the result of a test marshalling `Dummy` objects again, comparing Jumbo OOS (with threshold value of 100) to Sun OOS. To be fair to Jumbo OOS, we note that, in addition to using native methods, Sun OOS uses a custom, lightweight hashtable implementation, which is considerably more efficient than the standard implementation in this context. We incorporated this hashtable implementation into our code, too. In this test, Jumbo OOS is only 7% slower than Sun OOS on the largest data set, with 20,000 objects.

So, to summarize, while remaining entirely in the realm of verifiable Java code, we have obtained an implementation that can marshal large data sets nearly as fast as Sun's implementation.

Finally, we have experimented with applying RTPG to Sun OOS. We implemented Jumbo OOS and `ProgGen` using the same principles we discussed in Section 2 and 4, but based on Sun OOS instead of Kaffe OOS. (Although Sun OOS achieves its speed from using native methods in critical places, much of it is written in Java.) Comparing this version of Jumbo OOS to Sun OOS, we achieve speedups as high as 30% when run-time compilation cost is excluded. However, the crossover point is around 12,000 objects for homogeneous data sets.

8 Prior Work

Most work on optimizing marshalling is not directly comparable to ours in that the goal is not to optimize the existing, generic marshaller, but to create more efficient marshallers for special cases. For example, Nester et al. [1] require that classes that are to be marshalled must provide their own `writeObject` method, and also depart from the Sun serialization format in other ways which are valid in their environment, but not in general.

Manta [7] and Ibis [9] both use run-time code generation to produce specialized marshallers at run time. Their methods are different from ours: In Manta,

a compiler is invoked at run time (again requiring that all computers have a specified set-up in order to use their system); in Ibis, a specially built program generator producing JVM code has been written just to generate serializers.

Run-time program generation is the topic of many papers. The system closest to Jumbo is DynJava [16]. DynJava has certain restrictions, such as disallowing run-time generation of class names, which suggest that the translation from Kaffe OOS to DynJava might not be as straightforward as the translation to Jumbo; these restrictions are fundamental, as DynJava is type-safe. Nonetheless, we assume that, in general, DynJava could be used for marshalling much as we have done. Serialization is used as an example in two papers on RTPG systems that we know of. Neverov and Roe give the definition of a multi-stage language called Metaphor [17], in which, in principle, serialization code can be generated in a *type-safe* manner. However, they do not tackle the entire Java serialization specification, and it is not clear whether their techniques could scale to this case. Consel et al. [18] discuss marshalling for C, using the C-based Tempo system.

9 Conclusions

We have shown that marshalling code in Java can be highly optimized by generating marshallers at run-time. The speedup we obtained was an order of magnitude when compared to the marshalling code of Kaffe. For some data sets we nearly reached the speed of Sun's object serializer, which extensively uses unsafe native code, while staying entirely in the realm of verifiable byte code.

We applied a heuristic approach similar to just-in-time compilation to lower the break-even point for heterogeneous data sets. Another method which can further decrease runtime compilation cost is to optimize program generators statically. This approach is discussed in [19].

We have also shown that the transformation from the classical code to program generating code using `Jumbo` is straightforward. It does not require skills beyond ordinary programming. We conclude that considering this fact and the high speedup we obtained, optimizing marshalling is a potentially useful application of run-time program generation.

Jumbo itself, and all the code used in the experiments for this paper, can be obtained at `loome.cs.uiuc.edu/Jumbo/index.php`.

Acknowledgements

We would like to thank the (anonymous) reviewers, whose detailed comments on this paper were extremely valuable.

References

1. Nester, C., Philippsen, M., Haumacher, B.: A more efficient RMI for Java. In: Proc. of the ACM 1999 Java Grande, New York, NY, USA, ACM Press (1999) 152–159

2. Kamin, S., Clausen, L., Jarvis, A.: Jumbo: Run-time Code Generation for Java and its Applications. In: Proc. of the Intl. Symposium on Code Generation and Optimization (CGO '03), IEEE Computer Society (2003) 48–56
3. Clausen, L.: Optimizations In Distributed Run-time Compilation. PhD thesis, University of Illinois at Urbana-Champaign (2004)
4. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-2: Binary-Level Components. In: Intl. Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00), Springer-Verlag (2000) 28–50
5. Kamin, S.: Routine Run-time Code Generation. In: Companion of the 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), ACM Press (2003) 208–220
6. Kamin, S.: Program Generation Considered Easy. In: Proc. of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04), ACM Press (2004) 68–79
7. Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H.E., Kielmann, T., Jacobs, C., Hofman, R.: Efficient Java RMI for Parallel Programming. ACM Trans. Program. Lang. Syst. **23** (2001) 747–775
8. Veldema, R., Philippsen, M.: Compiler Optimized Remote Method Invocation. In: IEEE International Conference on Cluster Computing (CLUSTER'03). (2003) 127
9. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R.F.H., Jacobs, C.J.H., Kielmann, T., Bal, H.E.: Ibis: A Flexible and Efficient Java-based Grid Programming Environment. Concurrency and Computation: Practice and Experience **17** (2005) 1079–1107
10. Kaffe JVM. (`http://www.kaffe.org`)
11. Java Object Serialization Specification. `http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html`
12. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
13. Java 1.4.2 API Documentation. (`http://java.sun.com/j2se/1.4.2/docs/api/`)
14. IBM-JVM. (`http://www-106.ibm.com/developerworks/java/jdk/`)
15. Advanced Programming for the Java 2 Platform: Ch. 8; Performance Features, Tools. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html`
16. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type Safe Dynamic Code Generation in Java. In: The 3rd JSSST Workshop on Programming and Programming Languages. (2001)
17. Neverov, G., Roe, P.: Metaphor: A Multi-stage, Object-Oriented Programming Language. In: Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE '04). (Lecture Notes in Computer Science)
18. Consel, C., Lawall, J.L., Meur, A.F.L.: A Tour of Tempo: A Program Specializer for the C Language. Sci. Comput. Program. **52** (2004) 341–370
19. Kamin, S., Aktemur, T.B., Morton, P.: Source-level optimization of run-time program generators. In: Generative Programming and Component Engineering (GPCE '05). (2005)

Applying a Generative Technique for Enhanced Genericity and Maintainability on the J2EE Platform

Yang Jun and Stan Jarzabek

Department of Computer Science, School of Computing,
National University of Singapore,
Lower Kent Ridge Road,Singapore 117543
{yangjun, stan}@comp.nus.edu.sg

Abstract. One of the themes in building reusable and maintainable software is identifying similarities and designing generic solutions to unify similarity patterns. In this paper, we analyze capabilities of J2EE to effectively unify similarity patterns found in Web Portals (WP). Our experimentation involved a family of WPs to support information sharing and team collaboration, built by our industry partner. While J2EE provides useful mechanisms for reuse of common services across components, we found its limitations in systematic across-the-board reuse in application domain-specific areas. To solve these problems, we applied a generative programming (GP) technique of XVCL on top of J2EE. By unifying similarity patterns, we increased the clarity of portal's conceptual structure as perceived by developers, reducing also the size of the original J2EE WP by 61%. Our solution enhanced traceability of information that mattered during changes. Based on that we hypothesized that XVCL-enhanced J2EE WP would be easier to maintain than the original J2EE WP. In the paper, we describe our solution and evaluate its engineering merits in both quantitative and qualitative ways.

1 Introduction

Web Portals (WP for short) were introduced around 1998 when the WWW became a standard medium for accessing information. As an effective means for knowledge management and delivering business intelligence on demand, WPs have moved from the fringes of business to a core competency, in the span of a few short years. Today's enterprises gain competitive advantage from quick development and deployment of custom WPs. WPs are developed and maintained under tight schedules. They are characterized by imprecise and frequently changed requirements. Portability and scalability are also important, as WPs may need support complete enterprise-wide services, accessible by thousands of clients simultaneously. All this creates unique challenges for WP engineering. Conventional engineering methods and processes must be substantially adjusted to meet the realities of the Web development and are in great demand.

In our earlier study of 17 WPs, we found similarity rates of 17-63% [17], measured in terms of code contained in clones. We analyzed only simple clones, that is, similar

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 237 – 255, 2005.

code fragments. It should be expected that taking into account design-level similarities, the rate of cloned code could be higher. Indeed, a project by our industry partner ST Electronics (Info-Software Systems) Pte. Ltd. fully confirmed this expectation [13]. Similarity rates in Active Server Pages (ASP) [1] WPs were around 60%, and in certain areas as high as 90%. By applying a generative programming (GP) technique of XVCL [22] on top of the ASP, a "generic WP" could be designed, turning the original WP into a WP product line architecture, from which many similar but distinct WPs could be derived at much a lower cost than it was possible with conventional methods. Maintenance productivity figures for the generic WP, as well as for WPs derived from it, were very encouraging, too. The results clearly indicated that, in this project, despite a careful model-based design, ASP and associated techniques were not effective in unifying many types of WP similarity patterns.

The above studies hinted at a potential of GP to improve major Web engineering productivity indicators, such as development/maintenance effort or time-to-market. This motivated us to pursue research in similar direction on the J2EE platform, as described in this project. The Java™ 2 Platform, Enterprise Edition (J2EE) is widely used for WP development. J2EE simplifies WPs by basing them on standardized, modular components, and providing many useful services for those components. This allows developers to focus on essential business logic of applications, while reusing the infrastructure code. Unlike ASP, J2EE supports inheritance, generics and other OO features via Java 1.5. An intriguing question was which of the problems observed in our earlier studies could be solved by J2EE and which still remained a challenge.

Despite many powerful J2EE features, we have found that many similarity patterns in WPs could not be unified with generic design solutions expressed by J2EE mechanisms. This weakness of J2EE particularly showed in application domain-specific program areas. It led to many repetitions in WP development, at both design and implementation levels, which made reuse and maintenance more difficult.

In this paper, we describe a possible solution to the above problems. The essence of our solution is synergistic application of J2EE mechanisms together with a generative programming (GP) technique of XVCL [22]: Whenever J2EE component mechanisms become to restrictive to conveniently unify certain similarity patterns in WPs, we apply GP to do the job. We call such an application of a GP together with conventional technologies a *mixed strategy* solution.

We based our experiment on CAP-WP, a portal developed on the J2EE platform by our industry partner ST Electronics (Info-Software Systems) Pte. Ltd. CAP-WP supported collaborative work and included modules such as News or Forum. We studied both inter- and intra-module similarities in CAP-WP design. We applied XVCL to unify similarity patterns, with the goal of enhanced maintainability and reusability of the CAP-WP. By doing this, we increased the clarity of portal's conceptual structure as perceived by developers, reducing also the size of the original CAP-WP by 61%. Our solution enhanced the visibility of relationships among program elements that mattered during changes, reducing the risk of update anomalies. Based on that we hypothesized that XVCL-enhanced WP would be easier to maintain than the original CAP-WP. We

conducted a controlled experiment in which we observed that the number of modifications required for the same WP enhancement, as well as the effort to implement them, was substantially smaller in our solution than in the original CAP-WP. We observed a similar correlation between non-redundancy of program representation and the number of modification points in other projects. Still, we realize that the artificial setting and small scope of the experiment only partially supports our hypothesis.

In addition to the above mentioned portal simplifications, similarity unification with XVCL also paved a way to a WP product line architecture that could facilitate reuse-based development of other similar WPs.

In other projects, we described XVCL solutions to generic design problems in Java [11] and C++ [2] class libraries. In yet other papers, we described capabilities of XVCL as a variability realization mechanism for product lines [22][23]. In this paper, we show that fundamental problems that we observed in other projects also manifest in Web Portals and modern component platforms such as J2EE provide only partial solution for them. We show that the roots of the problems are the same and a possible treatment can also be the same.

This paper is organized as follows: In Section 2, we introduce the CAP-WP. In Section 3, we analyze J2EE mechanisms for reuse, focusing on those that were applied in CAP-WP. In Section 4, we analyze similarity patterns in CA-WP. In Section 5, we describe an XVCL-enhanced WP obtained by unifying similarity patterns. In Section 6, we compare the original CAP-WP and our solution, in quantitative and qualitative ways. Related work and conclusions end the paper.

2 The Common Application Platform Web Portal (CAP-WP)

A Common Application Platform Web Portal (CAP-WP for short), developed by ST Electronics (Info-Software Systems) Pte. Ltd, was a typical J2EE-based WP. CAP-WP supported News, Forum, Access Statistics, Posting/Feedback facilities, and many other functions typically found in collaborative environments. CAP-WP facilitated information sharing via management of users, HTML-content, images and video-clips. It could be used to help team members collaborate online in software development projects.

The developers of CAP-WP utilized model-based design[1] and J2EE mechanisms to achieve simplicity of the design, maintainability, and reusability within the scope of CAP-WP. As most of other component platforms, J2EE considerably influenced the design of CAP-WP. All the WP modules (such as News or Forums) were portlet components built on top of the J2EE portlet container, using the Portlet API. Portlet APIs conformed to Java Portlet Specification (JSR168).

CAP-WP comprised 14 portlet modules built of 148 packages. A subset of the functional portlets in CAP-WP is illustrated in Fig. 1.

[1] In model-based design, we create a model of an application from which we derive (generate or manually) parts of an application.

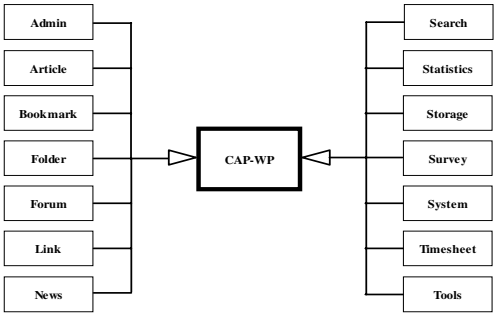


Fig. 1. Some of the CAP-WP portlet modules

In addition to the functional portlets shown in Fig. 1, CAP-WP also provided facilities for WP page configuration and customization. Developers could develop their own portlets on any J2EE platform and plug them into the CAP-WP system. Then, by simply registering those portlets and WP pages using the function provided by the Admin portlet in CAP-WP, new WP pages could be created freely, so consequently new WPs could be generated as well.

3 J2EE's Support for Reuse in CAP-WP

CAP-WP benefited from J2EE mechanisms from the architecture level to the implementation of specific portlets. CAP-WP architecture was based on the five-layer J2EE model (Fig. 2). Users interacted with dynamic HTML (the Client Layer), and JSP/Portlet played as the Presentation Layer. Portlet Container implemented the Business Layer. JDBC and MySQL Database were at the Data Access Layer and the Resource Layer. The five-layer architecture allowed developers to build modules comprising a collection of client and server components forming logical parts of an

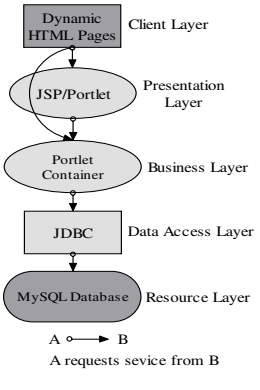


Fig. 2. Five-layer architecture in CAP-WP

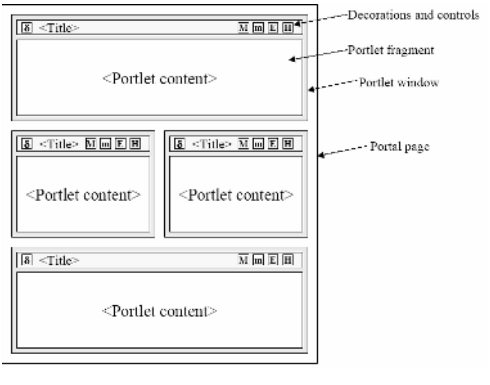


Fig. 3. Elements of a typical WP page

application. Such modules are more meaningful than isolated components. Their reuse was further helped by suitable APIs.

Portlet technology is an important reuse mechanism provided by J2EE. Portlets define generic functionality that can be reused in many variant forms. A portlet generates markup fragments for WPs. A specific WP, such as CAP-WP, normally adds a title, control buttons and other decorations to the markup fragment generated by the portlet. This new instantiated fragment is called a portlet window. WP then aggregates portlet windows into a complete document, a WP page displayed to the user. A WP page can be a composite of portlet-generated elements of various sizes, forming a hierarchical structure, as shown in Fig. 3. Being independent of WPs and WP pages, J2EE portlets are generic and reusable.

The advantage of portlet technology is that with the service provided by the portlet container, portlet actions can be separated from the WP core. Using portlets, developers do not need to completely re-implement the whole WP for each of the required products over and over again. Simply changing the portlets arrangements and making corresponding minor modifications to the WP configuration serve the purpose.

J2EE design patterns are supportive to reuse and CAP-WP used many of them. CAP-WP used the J2EE Intercepting Filter pattern (to facilitate pre- and post-processing of a portlet request), the Front Controller pattern (to provide a centralized controller for handling portlet requests), and many others. The Data Access Object pattern (DAO) played a particularly important role. DAO provides an abstraction layer between the business logic and data source objects (the persistent storage). Business objects access data sources via data access objects. The DAO pattern separates the WP from the low-level database-access code for specific persistent storage (database) a WP uses. CAP-WP system used MySQL database. The DAO pattern made CAP-WP components easily portable across databases of other vendors.

4 Analysis of Similarity Patterns in CAP-WP

In the above section, we saw examples of J2EE mechanisms that CAP-WP developers used to achieve reuse. Here, we concentrate on similarity patterns in CAP-WP that could not be treated with J2EE mechanisms. By similarity patterns, we mean similar program structures of any kind and granularity, repeated many times within a program or across programs. Similarity patterns show as clones. We distinguished two kinds of clones, namely:

- *simple clones*: contiguous segments of similar code such as class methods or fragments of method implementation, and
- *structural clones*: patterns of inter-related classes emerging from design and analysis spaces; patterns of components at the architecture level; design solutions repeatedly applied by programmers to solve similar problems.

4.1 Types of Similarity Patterns in CAP-WP

Fig. 4 shows portlets implementing CAP-WP modules. Each portlet (Level 1) is built of different *functional packages* (Level 2), whose operations are implemented as *portlet classes* (Level 3).

In each of the 14 portlets (Level 1), there are four types of functional packages, namely Action, Request, Util and View (Level 2). Functional packages of the same type for different portlets are implemented by similar portlet classes (Level 3). We explain this structure and the nature of similarities in more details later.

We classify similarity patterns in the CAP-WP into two types, namely:

- *inter-portlet similarities*: similarity patterns across different portlets, and
- *intra-portlet similarities*: similarity patterns within the same portlet.

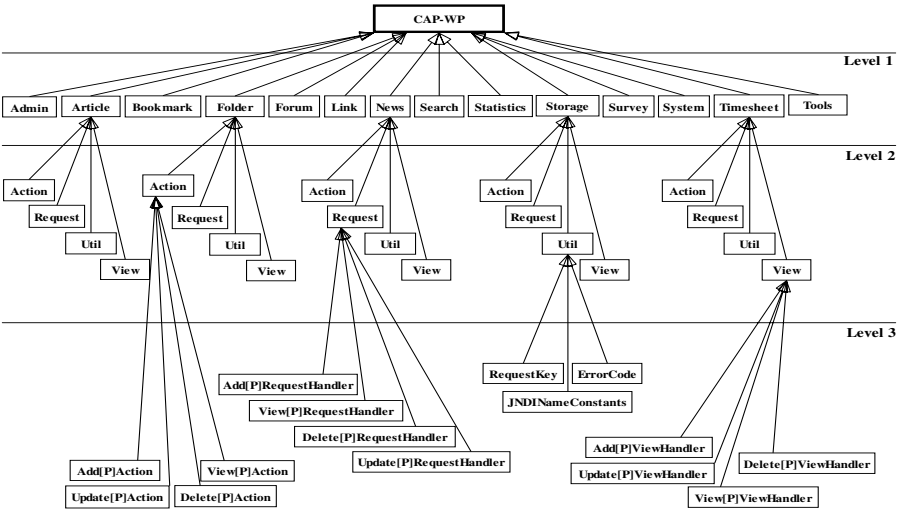


Fig. 4. Similarities in CAP-WP

Inter-portlet Similarities: Many functional packages were similar across CAP-WP portlets. We classified 66 functional packages found in 14 portlets into nine package types formed by functional packages that occurred in some or all of the CAP-WP portlets, in similar form. Four of those package types, namely Action, Request, Util and View, were in each of the 14 portlets. In addition, eight portlets contained Model package; seven portlets contained Dao and Dao.impl package; three portlets contained View.taglib package; finally, just one portlet contained View.servlet package.

At Level 3 of Fig. 4, we indicated portlet classes any of the four functional packages shown at Level 2 was built of. Parameter **P** indicates a portlet (such as Link, News, etc.) a given portlet class belongs to. While similar, portlet classes implementing functional packages of different portlets P (e.g., Add[**P**]Action) also differed in various ways. However, the similarities were significant enough to consider each group of such portlet classes as a "generic" class. Consequently, functional packages of, e.g., Action type for different portlets were similar enough to consider them instances of a "generic" Action package (similarly for Request, Util and View packages).

As an example, consider a group of 14 Add[**P**]Action classes. In all 14 classes, constructors were implemented in a similar way. Each class contained only two public

methods init(…) and service(…). Ten classes contained private method getUserTransaction (…), which did not exist in the remaining four classes.

However, there were differences in both method signatures (e.g., different return type or different number and types of arguments), and the details of method implementation across classes. Fig. 5 shows signatures of method service(…) in classes AddBookmarkAction and AddNewsAction from portlet Bookmark and News, respectively.

The differences in method implementation detailed ranged between small parametric variations to completely different implementation of a the same method in different classes. Fig. 6 shows different implementation of method init(…) in classes AddArticleAction and AddNewsAction from portlets Article and News.

```
public class AddBookmarkAction implements
Action {
      //other methods
      public AppResultSupport service (AppEvent
event) throws ActionException {
      …
      }
}
```

```
public class AddNewsAction implements Action {

      //other methods
      public AppResult service (AppEvent event,
News news) throws ActionException {
      …
      }
}
```

Fig. 5. Comparison between service(…) methods across classes

```
public class AddArticleAction implements Action {
      private PortletConfig    _config = null;
      public void init (PortletConfig config) throws
PortletException    {
            _config = config;
      }
}
```

```
public class AddNewsAction implements Action {

      public void init (PortletConfig config) throws
PortletException    {
      }
}
```

Fig. 6. Comparison between init(…) methods across classes

Intra-portlet Similarities: We also observed much similarity within functional packages for specific portlets. Consider *Action* package of portlet Survey as an example. This package contained 23 portlet classes, and all of them had the same class attributes and constructors. Methods in those classes were also very similar to each other. For example, we found method **service(AppEvent event)**, shown in Fig. 7, recurring in those classes, with small variations highlighted in bold. Those variations in data type names and algorithmic details resulted from the overlapping impact of various features on classes (or specific methods). Some of such variations often cannot be unified with generics or templates in a simple way [11][2].

```
public AppResult service (AppEvent event) throws ActionException {
        UserTransaction tx = null;
        try    {
                HashMap map = (HashMap) event.getEventObject ();
                String surveyId = (String) map.get ("survey_id");
                String [] roles = (String []) map.get ("roleids");
                tx = getUserTransaction ();
                tx.begin ();
                SurveyAccess access = SurveyAccess.getInstance ();
                for (int i = 0; i < roles.length; i++)
                        access.assignSurveyToRole (surveyId, roles [i]);
                tx.commit ();
                tx = null;
                return new AppResultSupport (getClass ().getName (), null, true);
        }
        catch (Exception ec)    {
                throw new ActionException
                        (ErrorCode.ERROR_ASSIGN_SURVEY_ROLE, ec);
        }
        //other implementation details
}
```

Fig. 7. Method service(AppEvent event)

Summary of observed similarities: Both the nature and degree of intra-portlet similarities were quite different from inter-portlet similarities. Intra-portlet similarities were mostly confined to simple clones, that is similar class constructors/methods or fragments of them. For example, in portlet classes AddArticleAction and UpdateArticleAction within Action package of portlet Article, about 75% of the code was contained in exact clones. Further 20% of the code was contained in clones that differed in various details. This leaves us with only 5% of unique code in each of the two classes. The cloning situation in other groups of portlet classes was similar.

On the other hand, inter-portlet analysis revealed higher, design-level similarities that showed as groups of similar classes. The extent of inter-portlet similarities among portlet classes was normally less than that of the intra-portlet classes. Classes involved had similar class structure, but the specific method implementations inside were quite different. For example, portlet class AddArticleAction from Action package of portlet Article and AddLinkAction from Action package of portlet Link, had similar class constructors, method names and declarations. However, the class attributes and specific method implementations were quite different between these two classes. There were no identical code fragments. Still, the rate of similar code with slight variations was about 40%.

We conclude that both inter- and intra-portlet similarities were important and worth noticing. Their unification with suitable generic solutions could be beneficial from both maintainability and reusability perspective. Despite lower similarity rates, we expected much engineering benefit from unification of inter-portlet similarities, as groups of similar classes represented fundamental design concepts in the WP domain.

4.2 Why Did Conventional Methods Fail to Unify Similarities?

Other than general OO techniques, we did not find any J2EE mechanisms that could help us design generic solutions unifying similarity patterns such as observed in the previous section. Here are specific examples illustrating the difficulties in unifying similarity patterns:

```
/*Translate the HTTP request to a specific application event object.*/

    public AppEvent translateRequest (PortletRequest req,
    RequestHandlerParam param) throws EventException {
        String username = req.getRemoteUser ();
        String id = req.getParameter ("link_id");
        HashMap map = new HashMap ();
        map.put ("link_id", id);
        map.put ("username", username);
        return new AppEventSupport (EVENT_NAME, map);
    }
```

Fig. 8. Recurring method translateRequest(…)

Method **translateRequest(…)** shown in Fig. 8 is repeated in four different classes in the *Request* package of portlet Link, in the same form. The method cannot be pushed up to the parent class, as the class attribute **EVENT_NAME** must be initialized differently in different classes. For example, in class ViewUserLinkGroupRequestHandler variable **EVENT_NAME** must be initialized as **"wcap.portal.link.event. ViewUserLinkGroupEvent"** while in class DeleteUserLinkRequestHandler such a variable needed to be initialized as **"wcap.portal.link. event.DeleteUserLinkEvent"**.

In some situations, to unify similar methods or classes, we need parameters representing algorithmic elements rather than data types. In yet other situations, we

```
/**Render the content of the portlet based on the result generated**/
public void render (PortletRequest req, PortletResponse resp, AppResult [] result)
                                                    throws ViewException {
    PortletContext context = _config.getPortletContext ();
    PortletURL url = context.createPortletURL ();
    try    {
        url.setPortletModule ("Folder");
        url.setPortletTask (PortletTask.LIST);
        url.setParameter ("content_id", req.getParameter ("content_id"));
        resp.sendRedirect (url.toString ());
    }
    catch (Exception e)    {
        Log.error ("DeleteFileViewHandler.render –Unable to include view
                                                    folder:" + e.getMessage ());
        throw new ViewException
                                    (ErrorCode.ERROR_DELETE_FILE, e);
    }
}
```

Fig. 9. Method render(...)

have to do with many small differences across implementations of the same method in different classes. We observe this in the classes in *View* package of portlet Folder. For example, method **render(…)** recurs in all the classes in that particular package with small changes highlighted in bold (as in Fig. 9). This happens when the impact of various features overlaps in code fragments, affecting data type names, constant values or details of algorithms. In such cases the developer has to create wrapper classes just for the purpose of parameterization. And those wrapper classes introduce extra complexity and hamper performance [11].

The differences among classes playing the same role in different portlets were a consequence of differences in portlet requirements. The differences were rarely confined to type parameters, and propagated across classes in rather ad hoc way, making abstracting commonalities to parent classes difficult. J2EE basically relies on the OO mechanisms to handle the above problems. The problems with applying conventional OO techniques to unify similar classes and class methods encountered in this project were very similar to those we observed in our earlier studies of class libraries [11][2].

The explosion of similar components in the original CAP-WP was a symptom of "feature combinatorics" problem first observed by Batory [5]: Our features are functions depicted at Level 2 and Level 3 in Fig. 4. Legal combinations of features lead to repetitions. For a domain with n optional features, in the worst case, 2^n concrete programs have to be created. The impact of features often spreads through many component layers, and multiple components within each layer. The arbitrary nature of feature impact causes that similarity patterns are often irregular, difficult to unify with conventional programming techniques. The findings about the nature of similarities and their reasons from the WP domain resemble observations from our earlier studies [11][2].

5 XVCL-Enhanced J2EE Solution

5.1 Introduction to XVCL

XVCL (XML-based Variant Configuration Language) [19][22] is a general-purpose meta-language, method and tool for enhancing changeability and genericity of programs. First, we develop a program with one of the programming languages and with conventional design techniques, to achieve proper program modularization and required runtime properties such as performance or reliability. Then, we apply XVCL on top of a program to facilitate change and/or to inject extra levels of genericity into it.

XVCL partitions a program into meta-components. XVCL meta-components are called x-frames. Meta-level partitioning is independent of (therefore, does not conflict with) decomposition into program modules (e.g., classes, functions, higher-level components and sub-systems). X-frames form a hierarchically structured architecture, called an x-framework (Fig. 10). Decomposition along x-frame boundaries, hierarchical organization of x-frames and unrestrictive parameterization of x-frames with XVCL commands are the main XVCL mechanisms to facilitate changeability and

genericity. The x-frame body is written in the base language, which could be a programming language such as Java, an architecture specification language, or a natural language such as English.

XVCL commands allow composition of the x-frames, via <adapt> commands shown as arrows in Fig. 10. XVCL commands parameterize x-frames by marking variation points at which an x-frame can be customized. Meta-variables and meta-expressions offer a basic parameterization mechanism (via <set> and <value-of> commands). Further parameterization is achieved by selecting pre-defined options based on certain conditions (via <select> command), or modifying an x-frame contents by inserting code at designated break points (via <insert> and <break> commands).

Customization of an x-framework is supported by the XVCL Processor (Fig. 10). The Processor traverses an x-framework, interprets XVCL commands embedded in visited x-frames and emits the output (e.g., a custom program) into one or more output files. The main navigation over the x-framework customization process is exercised by a specification x-frame, called an SPC for short (though each x-frame typically also contains customization instructions). Each SPC specifies a different customization of an x-framework that results in different output program.

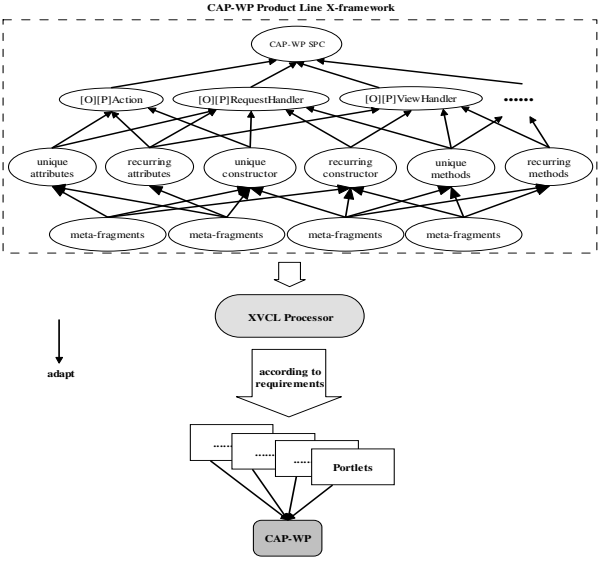


Fig. 10. Generating CAP-WP from a generic x-framework

An x-framework enables us to handle variants at all the granularity levels, which makes XVCL a variability realization technique, with possible applications in design of product line architectures. In this context, customization of an x-framework generates a member of a product line, in our case a WP similar to CAP-WP.

5.2 Construction of Portlet Classes with XVCL

Based on the analysis of similarities described in Section 4, we identified six major groups of similar classes, namely:

1. [**O**][**P**]Action: 18 classes which belong to *Action* packages in all portlets, for combinations of operations — **O**: Add, Update, View and Delete, with different portlets — **P**, like Forum, Link, News, etc.
2. [**O**][**P**]RequestHandler: 19 classes which belong to *Request* packages in all portlets, for combinations of operations **O** with different portlets — **P**.
3. [**O**][**P**]ViewHandler: 18 classes which belong to *View* packages in all portlets, for combinations of operations **O** with different portlets — **P**.
4. RequestKey: 9 classes with the same name appearing in *Util* packages in all portlets.
5. ErrorCode: 8 classes with the same name appearing in *Util* packages in all portlets.
6. JNDINameConstants: 9 classes with the same name appearing in *Util* packages in all portlets.

For each of the above groups, we designed x-frames to unify differences among classes in a given group. For example, meta-class [**O**]NewsAction (Fig. 11) facilitates generation of all the four portlet classes in *Action* package of portlet News, namely **AddNewsAction**, **UpdateNewsAction**, **DeleteNewsAction** and **ViewNewsAction**. These four classes are identical except the differences in class names and some implementation details in method **service(AppEvent event)**. The differences are handled by a <break> and <insert> command. The <break> point *seviceDetail* marks a variation point where method **service(AppEvent event)** can be customized, as needed in different classes. This customization is specified in the SPC (Fig. 12) : <adapt> commands for different options (indicating different classes) contain suitable <insert> commands that cater for differences among classes.

meta-class name: [O]NewsAction

Text

```
package wcap.portal.news.action;
public class @Action_CLASSNewsAction implements Action
{    // attributes and methods here
     public AppResult service (AppEvent event) throws ActionException {
          System.out.println ("Returning result action for event" +
                                                  event.getEventName());
```

break *serviceDetail*

text

```
          try {
               NewsFacade ts = NewsFacade.getInstance();
               InitialContext initialcontext = new InitialContext();
               UserTransaction ut = (UserTransaction) initialcontext.lookup
                         (JNDINameConstants.USER_TRANSACTION);
               ut.begin();
               try {
                    ut.commit();
               }
               //other implementation code …
          return new AppResultSupport (@Action_CLASS, @Return, true);
     }
}
```

Fig. 11. Meta-class [O]NewsAction

The SPC of Fig. 12 controls the overall process of generating all the classes in package *Action* of portlet News. First, the SPC <set>s the value of meta-variable Action_CLASS to <*Add, Update, Delete, View*> for the different class names and the value of meta-variable Return to <*news, news, null, news*> as the parameters of return object. Values of those meta-variables are propagated down to the <adapt>ed meta-components.

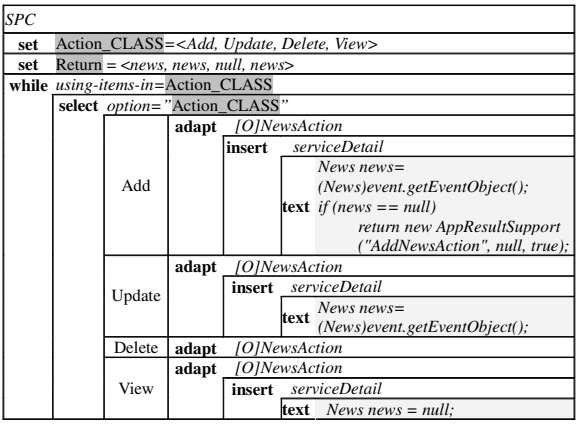


Fig. 12. SPC to construct classes in Action package for portlet News

The value of Action_CLASS is a list. Command <while> iterates over its body four times. In each iteration, Action_CLASS accepts one value from the list, in the left-to-right order. Based on that value, the processor <select>s a suitable option (such as Add, Update or otherwise) and generates code for appropriate class(es) (**AddNewsAction**, **UpdateNewsAction** and all the remaining classes, respectively). Generation is done by <adapt>ing the meta-class [**O**]NewsAction. To generate class **DeleteNewsAction**, we just adapt the meta-class without inserting any code at the break point for there is nothing in class **DeleteNewsAction** at this point. For the remaining classes, we inserted different code according to the different implementations in method **service(AppEvent event)** at this point.

Methods that recur in portlet classes without changes, such as method **translateRequest(…)** in Fig. 8, are included "as is". Methods that recur in portlet classes with some changes are adapted, depending on the context. For example, we parameterized method **render(…)** (Fig. 9) with meta-variables as shown in Fig. 13.

Such adaptations are achieved by means of parameterization via meta-variables and meta-expressions, insertions of code and specifications at designated break points, selection among given options based on conditions, code generation by iterating over sections of meta-components, etc. Parameterization via meta-variables and meta-expressions plays an important role in building generic, reusable programs. It provides the means for creating generic names and controlling the traversal and adaptation of a meta-component architecture.

meta-fragment name: render

text

```
/**Render the content of the portlet based on the result generated**/
    public void render (PortletRequest req,  PortletResponse resp,
            AppResult [] result) throws ViewException  {
        PortletContext context = _config.getPortletContext ();
        PortletURL url = context.createPortletURL ();
        try  {
            url.setPortletModule ("Folder");
            url.setPortletTask (PortletTask.LIST);
            url.setParameter (@urlParameters);
            resp.sendRedirect (url.toString ());
        }
        catch (Exception e)  {
            Log.error (@errorMessage);
            throw new ViewException
                        (@exceptionType, e);
        }
    }
```

Fig. 13. Meta-fragment render.xvcl

A reference to a meta-variable, such as @exceptionType, is replaced by the meta-variable's value during processing x-framework. The value of meta-variable *exceptionType* may be <set> to *ErrorCode.ERROR_UPLOAD_FILE*, *ErrorCode. ERROR_RETRIEVE_FILE*, *ErrorCode.ERROR_UPDATE_FILE*, etc., as required at the adaptation point. For example, to produce method **render (Portlet Request req, PortletResponse resp, AppResult [] result)** for class **UpdateFile ViewHandler,** we <set> the value of meta-variable *exceptionType* to *ErrorCode.ERROR_UPDATE_FILE*, and for classes **ViewFileViewHandler** and **DownloadFileViewHandler**, we <set> the value of this variable to *ErrorCode.ERROR_RETRIEVE_FILE*.

By unifying the similarity patterns at the meta-level as described above, we obtained a generic, XVCL-enhanced CAP-WP. Mechanisms implemented into our new CAP-WP provide a generic way to deal with variant features affecting major CAP-WP components. In future work, we plan to analyze WPs similar to CAP-WP, and extend the solution described in this paper to form a CAP-WP product line architecture like in Fig. 10.

6 Analysis of the Results

We applied XVCL to unify similarity patterns, taking into account both inter- and intra-module similarities in CAP-WP. Our solution represented each similarity pattern with a unique generic meta-level structure. Not only did such unification reduce the solution size by 61% (Fig. 14), but most importantly it increased the clarity of portal's conceptual structure as perceived by developers. In particular, it reduced the number of conceptual elements a programmer had to deal with and enhanced the visibility of relationships among program elements that mattered during changes: Rather than maintaining multiple variant code structures delocalized across the CAP-WP, in XVCL-enhanced solution a programmer dealt with one generic structure, with full

visibility of customizations required to produce instances of variant structures, as well as their exact locations. The non-redundancy of the XVCL-enhanced CAP-WP, achieved by generic meta-level structures unifying similarity patterns, reduced the risk of update anomalies.

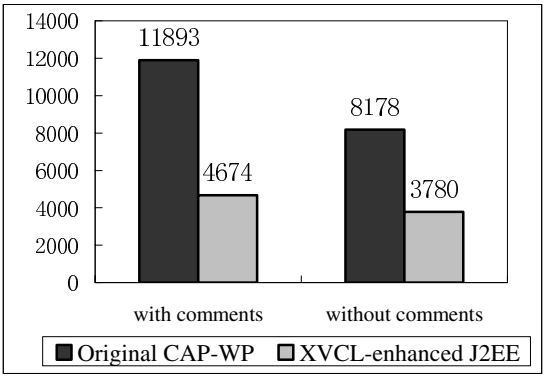


Fig. 14. Size comparison of the original CAP-WP and XVCL-enhanced solution

Based on the above analysis, we hypothesized that our solution, by enhancing traceability of information that mattered during changes, could facilitate easier maintenance. To test this hypothesis, we conducted a controlled experiment in which we implemented the same functional enhancement in both the original CAP-WP and our solution. The number of required modifications to implement the enhancement in our solution was 443 as compared to 1182 modifications in the original CAP-WP. This 63% reduction of modifications points interestingly correlates with 61% reduction of the solutions size counted in LOC. Reduction of modification points is not surprising as it is a direct consequence of unifying similar code structures with generic meta-level structures: One modification point in the meta-level structure usually maps to many modification points in its instances. We observed a similar correlation between non-redundancy of program representation and the number of modification points in other projects [11].

We also observed that due to improved traceability of information and conceptual clarity of the XVCL-enhanced CAP-WP, most of the modifications were easier to implement. Modifications done at one point of a meta-level representation consistently propagated to all the affected components. If the impact of change was not uniform in all such components, the exceptions could just be handled at the specific adaptation point, without directly modifying the component involved. The XVCL-enhanced CAP-WP solution effectively explicated and separated the impact of various sources of change.

Consider the meta-class [**O**]NewsAction (Fig. 11) which facilitates generation of classes **AddNewsAction**, **UpdateNewsAction**, **DeleteNewsAction** and **ViewNewsAction**. These four classes are identical except the differences in class names and some implementation details in method **service(AppEvent event)**. Suppose

there is a method **foo()** shared by the four classes requires some modifications during maintenance. In the original CAP-WP, we must examine four different locations and repeat the change four times. In our solution, we need modify method **foo()** only once in the meta-class [**O**]NewsAction and the changes propagate from there to all the four classes by adapting [**O**]NewsAction (Fig. 12). In case the change to method **foo()** is not uniform across classes, the differences can be specified at relevant adaptation points. We realize that the artificial setting and small scope of the experiment only partially supports our claims.

The size of a solution is just one among many factors that collectively determine the complexity of a program solution as perceived by a programmer. For example, by applying data compression techniques, we do not make a program any simpler for a programmer to understand. While in general code size reduction need not necessarily imply a simpler program, in the case of our experiment we found a correlation between the size and engineering qualities of the solutions.

To further support the hypothesis of improved maintainability of our solution, we analyzed experiences from the industrial application of XVCL to Web WP engineering [15]. This industrial project involved evolution from a single WP to an XVCL-enhanced generic WP, with more than 20 different WPs produced based on the generic WP. Despite differences in underlying technologies, the nature of problems solved with XVCL in this project was the same as in the CAP-WP project described in this paper. Therefore, productivity figures and qualitative analysis of the solution published in [15] further support, although only indirectly and speculatively, our hypothesis of improved maintainability of the J2EE-enhanced CAP-WP.

Imposing XVCL on top of the J2EE CAP-WP did not create major conceptual or technical difficulties, either. However, despite those benefits, design in terms of meta-components is not easy and also different from the conventional program design. A limitation of XVCL is that, being generic, x-frames can be difficult to understand. The verbose XML syntax also has negative impact on understanding x-frames. These problems can be mitigated by carefully designing x-frameworks according to XVCL design rules, and by re-factoring the design as an x-framework evolves. We are also developing XVCL Workbench to facilitate x-framework development. The XVCL Workbench includes tools such as a smart x-frame editor that hides XML syntax and displays graphical views of x-frames, and a static analysis tool that helps us understand an x-framework. We refer to other papers for a comprehensive discussion of trade-off involved in enhancing conventional program solutions with XVCL.

7 Related Work

Modular decomposition with information hiding [14], macros, generics in Ada or Java [10], templates in C++, other forms of parameterization such as higher order functions [21], inheritance with dynamic binding, and design patterns [9] are some of the conventional design techniques to achieve genericity. Aspect-Oriented Programming [12] and MDSOC [20] support genericity by separating cross-cutting concerns. In AHEAD [4], genericity is supported by feature composition and refinement. AHEAD models software as a mathematical structure of nested equations, making it possible to

study formal properties of refinements and resulting programs. Many techniques described under the umbrella of generative techniques [8] achieve forms of separation of concerns that is supportive to genericity (although avoiding repetition may not be a prime goal of these techniques). Domain analysis [15] is essential in identifying high-level, large granularity patterns of similarity. Generic solutions unifying such patterns are most beneficial for programmer's productivity as they can significantly reduce the size and complexity of the solution. Software architectures [6][7], architectural styles [17] and patterns [7] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability.

8 Conclusions

We analyzed capabilities of J2EE to effectively unify similarity patterns found in Web Portals (WP). Our experimentation involved a portal, called CAP-WP, to support information sharing and team collaboration, built by our industry partner. While J2EE provides useful mechanisms for reuse of services common to component-based systems, we found its limitations in systematic across-the-board reuse in application domain-specific areas. To solve these problems, we applied a generative programming (GP) technique of XVCL on top of J2EE. Our solution reduced the size of the original CAP-WP by 61% and increased the clarity of portal's conceptual structure as perceived by developers. It also enhanced traceability of information that mattered during changes, hopefully leading to improved maintainability of our solution over the original J2EE portal. In the paper, we described our solution and evaluated its engineering merits in both quantitative and qualitative ways.

In addition, the approach paved the way to a WP product line: The meta-structures of the XVCL-enhanced-J2EE solution could become building blocks of a WP product line architecture facilitating rapid development of other, WPs, similar to CAP-WP.

An important characteristic of the presented approach is synergistic application of a GP technique together with conventional OO and component-based development techniques. In such a "mixed strategy" solutions, a developer uses the OO paradigm to define a class/component structure of a program solution, and a GP technique (e.g., XVCL) to deal with genericity and changeability concerns. We believe that the GP techniques can add much value to the existing technologies for Web engineering. In the future, we plan to analyze WP functional and platform variants, and extend our solution to form a WP product line architecture. We plan to further investigate essential properties of WPs and technologies used for Web engineering in industries. We plan to investigate engineering qualities of "mixed strategy" with XVCL applied on top of other Web technologies.

Acknowledgements

ST Electronics (Info-Software Systems) Pte. Ltd. provided as with many useful inputs to this project. This work was supported by NUS research grant R-252-000-211-122.

References

1. Active Server Pages – ASP, http://msdn.microsoft.com/asp.net
2. Basit, H.A., Rajapakse, D.C., and Jarzabek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," Int. Conf. Software Engineering, ICSE'05, St. Louis, USA, May 2005, pp. 451-459
3. Basit, A.H. and Jarzabek, S. "Detecting Higher-level Similarity Patterns in Programs," to appear in ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2005, Lisbon
4. Batory, D., Sarvela, J.N. and Rauschmayer, A. 2003 "Scaling Step-Wise Refinement," *Proc. Int. Conf. on Software Engineering, ICSE'03*, May 2003, Portland, Oregon, pp. 187-197
5. Batory, D., Singhai, V., Sirkin, M. and Thomas, J. "Scalable software libraries,", *ACM SIGSOFT'93: Symp. on the Foundations of Software Engineering*, Los Angeles, California, Dec. 1993, pp. 191-199
6. Bosch, J. Design and Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Welsey, 2000
7. Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002
8. Czarnecki, K. and Eisenecker, U. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000
9. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley
10. Garcia, R. et al., "A Comparative Study of Language Support for Generic Programming," *Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, 2003, pp. 115-134.
11. Jarzabek, Stan. and Li, S., Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique. In Proceedings of ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2003, Helsinki, pp. 237-246.
12. Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. "Aspect-Oriented Programming," *Europ. Conf. on Object-Oriented Programming,* Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242
13. Len, B., Paul, C.and Rick, K., Software architecture in practice. Addison-Wesley. 1998
14. Parnas, D., 1972. On the Criteria To Be Used in Decomposing Software into Modules, *Communications of the ACM,* Vol. 15, No. 12, December, 1972, pp.1053-1058
15. Pettersson, U., and Jarzabek, S. "Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach," to appear in ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2005, Lisbon
16. Prieto-Diaz, R. "Domain analysis for reusability," *Proc. COMPSAC'87*, October 1987, Tokyo, Japan, pp. 23-29
17. Rajapakse, D.C and Jarzabek, S. "An Investigation of Cloning in Web Portals," to appear in Int. Conf. on Web Engineering, ICWE'05, July 2005, Sydney, pp. (also poster at WWW'05)
18. Shaw, M. and Garlan, D. Software Architecture: Perspectives on Emerging Discipline, Prentice Hall, 1996

19. Soe, M.S., Zhang, H. and Jarzabek, S. (2002). XVCL: A Tutorial. In Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia, Italy, 2002, pp.341-349.
20. Tarr, P., Ossher, H., Harrison, W. and Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proc. International Conference on Software Engineering*, *ICSE'99*, Los Angeles, 1999, pp. 107-119
21. Thompson, S., "Higher Order + Polymorphic = Reusable", unpublished manuscript available from the Computing Laboratory, University of Kent. http://www.cs.ukc.ac.uk/pubs/1997
22. XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://fxvcl.sourceforge.net
23. Zhang, W. and Jarzabek, S. "Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices," to appear in 9th Int. Software Product Line Conference, SPLC'05, September 2005, Rennes, France

Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code

Jacques Carette[1] and Oleg Kiselyov[2]

[1] McMaster University, 1280 Main St. West,
Hamilton, Ontario Canada L8S 4K1
[2] FNMOC, Monterey, CA 93943

Abstract. With Gaussian Elimination as a representative family of numerical and symbolic algorithms, we use multi-stage programming, monads and Ocaml's advanced module system to demonstrate the complete elimination of the abstraction overhead while avoiding any inspection of the generated code. We parameterize our Gaussian Elimination code to a great extent (over domain, matrix representations, determinant tracking, pivoting policies, result types, etc) at no run-time cost. Because the resulting code is generated just right and not changed afterwards, we enjoy MetaOCaml's guaranty that the generated code is well-typed. We further demonstrate that various abstraction parameters (aspects) can be made orthogonal and compositional, even in the presence of name-generation for temporaries and other bindings and "interleaving" of aspects. We also show how to encode some domain-specific knowledge so that "clearly wrong" compositions can be statically rejected by the compiler when processing the generator rather than the generated code.

1 Introduction

In high-performance, symbolic, and numeric computing, there is a well-known issue of balancing between maximal performance and the level of abstraction at which code is written. Furthermore, already in linear algebra, there is a wealth of different aspects that *may* need to be addressed. For example, implementations of the widely used Gaussian Elimination (GE) algorithm — the running example of our paper — may need to account for the representation of the matrix, whether to compute and return the determinant or rank, how and whether search for pivot, etc. Furthermore, current architectures demand more and more frequent tweaks which, in general, cannot be done by the compiler because the tweaking often involves domain knowledge. A survey [3] of Gaussian elimination implementations in an industrial package Maple found 6 different aspects and 35 different implementations of the algorithm, as well as 45 implementations of directly related algorithms. We can manually write each of these implementations optimizing for particular aspects and using cut-and-paste to "share" similar pieces of code. We can write a very generic GE procedure that accounts for

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 256–274, 2005.

all the aspects with appropriate abstractions [16]. The abstraction mechanisms however – be they procedure, method or a function call – have a significant cost, especially for high-performance numerical computing [3].

A more appealing approach is generative programming [7,33,25,29,17,36]. The approach is not without problems, e.g., making sure that the generated code is well-formed. This is a challenge in string-based generation systems, which generally do not offer such guarantees and therefore make it very difficult to determine which part of the generator is at fault when the generated code cannot be parsed. Other problems is preventing accidental variable capture (so-called hygiene [21]) and ensuring the generated code is well-typed. Lisp-style macros, Scheme hygienic macros, camlp4 preprocessor [9], C++ template metaprogramming, Template Haskell [8] solve some of the above problems. Of the widely available maintainable languages, only MetaOCaml [2,23] solves all the above problems including the well-typing of the generated code [32,30].

But more difficult problems remain. Is the generated code optimal? Do we still need post-processing to eliminate common subexpressions and fold constants, remove redundant bindings? Is the generator readable, resembling the original algorithm, and extensible? Are the aspects truly modular? Can we add another aspect to it or another instance of the existing aspect without affecting the existing ones? Finally, can we express domain-specific knowledge, e.g., one should not attempt to use full division when dealing with matrices of exact integers, nor is it worthwhile to use full pivoting on a matrix over $\mathbb{Q}$.

MetaOCaml is *generative*: generated code can only be treated as a black box: it cannot be inspected and it cannot be post-processed (i.e., no intensional analysis). This approach gives a stronger equational theory [31], and avoids the danger of creating unsoundness [30]. Furthermore, intensional code analysis essentially requires one to insert both an optimizing compiler and an automated theorem proving system into the code generating system [28,18,4,34]. While this is potentially extremely powerful and an exciting area of research, it is also extremely complex, which means that it is currently more error-prone and difficult to ascertain the correctness of the resulting code.

Therefore, in MetaOCaml, code must be generated just right (see [30] for many simple examples). For more complex examples, new techniques are necessary, e.g., abstract interpretation [20]. But more problems remain [6]: generating binding statements ("names"), especially when generating loop bodies or conditional branches; making continuation-passing style (CPS) code clear. Many authors understandably shy away from CPS code as it quickly becomes unreadable. But this is needed for proper name generation. The problems of compositionality of code generators, expressing dependencies among them and domain-specific knowledge remain.

In this paper we report on progress of solving these problems using GE as our running example. Specifically, our contributions:

– Extending a let-insertion, memoizing monad of [12,20] for generating control structures such as loops and conditionals. The extension is non-trivial

because of control dependencies and because let-insertion, as we argue, is a control effect on its own.
- Implementation of the `doM`-notation (patterned after `do`-notation of Haskell) to make monadic code readable.
- Use of functors (including higher-order functors) to modularize the generator, express aspects (including results of various types) and *assure composability of aspects* even for aspects that use state and have to be accounted in many places in the generated code.
- Use functor type sharing constraints to encode domain-specific knowledge.

The rest of this paper is structured as follows: The next section introduces code generation in MetaOCaml, the problem of name generation, and continuation-passing style (CPS) as a general solution. We also introduce the monad and the issues of generating control statements. Section 3 describes the use of parametrized modules of OCaml to encode all of the aspects of the Gaussian Elimination algorithm family in completely separate, independent modules. We briefly discuss related work in section 4. We then outline the future work and conclude. Appendices give samples of the generated code (which is available in full at [5]).

2 Generating Binding Statements, CPS, and Monad

We build code generators out of primitive ones using code generation combinators. MetaOCaml, as an instance of a multi-stage programming system [30], provides exactly the needed features: to construct a code expression, to combine them, and to execute them. Figure 1 shows the simplest code generator `one`, as well as more complex generators.

```
let one = .<1>. and plus x y = .<.~x + .~y>.
let simplest_code = let gen x y = plus x (plus y one) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
⟹.<fun x_1 -> fun y_2 -> (x_1 + (y_2 + 1))>.
let simplest_param_code plus one = let gen x y = plus x (plus y one) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
let param_code1 plus one =
  let gen x y = plus (plus y one) (plus x (plus y one)) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
let param_code1' plus one =
  let gen x y = let ce = (plus y one) in  plus ce (plus x ce) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
param_code1' plus one
⟹.<fun x_1 -> fun y_2 -> ((y_2 + 1) + (x_1 + (y_2 + 1)))>.
```

Fig. 1. Code generation and combinators. ⟹ under an expression shows the result of its evaluation

We use MetaOCaml brackets `.<...>.` to generate code expressions, i.e., to construct future-stage computations. We use escapes `.~` to perform an immediate

code generating computation *while* we are building the future-stage computation. The immediate computation in `simplest_code` is the evaluation of the function `gen`, which in turn applies `plus`. The function `gen` receives code expressions `.<x>.` and `.<y>.` as arguments. At the generating stage, we can manipulate code expressions as (opaque) values. The function `gen` returns a code expression, which is inlined in the place of the escape. MetaOCaml can print out code expressions, so we can see the final generated code. It has no traces of `gen` and `plus`: their applications are done at the generation stage.

The final MetaOCaml feature, `.!` (pronounced "run") executes the code expression: `.! simplest_code` is a function of two integers, which we can apply: `(.! simplest_code) 1 2`. The original `simplest_code` is not a function on integers – it is a code expression.

To see the benefit of code generation, we notice that we can easily parameterize our code, `simplest_param_code`, and use it to generate code that operates on integers, floating point numbers or booleans – in general, any domain that implements `plus` and `one`.

The generator `param_code1` has two occurrences of `plus y one`, which may be quite a complex computation and so we would rather not do it twice. We may be tempted to rely on the compiler's common-subexpression elimination optimization. When the generated code is very complex, however, the compiler may overlook common subexpressions. Or the subexpressions may occur in such an imperative context where the compiler might not be able to determine if lifting them is sound. So, being conservative, the optimizer will leave the duplicates as they are. We may attempt to eliminate subexpressions as in `param_code1'`. However, the result of `param_code1' plus one` still exhibits duplicate sub-expressions. Our `let`-insertion optimization saved the computation at the generating stage. We need a combinator that inserts the `let` expression in the generat*ed* code. We need a combinator `letgen` to be used as `let ce = letgen (plus y one) in plus ce (plus x ce)` yielding the code like `.<let t = y + 1 in t + (x + t)>.`. But that seems impossible because `letgen exp` has to generate the expression `.<let t = exp in body>.` but `letgen` does not have the `body` yet. The body needs a temporary identifier `.<t>.` that is supposed to be the result of `letgen` itself. Certainly `letgen` cannot generate only part of a let-expression, without the `body`, as all generated expressions in MetaOCaml are well-formed and complete.

The key is to use continuation-passing style (CPS). Its benefits were first pointed out by [1] in the context of partial evaluation, and extensively used by [12,20] for code generation. Now, `param_code2 plus one` gives us the desired code.

```
let letgen exp k = .<let t = .~exp in .~(k .<t>.)>.
let param_code2 plus one =
 let gen x y k = letgen (plus y one) (fun ce -> k(plus ce (plus x ce)))
 and k0 x = x
 in .<fun x y -> .~(gen .<x>. .<y>. k0)>.
param_code2 plus one
⟹.<fun x_1 -> fun y_2 -> let t_3 = (y_2 + 1) in (t_3 + (x_1 + t_3))>.
```

Comparison of the code that did let-insertion at the generating stage
`let ce = (plus y one) in  plus ce (plus x ce)`
with the corresponding code inserting let at the generated code stage
`letgen (plus y one) (fun ce -> k (plus ce (plus x ce)))`
clearly shows the difference between direct-style and CPS code. What was `let ce = init in ...` in direct style became `init' (fun ce -> ...)` in CPS. For one thing, `let` became "inverted". For another, what used to be an expression that yields a value, `init`, became an expression that takes an extra argument, the continuation, and invokes it. The differences look negligible in the above example. In larger expressions with many let-forms, the number of parentheses around `fun` increases, the need to add and then invoke the `k` continuation argument become increasingly annoying. The inconvenience is great enough for some people to explicitly avoid CPS or claim that numerical programmers (our users) cannot or will not program in CPS. Clearly a better notation is needed.

The `do`-notation of Haskell [27] shows that it is possible to write CPS code in a conventional-looking style. The `do`-notation is the notation for monadic code [24]. Not only can monadic code represent CPS [13], it also helps in composability by offering to add different layers of effects (state, exception, non-determinism, etc) to the basic monad [22] in a controlled way.

A monad [24] is an abstract data type representing computations that yield a value and may have an *effect*. The data type must have at least two operations, `return` to build trivial effect-less computations and `bind` for combining computations. These operations must satisfy *monadic laws*: `return` being the left and the right unit of `bind` and `bind` being associative. Figure 2 defines the monad used throughout the present paper and shows its implementation.

```
type ('v,'s,'w) monad = 's -> ('s -> 'v -> 'w) -> 'w
let ret (a :'v) : ('v,'s,'w) monad = fun s k -> k s a
let bind a f = fun s k -> a s (fun s' b -> f b s' k)
let fetch s k = k s s  and  store v s k = k (v::s) ()

let k0 s v = v
let runM m = m [] k0

let l1 f = fun x -> doM { t <-- x; f t}
let l2 f = fun x y -> doM { tx <-- x; ty <-- y; f tx ty}

let retN a = fun s k -> .<let t = .~a in .~(k s .<t>.)>.

let ifL test th el = ret .< if .~test then .~th else .~el >.
let ifM test th el = fun s k ->
  k s .< if .~(test s k0) then .~(th s k0) else .~(el s k0) >.
```

Fig. 2. Our monad

Our monad represents two kinds of computational effects: reading and writing a computation-wide state, and control effects. The latter are normally associated

with exceptions, forking of computations, etc. – in general, whenever a computation ends with something other than invoking its natural continuation in the tail position. In our case the control effects manifest themselves as code generation.

In Figure 2, the monad (yielding values of the type `v`) is implemented as a function of two arguments: the state (of type `s`) and the continuation. The continuation receives the current state and the value, and yields the answer of the type `w`. The monad is polymorphic over the three type parameters. Other implementations are possible. Except for the code in Figure 2, the rest of our code treats the monad as a truly abstract data type. The implementation of the basic monadic operations `ret` and `bind` is conventional and clearly satisfies the monadic laws. Other monadic operations construct computations that do have specific effects. Operations `fetch` and `store v` construct computations that read and write the state. In our case the state is a list (of polymorphic variants), which models an open discriminated union, as we shall see later.

The operation `retN a` is the let-insertion operation, whose simpler version we called `letgen` earlier. It is the first computation with a control effect: indeed, the result of `retN a` is *not* the result of invoking its continuation `k`. Rather, its result is a `let` code expression. Such a behavior is symptomatic of control operators (in particular, `abort`).

Finally, `runM` runs our monad, that is, performs the computation of the monad and returns its result, which in our case is the code expression. We run the monad by passing it the initial state and the initial continuation `k0`. We can now re-write our `param_code2` example of the previous section as `param_code3`.

```
let param_code3 plus one =
  let gen x y = bind (retN (plus y one)) (fun ce ->
                ret (plus ce (plus x ce)))
  in .<fun x y -> .~(runM (gen .<x>. .<y>.))>.
let param_code4 plus one =
  let gen x y = doM { ce <-- retN (plus y one);
                      ret (plus ce (plus x ce)) }
  in .<fun x y -> .~(runM (gen .<x>. .<y>.))>.
let ifM' test th el = doM {
  testc <-- test; thc <-- th; elc <-- el;
  ifL testc thc elc}
let gen a i = ifM' (ret .<(.~i) >= 0>.)
                   (retN .<Some (.~a).(.~i)>.) (ret .<None>.)
 in .<fun a i -> .~(runM (gen .<a>. .<i>.))>.
⟹.<fun a_1 i_2 ->let t_3 = (Some a_1.(i_2)) in if (i_2 >= 0) then t_3 else None>.
let gen a i = ifM (ret .<(.~i) >= 0>.)
                  (retN .<Some (.~a).(.~i)>.) (ret .<None>.)
 in .<fun a i -> .~(runM (gen .<a>. .<i>.))>.
⟹.<fun a_1 i_2 ->if (i_2 >= 0) then let t_3 = (Some a_1.(i_2)) in t_3 else None>.
```

That does not seem like much of an improvement. With the help of camlp4 pre-processor, we introduce the `doM`-notation [5], patterned after the `do`-notation of Haskell. The function `param_code4`, written in the `doM`-notation, is equivalent

to `param_code3` – in fact, the camlp4 preprocessor will convert the former into the latter. And yet, `param_code4` looks far more conventional, as if it were indeed in direct style.

We can write operations that generate code other than let-statements, e.g., conditionals: see `ifL` in Figure 2. The function `ifL`, albeit straightforward, is not as general as we wish: its arguments are already generated pieces of code rather than monadic values. We "lift it", see `ifM'`. We define functions `l1`, `l2`, `l3` (analogues of `liftM`, `liftM2`, `liftM3` of Haskell) to make such a lifting generic. However we also need another `ifM` function, with the same interface (see Figure 2). The difference between them is apparent: in the code above with `ifM'`, the let-insertion happened *before* the if-expression, that is, before the test that the index `i` is positive. If `i` turned out negative, `a.(i)` would generate an out-of-bound array access error. On the other hand, the code with `ifM` accesses the array only when we have verified that the index is non-negative. This example makes it clear that the code generation (such as the one in `retN`) is truly an effect and we have to be clear about the sequencing of effects when generating control constructions such as conditionals. The form `ifM` handles such effects correctly. We need similar operators for other OCaml control forms: for generating case-matching statements and `for`- and `while`-loops.

3 Aspects and Functors

The monad represents finer-scale code generation. We need tools for larger-scale modularization; we can use any abstraction mechanisms we want to structure our code generators, as long as none of those abstractions infiltrate the generated code.

While the Object-Oriented Design community has acquired an extensive vocabulary for describing modularity ideas, the guiding principles for modular designs has not changed since they were first articulated by Parnas [26] and Dijkstra [10]: information hiding and separation of concerns. To apply these principles to the study of Gaussian Elimination, we need to understand what are the changes between different implementations, and what concerns need to be addressed. We also need to study the degree to which these concerns are independent. A study of Gaussian Elimination [3] shows that the following variations occur:

1. **Domain:** In which (algebraic) domain do the matrix elements belong to. Sometimes the domains are very specific (e.g., $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p$ and floating point numbers), while in other cases the domains were left generic, e.g., multivariate polynomials over a field. In the roughly 85 pieces of code surveyed [3] 20 different domains were encountered.
2. **Container:** Whether the matrix is represented as an array of arrays, a one-dimensional array, a hash table, a sparse matrix, etc., and whether indexing is done in C or Fortran style. Additionally, if a particular representation had a special mechanism for efficient row exchanges.

3. **Output choices:** Whether just the reduced matrix, or additionally the rank, the determinant, and the pivoting matrix are to be returned. In the larger algorithm family, routines like Maple's `LinearAlgebra:-LUDecomposition` have up to $2^6 + 2^5 + 2^2 = 100$ outputs.
4. **Fraction-free:** Whether the Gaussian Elimination algorithm is allowed to use unrestricted division, or only exact (remainder-free) division.
5. **Pivoting:** Whether to use no, column-wise, or full pivoting.
6. **Augmented Matrices:** Whether all or only some columns of the matrix participate in elimination. We currently do not implement this aspect.

In addition to the above variations, there are two aspects that recur frequently:

1. **Length measure:** For stability reasons (numerical or coefficient growth), if a domain possesses an appropriate length measure, this is sometimes used to choose an "optimal" pivot.
2. **Normalization and zero-equivalence:** Whether the arithmetic operations of the domain give normalized results, and whether a specialized zero-equivalence routine is to be used.

These are separated out from the others as they are cross-cutting concerns: in the case of the length measure, a property of the domain will influence the pivoting method *if* pivoting is to be performed.

The simplest parametrization is to make the domain abstract. As it turns out, we need the following to exist in our domains: 0, 1, $+$, $*$, (unary and binary) $-$, at least *exact* division, normalization, and potentially a relative size measure. The simplest case of such domain abstraction is `param_code1` in Fig. 1. There, code-generators such as `plus` and `one` were passed as arguments. We need far more than two parameters, so we have to group them. Instead of the grouping offered by regular records, we use OCaml *structures* (i.e., modules) so we can take advantage of extensibility, type abstraction and constraints, and especially parameterized structures (*functors*). We define the type of the domain, the signature `DOMAIN`, which different domains must satisfy:

```
module type DOMAIN = sig
  type v    type 'a vc = ('a,v) code
  type kind (* Field or Ring ? *)
  val zero : 'a vc   val one : 'a vc
  val plus : 'a vc -> 'a vc -> ('a vc, 's, 'w) monad
  (* times, minus, uminus, div elided for brevity *)
  val better_than : ('a vc -> 'a vc ->
      (('a,bool) code, 's, 'w) monad) option
  val normalizerf : (('a,v -> v) code ) option
end
module IntegerDomain : DOMAIN = struct
  type v = int  type kind = domain_is_ring
  type 'a vc = ('a,v) code
  let zero = .< 0 >.  and one = .< 1 >.
```

```
    let plus x y = ret .<.~x + .~y>.
    let better_than = Some (fun x y -> ret .<abs .~x > abs .~y >. )
    let normalizerf = None
    ...
  end
```

The types above are generally lifted twice: once from the value domain `v` to the code domain `'a vc`, and once more from values to monadic computations `('a vc, 's, 'w) monad`.

One particular domain instance is `IntegerDomain`. The notation `module IntegerDomain : DOMAIN` makes the compiler verify that our `IntegerDomain` is indeed a `DOMAIN`, that is, satisfies the required signature. The constraint `DOMAIN` may be omitted; in that case, the compiler will verify the type when we try to use that structure as a `DOMAIN`. In any case, the errors such as missing "methods" or methods with incorrect types will be caught statically, even *before* any code generation takes place. The abstract type `domain_is_ring` encodes a semantic constraint that the full division is not available. While the `DOMAIN` type may have looked daunting to some, the implementation is quite straightforward. Other domains such as `float` and arbitrary precision exact rational numbers `Num.num` are equally simple.

Parameterizing by the kind of container representing a matrix is almost as straightforward. Our containers are parametric over a `DOMAIN`, i.e., functors from a `DOMAIN` module to the actual implementation of a container. The functor signature `CONTAINER2D` specifies that a container must provide functions `dim1` and `dim2` to extract the dimensions, functions `get` and `set` to generate container getter and setters, the cloning generator `copy` and functions that generate code for row and column swapping. The inclusion of these functions in the signature of all containers makes it simpler to optimize the relevant functions depending on the actual representation of the container while not burdening the users of containers with efficiency details.

The use of a `functor` for making a container parametric is fairly straightforward. More interesting is the aspect of what to return from the GE algorithm. One could create an algebraic data type (as was done in [3]) to encode the various choices: the matrix, the matrix and the rank, the matrix and the determinant, the matrix, rank and determinant, and so on. This is wholly unsatisfying as we know that for any single use, only one of the choices is ever possible, yet any routine which calls the generated code must deal with these unreachable options. Instead we use a module type with an *abstract* type `res` for the result type; different instances of the signature set the result type differently. Given below is this module type and one instantiation, which specifies the output of a GE algorithm as a 3-tuple `contr * Det.outdet * int` of the U-factor, the determinant and the rank.

```
module type OUTPUT = sig
  type contr  type res
  module D : DETERMINANT   module R : RANK   module P : TRACKPIVOT
  val make_result : ('a,contr) code ->
```

```
    (('a,res) code,
     [> `TDet of 'a D.lstate | `TRan of 'a R.lstate | `TPivot of 'a P.lstate]
       list, ('a,'w) code) monad
end
module OutDetRank(Dom:DOMAIN)(C: CONTAINER2D)
    (Det : DETERMINANT with type indet = Dom.v and type outdet = Dom.v)
    (Rank : RANK) = struct
  module Ctr = C(Dom)
  type contr = Ctr.contr
  type res = contr * Det.outdet * int
  module D = Det   module R = Rank   module P = DiscardPivot
  let make_result b = doM { det  <-- D.fin ();  rank <-- R.fin ();
    ret .< ( .~b, .~det, .~rank ) >. }
end
```

As is apparent from the output choices, several different quantities *may* need to be tracked in a particular GE implementation. We therefore need to be able to conditionally generate variables representing the tracking state, and weave in corresponding tracking code. We may need to (independently) keep track of the rank, the determinant and the permutation list. The tracking state variables then become part of the *state* that is tracked by our monad. To have all this choice when needed, and yet have our code be modular and composable as well as ensuring that the generated code does not contain any abstraction artifacts, it is important to make this state modular. For example,

```
module type DETERMINANT = sig
  type indet  type outdet  type 'a lstate
  type tdet = outdet ref
  val decl : unit ->
    (unit, [> `TDet of 'a lstate ] list, ('a,'b) code) monad
  val upd_sign : unit ->
    (('a,unit) code, [> `TDet of 'a lstate ] list, ('a,'b) code) monad
  ...
end
```

to track determinant we should be able to generate code for: defining variables used for tracking (`decl`), updating the sign or the absolute value of the determinant, converting the tracking state to the final determinant value of the type `outdet`. GE of a floating-point matrix with no determinant tracking uses the instantiation of `DETERMINANT` where `outdet` is `unit` and all the functions of that module generate no code. For integer matrices, we have to track some aspects of the determinant, even if we don't output it. The determinant tracking aspect is complex because tracking variables, if any, are to be declared at the beginning of GE; the sign of the determinant has to be updated on each row or column permutation; the value of the determinant should be updated per each pivoting. We use `lstate` to pass the tracking state, e.g., a piece of code for the value of the type `Dom.v ref`, among various determinant-tracking functions. The `lstate` is a part of the overall monadic state. Other aspects, e.g., rank tracking, may use the monadic state for passing of rank tracking variables. To be able to compose

determinant and rank tracking functors – each of which may (or may not) use the monadic state for passing its own data – we make extensive use of open records (a list of polymorphic variants appeared to be the easiest way to implement such a union, in a purely functional way). This lets us freely compose determinant-tracking, rank-tracking, and other aspects.

The GE generator functor itself is parameterized by the domain, container, pivoting policy (full, row, nonzero, no pivoting), update policy (with either 'fraction-less' or full division), and the result specification. Some of the argument modules such as `PIVOT` are functors themselves (parameterized by the domain, the container, and the determinant functor). The sharing constraints express obvious constraints on the instantiation of `Gen`, for example, pivoting, determinant etc. components all use the same domain. It must be stressed that all structures (i.e., module instances) are stateless, and so we never have to worry that different aspect functors (such as `CONTAINER2D` and `PIVOT`) are instantiated with different but type-compatible instances of `DOMAIN`. That is, we are not concerned

```
module Gen(Dom: DOMAIN)(C: CONTAINER2D)(PivotF: PIVOT)
        (Update: UPDATE with type baseobj = Dom.v and type ctr = C(Dom).contr)
        (Out: OUTPUT with type contr = C(Dom).contr and type D.indet = Dom.v
                      and type 'a D.lstate = 'a Update.D.lstate) = struct
   module Ctr = C(Dom)
   module Pivot = PivotF(Dom)(C)(Out.D)
   let gen =
   let zerobelow b r c m n brc =
       let innerbody i = doM {
           bic <-- Ctr.get b i c;
           whenM (l1 LogicCode.not (LogicCode.equal bic Dom.zero ))
               (seqM (retLoopM (Idx.succ c) (Idx.pred m)
                         (fun k -> Update.update b r c i k) )
                     (Ctr.set b i c Dom.zero)) } in
       doM {
             seqM (retLoopM (Idx.succ r) (Idx.pred n) innerbody)
                  (Update.update_det brc) } in
   let dogen a = doM {
         r <-- Out.R.decl ();
         c <-- retN (liftRef Idx.zero);
         b <-- retN (Ctr.mapper Dom.normalizerf (Ctr.copy a));
         m <-- retN (Ctr.dim1 a);
         n <-- retN (Ctr.dim2 a);
         () <-- Update.D.decl ();
         () <-- Out.P.decl ();
         seqM
           (retWhileM (LogicCode.and_ (Idx.less (liftGet c) m)
                   (Idx.less (liftGet r) n) )
              ( doM {
              rr <-- retN (liftGet r);
              cc <-- retN (liftGet c);
              pivot <-- l1 retN (Pivot.findpivot b rr m cc n);
```

```
                seqM (retMatchM pivot (fun pv ->
                         seqM (zerobelow b rr cc m n pv)
                              (Out.R.succ ()) )
                         (Update.D.zero_sign () ))
                     (Code.update c Idx.succ) } ))
            (Out.make_result b) } in
    .<fun a -> .~(runM (dogen .<a>.)) >.
end
```

at all about value sharing. Aspects such as determinant tracking may be stateful so that the determinant update code have access to the determinant tracking variables declared previously. But that state is handled via the monadic state. As we have shown, open unions make the overall monadic state compositional with respect to the state of various aspects.

In addition to the "regular" type sharing constraints shown in the `Gen` functor, there are also "semantic" sharing constraints, shown in the following structure of the `UPDATE` signature:

```
module DivisionUpdate
  (Dom:DOMAIN with type kind = domain_is_field)
  (C:CONTAINER2D)
  (Det:DETERMINANT with type indet=Dom.v) = struct ... end
```

This structure implements an update policy of using `Dom.div` operation without restrictions – which is possible only if the domain has such an unrestricted operation. A domain such as the integer domain may still provide `Dom.div` of the same type, but that operation may only be used when we are sure that the division is exact. Our type sharing constraint expresses such domain-specific knowledge: instantiating `DivisionUpdate` with `IntegerDomain` leads to a compile-time error, when compiling the *generator* code. Thus, in some cases we can use module types for "semantic" constraints that cannot normally be expressed via the types of module members.

```
module GenIV5 = Gen(IntegerDomain)
  (GenericVectorContainer)(FullPivot)
  (FractionFreeUpdate(IntegerDomain)(GenericVectorContainer)(IDet))
  (OutDetRank(IntegerDomain)(GenericVectorContainer)(IDet)(Rank))
module GenFA1 = Gen(FloatDomain)
  (GenericArrayContainer)(RowPivot)
  (DivisionUpdate(FloatDomain)(GenericArrayContainer)(NoDet(FloatDomain)))
  (OutJustMatrix(FloatDomain)(GenericArrayContainer)(NoDet(FloatDomain)))
```

We can instantiate the `Gen` functor as shown above and inspect the generated code, e.g., by printing `GenFA1.gen`. The code can then be "compiled" as `!. GenFA1.gen` or with off-shoring. The code for `GenIV5` (Appendix A) shows full pivoting, determinant and rank tracking. The code for all these aspects is fully inlined; no extra functions are invoked and no tests other than those needed by the GE algorithm itself are performed. The GE function returns a triple `int array * int * int` of the U-factor, determinant and the rank. The code generated by `GenFA1` (Appendix B) shows absolutely no traces of determinant tracking: no declaration of spurious variables, no extra tests, etc. The code

appears as if the determinant tracking aspect did not exist at all. The generated code for the above and other instantiations of `Gen` can be examined at [5]. The website also contains benchmark code and timing comparisons.

4 Related and Future Work

The monad in this paper is similar to the one described in [12,20]. However the latter papers used only `retN` and fixpoints (for generation-time iterations). This paper does not involve monadic fixpoints because the generator is not recursive, but heavily relies on monadic operations for generating conditionals and loops.

`Blitz++` [33] and C++ template meta-programming in general similarly eliminate levels of abstraction. With traits and concepts, some domain-specific knowledge can also be encoded. However overhead elimination critically depends on full inlining of all methods by the compiler, which has been reported to be challenging to insure. Furthermore, all errors (such as type errors and concept violation errors, i.e., composition errors) are detected only when compiling the generated code. It is immensely difficult to correlate errors (e.g., line numbers) to the ones in the generator itself.

ATLAS [36] is another successful project in this area. However they use much simpler weaving technology, which leads them to note that *generator complexity tends to go up along with flexibility, so that these routines become almost insurmountable barriers to outside contribution*. Our results show how to surmount this barrier, by building modular, composable generators. SPIRAL [28] is another such even more ambitious project. But SPIRAL does intentional code analysis, relying on a set of code transformation "rules" which make sense, but which are not proved to be either complete or confluent. The strength of both of these project relies on their platform-specific optimizations performed via search techniques, something we have not attempted here.

The highly parametric version of our Gaussian Elimination is directly influenced by the generic implementations available in Axiom [16] and Aldor [35]. Even though the Aldor compiler frequently can optimize away a lot of abstraction overhead, it does not provide any guarantees that it will do so, unlike our approach.

We should also mention early work [15] on automatic specialization of mathematical algorithms. Although it can eliminate some overhead from a very generic implementation (e.g., by inlining aspects implemented as higher-order functions), specialization cannot change the type of the function and cannot efficiently handle aspects that communicate via a private shared state.

The paper [14] describes early simple experiments in *automatic* and manual staging, and the multi-level language based on an annotated subset of Scheme (which is untyped and has no imperative features). The generated code requires post-processing to attain efficiency.

We are looking into encapsulating staging annotations into just a few functors, so that the rest of the code (in particular, the `Gen` functor that puts it all

together) should be annotation-free and thus can be used as is in a one-stage environment (pure OCaml) as well as in a multi-stage environment (generating extensions). The one-stage code is a good baseline for benchmarks and regression tests. Obtaining a generating extension from properly modularized OCaml code (along the lines of our `Gen`) is an exciting area of our future research.

To the best of our knowledge, nobody has yet used functors to abstract code generators, or even mixed functors and multi-stage programming.

We plan to further investigate the connection between delimited continuations and our implementations of code generators like `ifM`. As well, by using some additional syntactic sugar (for `ifM`, `whileM`, etc.), the available notation should be even more direct-style, and potentially clearer. We also would like to extend our monad to a monad transformer.

There are many more aspects which can also be handled: Input variations (augmented matrices), error reporting (i.e. asking for the determinant of a non-square matrix), memory hierarchy issues, loop-unrolling [6], warnings when zero-testing is undecidable and a value is only probabilistically non-zero, etc. The larger program family of LU decompositions contains more aspects still.

5 Conclusion

In this paper we have demonstrated numerical code extensively parameterized by complex aspects at no run-time overhead. The combination of stateless functors and structures, and our monad with the compositional state makes aspects freely composable without having to worry about value aliasing. The only constraints to compositionality are the typing ones plus the constraints we specifically impose, including semantic constraints (e.g., rings do not have full division).

There is an interesting relation with aspect-oriented code [19]: in AspectJ, aspects are (comparatively) lightly typed, and are post-facto extensions of an existing piece of code. Here aspects are weaved together "from scratch" to make up a piece of code/functionality. One can understand previous work to be more akin to dynamically typed aspect weaving, while we have started investigating statically typed one.

Acknowledgments

We wish to thank Cristiano Calgano for his help in adapting camlp4 for use with MetaOCaml. Many helpful discussions with Walid Taha are very appreciated. We are grateful to anonymous reviewers for many helpful suggestions.

References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, 1992.

2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, LNCS. Springer-Verlag, 2003.
3. Jacques Carette. Gaussian Elimination: a case study in efficient genericity with MetaOCaml, 2005. submitted.
4. Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Rothe. Lapack for clusters project: An example of self adapting numerical software. *Hawaii International Conference on System Sciences HICSS-37*, 2004.
5. Source code. `http://www.cas.mcmaster.ca/~carette/metamonads/`.
6. Albert Cohen, Sebastien Donadio, Mari'a Jesu's Garzara'n, Cristoph Herrmann, and David Padua. In search for a program generator to implement generic transformations for high-performance computing. MetaOCaml Workshop, October 2004.
7. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
8. Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer, 2003.
9. Daniel de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/camlp4/manual/`, Jan. 2002.
10. Edsger W. Dijkstra. On the role of scientific thought. published as [11].
11. Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.
12. Jason L. Eckhardt, Roumen Kaiabachev, Kedar N. Swadi, Walid Taha, and Oleg Kiselyov. Practical aspects of multi-stage programming. Rice University Techical Report TR05-451, `http://www.cs.rice.edu/~taha/publications/preprints/2004-02-16.pdf`, February 2004.
13. Andrzej Filinski. Representing monads. In *POPL*, pages 446–457, 1994.
14. Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.
15. Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization*, 1995.
16. Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer Verlag, 1992.
17. III John V.W. Reynders and Julian C. Cummings. The POOMA framework. *Comput. Phys.*, 12(5):453–459, 1998.
18. Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.
19. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

20. Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 249–258, New York, NY, USA, 2004. ACM Press.
21. Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LISP and Functional Programming*, pages 151–161, 1986.
22. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, 1995. ACM Press.
23. MetaOCaml. `http://www.metaocaml.org`.
24. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
25. David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.
26. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
27. Simon Peyton Jones et al. *The Revised Haskell 98 Report*. Cambridge Univ. Press, 2003. Also on `http://haskell.org/`.
28. Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.
29. Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
30. Walid Taha. *Multi-Stage Programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
31. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival. In *PEPM*, pages 34–43, 2000.
32. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, June 1997. ACM Press.
33. Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
34. Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.
35. Stephen M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfennig, editors, *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer Verlag, 2003.
36. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

Appendix A

The code generated for `GenIV5`, fraction-free GE of the integer matrix represented by a flat vector, full pivoting, returning the `U`-factor, the determinant and the rank.

```
# val resIV5 : ('a,
   Funct4.GenIV5.Ctr.contr ->
   Funct4.OutDetRank(Funct4.IntegerDomain)(Funct4.GenericVectorContainer)
                    (Funct4.IDet)(Funct4.Rank).res) code =
  .<fun a_405 ->
   let t_406 = (ref 0) in let t_407 = (ref 0) in
   let t_408 = arr = (Array.copy a_405.arr) (a_405) in
   let t_409 = a_405.m in let t_410 = a_405.n in
   let t_411 = (ref 1) in let t_412 = (ref 1) in
   while (((! t_407) < t_409) && ((! t_406) < t_410)) do
    let t_413 = (! t_406) in let t_414 = (! t_407) in
    let t_415 = (ref (None)) in
    let t_435 =
     begin  (* full pivoting *)
      for j_431 = t_413 to (t_410 - 1) do
       for j_432 = t_414 to (t_409 - 1) do
        let t_433 = (t_408.arr).((j_431 * t_408.m) + j_432) in
        if (not (t_433 = 0)) then
         (match (! t_415) with
          | Some (i_434) ->
             if ((abs (snd i_434)) > (abs t_433)) then
              (t_415 := (Some ((j_431, j_432), t_433))) else ()
          | None -> (t_415 := (Some ((j_431, j_432), t_433))))
        else ()
       done
      done;
      (match (! t_415) with
       | Some (i_416) ->  (* swapping of columns *)
          if ((snd (fst i_416)) <> t_414) then begin
           let a_424 = t_408.arr and nm_425 = (t_408.n * t_408.m)
           and m_426 = t_408.m in
           let rec loop_427 =
            fun i1_428 -> fun i2_429 ->
              if (i2_429 < nm_425) then
               let t_430 = a_424.(i1_428) in
               a_424.(i1_428) <- a_424.(i2_429);
               a_424.(i2_429) <- t_430;
               (loop_427 (i1_428 + m_426) (i2_429 + m_426))
              else () in
           (loop_427 t_414 (snd (fst i_416)));
           (t_412 := (~- (! t_412))) (* adjust the sign of det *)
          end else (); (* swapping of rows elided *)
          (Some (snd i_416))
       | None -> (None))
```

```
 end in
(match t_435 with
 | Some (i_436) ->
    begin (* elimination loop *)
     for j_437 = (t_413 + 1) to (t_410 - 1) do
      if (not ((t_408.arr).((j_437 * t_408.m) + t_414) = 0)) then begin
       for j_438 = (t_414 + 1) to (t_409 - 1) do
        (t_408.arr).((j_437 * t_408.m) + j_438) <-
         ((((t_408.arr).((j_437 * t_408.m) + j_438) * (* elided *)
       done;
       (t_408.arr).((j_437 * t_408.m) + t_414) <- 0
      end else ()
     done; (t_411 := i_436)
    end;
    (t_406 := ((! t_406) + 1)) (* advance the rank *)
 | None -> (t_412 := 0));
(t_407 := ((! t_407) + 1))
done;
(t_408,
 if ((! t_412) = 0) then 0 (* adjust the sign of the determinant *)
 else if ((! t_412) = 1) then (! t_411)
 else (~- (! t_411)), (! t_406))>.
```

Appendix B

The code generated for `GenFA1`, GE of the floating point matrix represented by a 2D array, row pivoting, returning just the `U`-factor.

```
# val resFA1 : ('a,
   Funct4.GenFA1.Ctr.contr ->
   Funct4.OutJustMatrix(Funct4.FloatDomain)(Funct4.GenericArrayContainer)
                       (Funct4.NoDet(Funct4.FloatDomain)).res) code =
  .<fun a_1 ->
   let t_2 = (ref 0) in let t_3 = (ref 0) in
   let t_5 = (Array.map (fun x_4 -> (Array.copy x_4)) (Array.copy a_1)) in
   let t_6 = (Array.length a_1.(0)) in
   let t_7 = (Array.length a_1) in
   while (((! t_3) < t_6) && ((! t_2) < t_7)) do
    let t_8 = (! t_2) in let t_9 = (! t_3) in
    let t_10 = (ref (None)) in
    let t_16 =
     begin  (* row pivoting *)
      for j_13 = t_8 to (t_7 - 1) do
       let t_14 = (t_5.(j_13)).(t_9) in
       if (not (t_14 = 0.)) then
        (match (! t_10) with
         | Some (i_15) ->
            if ((abs_float (snd i_15)) < (abs_float t_14)) then
             (t_10 := (Some (j_13, t_14)))
```

```
          else ()
        | None -> (t_10 := (Some (j_13, t_14))))
     else ()
    done;
    (match (! t_10) with
     | Some (i_11) -> (* swapping of rows *)
        if ((fst i_11) <> t_8) then begin
         let t_12 = t_5.(t_8) in
         t_5.(t_8) <- t_5.(fst i_11);
         t_5.(fst i_11) <- t_12; () end else ();
        (Some (snd i_11))
     | None -> (None))
   end in
  (match t_16 with
   | Some (i_17) ->
      begin (* elimination loop, elided *) end;
      (t_2 := ((! t_2) + 1))
   | None -> ());
  (t_3 := ((! t_3) + 1))
done;
t_5>.
```

Implicitly Heterogeneous Multi-stage Programming*

Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi, and Walid Taha

Rice University, Houston, TX 77005, USA

Abstract. Previous work on semantics-based multi-stage programming (MSP) language design focused on *homogeneous* designs, where the generating and the generated languages are the same. Homogeneous designs simply add a hygienic quasi-quotation and evaluation mechanism to a base language. An apparent disadvantage of this approach is that the programmer is bound to both the expressivity and performance characteristics of the base language. This paper proposes a practical means to avoid this by providing specialized translations from subsets of the base language to different target languages. This approach preserves the homogeneous "look" of multi-stage programs, and, more importantly, the static guarantees about the generated code. In addition, compared to an explicitly heterogeneous approach, it promotes reuse of generator source code and systematic exploration of the performance characteristics of the target languages.

To illustrate the proposed approach, we design and implement a translation to a subset of C suitable for numerical computation, and show that it preserves static typing. The translation is implemented, and evaluated with several benchmarks. The implementation is available in the online distribution of MetaOCaml.

1 Introduction

Multi-stage programming (MSP) languages allow the programmer to use abstraction mechanisms such as functions, objects, and modules, without having to pay a runtime overhead for them. Operationally, these languages provide quasi-quotation and `eval` mechanisms similar to those of LISP and Scheme. In addition, to avoid accidental capture, bound variables are always renamed, and values produced by quasi-quotes can only be de-constructed by `eval`. This makes reasoning about quoted terms as programs sound [18], even in the untyped setting. Several type systems have been developed that statically ensure that all programs generated using these constructs are well-typed (c.f. [19,1]).

Currently, the main examples of MSP languages include MetaScheme [7], MetaML [20], MetaOCaml [10], and Metaphor [11]. They are based, respectively, on Scheme, Standard ML, OCaml, and Java/C#. In all these cases,

* Supported by NSF ITR-0113569 "Putting Multi-Stage Annotations to Work", Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers" and NSF ITR-0205303 "Building Practical Compilers Based on Adaptive Search."

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 275–292, 2005.

the language design is *homogeneous*, in that quoted values are fragments of the base language. Homogeneity has three distinct advantages. First, it is convenient for the language designer, as it often reduces the size of the definitions needed to model a language, and makes extensions to arbitrary stages feasible at little cost. Second, it is convenient for the language implementor, as it allows the implementation of the base language to be reused: In all three examples above, as in LISP and Scheme implementations, the eval-like construct calls the underlying implementation. In the case of MetaOCaml, the MetaOCaml compiler and the bytecode runtime system can be used to execute generated programs at runtime. Third, it is convenient for the programmer, as it requires learning only a small number of new language constructs.

While the homogeneous approach described above has its advantages, there are situations in which the programmer may wish to take advantage of the capability of other compilers that are only available for other languages. For example, very well-developed, specialized compilers exist for application domains such as numerical computing, embedded systems, and parallel computation.

At first glance, this situation might suggest a need for *heterogeneous* quotation mechanisms, where quoted terms can contain expressions in a different language. Indeed, this approach has been used successfully for applications such as light-weight components [8,9], FFT [6], and computer graphics [5]. But heterogeneous quotation mechanisms also introduce two new complications:

1. How do we ensure that the generated program is statically typed?
2. How do we avoid restricting a generator to a particular target language?

One approach to addressing the first issue is to develop specialized two-level type systems. This means the language designer must work with type systems and semantics that are as big as both languages combined. Another possibility is to extend meta-programming languages with dependent type systems [13] and thus give the programmers the ability to write data-types that encode the abstract syntax of only *well-typed* object-language terms. Currently, such type systems can introduce significant notational overhead for the programmer, as well as requiring familiarity with unfamiliar type systems.

In principle, the second problem can be avoided by parameterizing the generators themselves by constructs for the target language of choice. However, this is likely to reduce the readability of the generators, and it is not clear how a quotation mechanism can be used in this setting. Furthermore, to ensure that the generated program is statically typed, we would, in essence, need to parameterize the static type of the generator by a description of the static type system of the target language. The typing infrastructure needed is likely to be far beyond what has gained acceptance in mainstream programming.

1.1 Contributions

This paper proposes a practical approach to avoiding the two problems, which can be described as *implicitly heterogeneous* MSP. In this approach, the pro-

grammer does not need to know about the details of the target-language representation. Those details are supplied by the meta-language designer once and for all and invoked by the programmer through the familiar interface used to execute generated code. This is achieved by the language implementers providing specialized translations from subsets of the base language to different target languages. Thus, the homogeneous "look" of homogeneous MSP is preserved. An immediate benefit is that the programmer may not have to make any changes to existing generators to target different languages. Additionally, if the translation itself is type preserving, the static guarantee about the type correctness of generated code is maintained.

The proposed approach is studied in the case when the target language is C. After a brief introduction to MSP (Section 2), we outline the details of what can be described as an *offshoring* translation that we have designed and implemented. The design begins by identifying the precise subset of the target language that we wish to make available to the programmer (Section 3). Once that is done, the next challenge is to identify an appropriate subset in the base language that can be used to represent the target subset.

Like a compiler, offshoring translations are provided by the language implementor and not by the programmer. But the requirements on offshoring translators are essentially the opposite of those on compilers (or compiling translators, such as Tarditi's [21]): First, offshoring is primarily concerned with the target, not the source language. The most expressive translation would cover the full target language, but not necessarily the source language. In contrast, a compiler must cover the source but not necessarily the target language. Second, the translation must be a direct mapping from source to target, and not a complex, optimizing translation. The direct connection between the base and target representations is essential for giving the programmer access to the target language.

To ensure that all generated programs are well typed, we show that the offshoring translation is type preserving (Section 4). This requires being explicit about the details of the type system for the target language as well as the source language, but is manageable because both subsets are syntactically limited. Again, this is something the language designer does once and benefits any programmer who uses the offshoring translation.

Having an offshoring translation makes it easier for the programmer to experiment with executing programs either in OCaml or C (using different C compilers). We present a detailed analysis of changes in performance behavior for a benchmark of dynamic programming algorithms (Section 5). Not surprisingly, C consistently outperforms the OCaml bytecode compiler. We also find that in several cases, the overhead of marshalling the results from the OCaml world to the C world can be a significant bottleneck, especially in cases where C is much faster than the OCaml bytecode compiler. And while C outperforms the OCaml native code compiler in many cases, there are exceptions.

Due to space limitations, the paper only formalizes the key concepts that illustrate the main ideas. Auxiliary definitions and proofs can be found in an extended version of the paper available online [4].

2 Multi-stage Programming

MSP languages [20,16] provide three high-level constructs that allow the programmer to break down computations into distinct stages. These constructs can be used for the construction, combination, and execution of code fragments. Standard problems associated with the manipulation of code fragments, such as accidental variable capture and the representation of programs, are hidden from the programmer (cf. [16]). The following minimal example illustrates MSP programming in MetaOCaml:

```
let rec power n x = if n=0 then .<1>. else .< .~x * .~(power (n-1) x)>.
let power3 = .! .<fun x -> .~(power 3 .<x>.)>.
```

Ignoring the staging constructs (brackets `.<`e`>.`, escapes `.~`e, as well as run `.!` e) the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step simply returns a function that would invoke the `power` function every time it gets invoked with a value for x. In contrast, the staged version builds a function that computes the third power directly (that is, using only multiplication). To see how the staging constructs work, we can start from the last statement in the code above. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not, because the outer brackets contain an escaped expression that still needs to be evaluated. Brackets mean that we want to construct a future stage computation, and escapes mean that we want to perform an immediate computation *while* building the bracketed computation. In a multi-stage language, these constructs are not hints, they are imperatives. Thus, the application e `.<x>.` must be performed even though `x` is still an uninstantiated symbol. The expression `power 3 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` first results in the equivalent of

```
.! .<fun x -> x*x*x*1>.
```

Once the argument to run (`.!`) is a code fragment that has no escapes, it is compiled and evaluated, and returns a function that has the same performance as if we had explicitly coded the last declaration as:

```
let power3 = fun x -> x*x*x*1
```

Applying this function does not incur the unnecessary overhead that the unstaged version would have had to pay every time `power3` is used.

3 Offshoring for Numerical Computation in C

This section presents an example of an offshoring translation aimed at supporting implicitly heterogeneous MSP for basic numerical computation. The method for defining the types and the syntax is presented, along with the resulting formal definitions. The mapping from base to target is only described informally. It bears

repeating that the programmer will not need to write her program generators to explicitly produce the target language presented in this section. Rather, the interface to generating target programs is written using the source-language syntax (Section 3.3); however, the programmer does need to be aware of the target subset definitions to be able to express her algorithms in a way that makes the code they produce amenable to translation. The section concludes by presenting the details of the programmer's interface to offshoring, and discussing the practical issue of marshalling values between the runtime systems of the base and target languages.

3.1 Types for Target Subset

The types in the target C subset include:

1. Base numerical types int, char and double.
2. One- and two- dimensional arrays of these numerical types. One-dimensional arrays are represented as C arrays. Two-dimensional arrays are implemented as an array of pointers (C type *[]). While this representation is less efficient than two-dimensional arrays in C, it was chosen because it allows for a simpler translation, since it is a better semantic match with MetaOCaml arrays. OCaml array *types* do not explicitly declare their size, so it is not always possible to obtain a valid two-dimensional array type in C from a type of two-dimensional arrays in OCaml.
3. Functions that take base types or arrays of base types as arguments and return base type values. This subset does not include function pointers, functions with variable number of arguments, or functions that return void.

The C subset types are described by the following BNF:

$$\begin{array}{lll} \textit{Base types} & b \in & \{\mathsf{int}, \mathsf{double}, \mathsf{char}\} \\ \textit{Array types} & a ::= & b\,[\,] \mid *\,b\,[\,] \\ \textit{Types} & t ::= & b \mid a \\ \textit{Funtypes} & f ::= & b\,(t_0, \ldots, t_n) \end{array}$$

3.2 Syntax for Target Subset

Figure 1 presents the BNF of the target C subset. The set is essentially a subset of C that has all the basic numerical and array operations, first order functions, and structured control flow operators. The target subset is not determined in a vacuum but also involves considering what can be expressed naturally in the source language. The main restrictions we impose on this subset are:

1. All declarations are initialized. This restriction is caused by the fact that OCaml does not permit uninitialized bindings for variables, and representing uninstantiated declarations in OCaml can add complexity.
2. No unstructured control flow statements (i.e., goto, break, continue and fall-through non-defaulted switch statements). This restriction is motivated by the lack of equivalent unstructured control flow operators in OCaml. For the same reason, the increment operations are also limited.

Type keyword	t	$\in$	$\{\mathsf{int}, \mathsf{double}, \mathsf{char}\}$
Constant	c	$\in$	$Int \cup Float \cup Char$
Variable	x	$\in$	X
Declaration	d	$::=$	$t\ x = c \mid t\ x[n] = \{c^*\} \mid t\ \mathsf{*}x[n] = \{x^*\}$
Arguments	a	$::=$	$t\ x \mid t\ x[\,] \mid t\ \mathsf{*}x[\,]$
Unary operator	$f^{(1)}$	$\in$	$\{\texttt{(float)}, \texttt{(int)}, \texttt{cos}, \texttt{sin}, \texttt{sqrt}\}$
Binary operator	$f^{[2]}$	$\in$	$\{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&\&}, \texttt{||}, \texttt{\&}, \mid, \wedge, \texttt{<<}, \texttt{>>}, \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\}$
For loop op.	opr	$:=$	$\texttt{<=} \mid \texttt{>=}$
Expression	e	$::=$	$c \mid x \mid f^{(1)}\ e \mid e\ f^{[2]}\ e \mid x\ (e^*) \mid x[e] \mid x[e][e] \mid e\ \mathsf{?}\ e : e$
Incr.expression	i	$::=$	$x\texttt{++} \mid x\texttt{--}$
Statement	s	$::=$	$e \mid \mathsf{return}\ e \mid \mid \{d^*; s^*\} \mid x{=}e \mid x[e]{=}e \mid x[e][e]{=}e$ $\mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e)\ s \mid \mathsf{for}(x = e; x\ opr\ e; i)\ s$ $\mid \mathsf{switch}\ (e)\ \{w^* \mathsf{default:}\ s\}$
Switch branch	w	$::=$	$\mathsf{case}\ c\mathsf{:}\ s\ \mathsf{break;}$
Fun. decl.	g	$::=$	$t\ x\ (a^*)\{d^*; s^*\}$
Program	p	$::=$	$d^*; g^*$

Fig. 1. Grammar for the C target

3. Two-dimensional arrays are represented by an array of pointers (e.g., int **x[]) instead of standard two-dimensional arrays.
4. For-loops are restricted to the most common case where the initializer is a single assignment to a variable, and the mutator expression is simply increment or decrement by one. This covers the most commonly used C idiom, and matches the OCaml `for` loops.
5. No do-while loop commands. OCaml does not have control flow statements that correspond naturally to a do-while loop.
6. Return statements are not permitted inside switch and for statements. A return can only appear at the end of a block in a function declaration or in a terminal positions at both branches of the if statement. This restriction is enforced by the type system (See Appendix A).

While this subset is syntactically restricted compared to full C, it still provides a semantically expressive first-order language. Many missing C constructs can be effectively simulated by the ones included: for example, arithmetical mutation operators (e.g., +=) can be simulated by assignments. Similarly do-while loops can be simulated with existing looping constructs. Since staging in MetaOCaml gives the programmer complete control over what code is generated, this imposes little burden on the programmer. This subset is particularly well-supported by many industrial-strength compilers. Giving the programmer safe access to such compilers is the main purpose of implicitly heterogeneous MSP.

3.3 Types in Base Language

We now turn to the base language representation of the target language subset: the types in the OCaml subset must match those of the C subset. OCaml

$$
\begin{array}{lll}
\textit{Constant} & \hat{c} \in & \textit{Int} \cup \textit{Bool} \cup \textit{Float} \cup \textit{Char} \\
\textit{Variable} & \hat{x} \in & \hat{X} \\
\textit{Unary op.} & \hat{f}^{(1)} \in & \{\texttt{cos}, \texttt{sin}, \texttt{sqrt}, \texttt{float\_of\_int}, \texttt{int\_of\_float}\} \\
\textit{Limit op..} & \hat{f}^{(2)} \in & \{\texttt{min}, \texttt{max}\} \\
\textit{Binary op.} & \hat{f}^{[2]} \in & \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}, \texttt{**}, \texttt{mod}, \texttt{land}, \texttt{lor}, \texttt{lxor}, \\
 & & \texttt{lsl}, \texttt{lsr}, \texttt{asr}, \texttt{=}, \texttt{<>}, \texttt{<}, \texttt{>}, \leq, \geq, \texttt{\&\&}, \texttt{||}\} \\
\textit{Expression} & \hat{e} ::= & \hat{x} \mid \hat{x}\ (\hat{e}^*) \mid \hat{f}^{(1)}\ \hat{e} \mid \hat{f}^{(2)}\ \hat{e}\ \hat{e} \mid \hat{e}\ \hat{f}^{[2]}\ \hat{e} \mid \texttt{ if } \hat{e} \texttt{ then } \hat{e} \texttt{ else } \hat{e} \\
 & & \mid \texttt{!}\hat{x} \mid \hat{x}\texttt{.(}\hat{e}\texttt{)} \mid \hat{x}\texttt{.(}\hat{e}\texttt{).(}\hat{e}\texttt{)} \\
\textit{Statement} & \hat{d} ::= & \hat{e} \mid \hat{d}; \hat{d} \mid \texttt{let } \hat{x} \texttt{ = } \hat{e} \texttt{ in } \hat{d} \mid \texttt{let } \hat{x} \texttt{ = ref } \hat{c} \texttt{ in } \hat{d} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make } \hat{c}\ \hat{c} \texttt{ in } \hat{d} \mid \hat{x}\texttt{.(}\hat{e}\texttt{)} \leftarrow \hat{e} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make\_matrix } \hat{c}\ \hat{c}\ \hat{c} \texttt{ in } \hat{d} \mid \hat{x}\texttt{.(}\hat{e}\texttt{).(}\hat{e}\texttt{)} \leftarrow \hat{e} \\
 & & \mid \hat{x}\texttt{:=}\hat{e} \mid \texttt{if } \hat{e} \texttt{ then } \hat{d} \texttt{ else } \hat{d} \mid \texttt{while } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \\
 & & \mid \texttt{for } \hat{x}\texttt{=}\hat{e} \texttt{ to } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \mid \texttt{for } \hat{x}\texttt{=}\hat{e} \texttt{ downto } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \\
 & & \mid \texttt{match } \hat{e} \texttt{ with } ((\hat{c} \rightarrow \hat{d})^* \mid \_ \rightarrow \hat{d}) \\
\textit{Program} & \hat{s} ::= & \lambda(x^*).(\hat{d} : \hat{b}) \mid \texttt{let } \hat{x} \texttt{ = } \hat{c} \texttt{ in } \hat{s} \mid \texttt{let } f(\hat{x}^*) \texttt{ = } \hat{d} \texttt{ in } \hat{s} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = ref } \hat{c} \texttt{ in } \hat{s} \mid \texttt{let } \hat{x} \texttt{ = Array.make } \hat{c}\ \hat{c} \texttt{ in } \hat{s} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make\_matrix } \hat{c}\ \hat{c}\ \hat{c} \texttt{ in } \hat{s}
\end{array}
$$

Fig. 2. OCaml Subset Grammar

base types `bool`, `int`, `char`, and `float` map to C int, char and double, (OCaml booleans are simply mapped to C integers). The OCaml reference type (`ref`) is used to model C variables of simple type. One- and two-dimensional arrays are represented by OCaml $\hat{b}$ `array` and $\hat{b}$ `array array` types. To reflect the different restrictions on them, we will also distinguish types for function arguments, variable types and function declarations. The resulting types are as follows:

$$
\begin{array}{lll}
\textit{Base} & \hat{b} & \in \{\texttt{int}, \texttt{bool}, \texttt{float}, \texttt{char}\} \\
\textit{Reference} & \hat{r} & ::= \hat{b}\ \texttt{ref} \\
\textit{Array} & \hat{a} & ::= \hat{b}\ \texttt{array} \mid \hat{b}\ \texttt{array array} \\
\textit{Argument} & \hat{p} & ::= \hat{b} \mid \hat{a} \\
\textit{Variables} & \hat{t} & ::= \hat{b} \mid \hat{r} \mid \hat{a} \\
\textit{Function} & \hat{u} & ::= (\hat{p}_0, \ldots, \hat{p}_n) \rightarrow \hat{b}
\end{array}
$$

3.4 Syntax for Base Language

The syntax of the source subset is presented in Figure 2. Semantically, this subset represents the first-order subset of OCaml with arrays and reference cells. We point out the most interesting syntactic features of the OCaml subset:

1. Syntactically, `let` bindings are used to represent C declarations. Bindings representing C function declarations are further restricted to exclude function declarations nested in the bodies of functions.
2. Assignments in C are represented by updating OCaml references or arrays.
3. We use a special syntactic form to indicate a top level entry function (analogous to C main).

3.5 What the Programmer Sees

The programmer is given access to offshoring simply by making MetaOCaml run construct (`.!`e) customizable. In addition to this standard form, the programmer can now write `.!{Trx.run_gcc}`e or `.!{Trx.run_icc}`e to indicate when offshoring should be used and whether the `gcc` or `icc` compilers, respectively, should be used to compile the resulting code. Additional arguments can also be passed to these constructs. For example, the programmer can write `{Trx.run_gcc}`e (resp. `{Trx.run_icc}`e) as follows:

```
.!{Trx.run_gcc with compiler = "cc"; compiler_flags="-g -O2"} ...
```

to use an alternative compiler `cc` with the flags `-g -O2`.

3.6 Marshalling and Dynamic Linking

The C and OCaml systems use different runtime representations for values. So, to use the offshored program, inputs and outputs must be marshalled from one representation to the other.

In our implementation of the translation described above, we generate a marshalling function for the top-level entry point of the offshored function. The marshalling function depends only on the type of the top-level function. First, OCaml values are converted to their C representations. The standard OCaml library provides conversions for base types. We convert arrays by allocating a new C array, converting each element of the OCaml array, and storing it in the C array. The C function is invoked with the marshalled C values. Its results are collected, and marshalled back into OCaml. To account for updates in arrays, the elements of the C are converted and copied back into the OCaml array.

Once a C program has been produced, the marshalling code is added, and the result is then compiled into a shared library (a `.so`) file. To load this shared library into MetaOCaml's runtime system, we extend Stolpmann's dynamic loading library for OCaml [14] to support dynamic loading of function values.

4 Type Preservation

Showing that an offshoring translation can preserve typing requires formalizing the type system for the target language, the source language, as well as the translation itself. Here we give a brief outline of the technical development.

The type system for top-level statements in OCaml programs is defined by the derivability of the judgment $\hat{\Gamma} \vdash \hat{p} : \hat{t}$ (Appendix B). Similarly, the type system for C programs is defined by a judgment $\Gamma \vdash g$ (Appendix A). We also provide the definition of translation functions (Appendix C). For example, $\langle\!|\hat{s}|\!\rangle = (g_1, \ldots, g_n, l_1, \ldots, l_m)$ translates a top-level OCaml program into a set of C variable declarations and function definitions, and $[\![\hat{\Gamma}]\!]$ translates the OCaml variable and function environment, $\hat{\Gamma}$, into the corresponding C environment.

Any valid term in the base language translates to a valid one in the target language. We define an operation $|\cdot|$ on variable declarations and function definitions that translates them into a C type environment(see [4]).

Theorem 1 (Type Preservation). *If* $\hat{\Gamma} \vdash \hat{s} : \hat{a_n} \rightarrow \hat{b}$ *and* $\langle\!|\hat{s}|\!\rangle = (g, l)$, *then* $[\![\hat{\Gamma}]\!] \cup |l| \cup |g| \vdash g$.

Proof. By induction over the height of first derivation. The details of the proof are presented in the extended version of the paper [4]. □

5 Effect on Performance Characteristics

This section summarizes some empirical measurements that we have gathered to evaluate the performance impact of offshoring. The questions we wanted these experiments to answer are: How does the performance of offshored code compare to that of the same code executed with *run*? Does marshalling impose a significant overhead on offshoring? As MetaOCaml currently only supports bytecode OCaml execution, would extending MetaOCaml to support native OCaml-style compilation be an acceptable alternative to offshoring?

5.1 Benchmarks

As a benchmark, we use a suite of staged dynamic programming algorithms. These algorithms were initially implemented to study how dynamic programming algorithms can be staged [15]. The benchmark consists of (both unstaged and staged) MetaOCaml implementations of the following algorithms [2]:

- `forward`, the forward algorithm for Hidden Markov Models. Specialization size (size of the observation sequence) is 7.
- `gib`, the Gibonacci function, a minor generalization of the Fibonacci function. This is not an example of dynamic programming, but gives rise to the same technical problems while staging. Specialization is for $n = 25$.
- `ks`, the 0/1 knapsack problem. Specialization is for size 32 (number of items).
- `lcs`, the least common subsequence problems. Specialization is for string sizes 25 and 34 for the first and second arguments, respectively.
- `obst`, the optimal binary search tree algorithm. Specialization is a leaf-node-tree of size 15.
- `opt`, the optimal matrix multiplication problem. Specialization is for 18 matrices.

To evaluate offshoring, we compare executing the result of these algorithms in MetaOCaml to executing the result in C using offshoring. [1]

[1] **Platform specs.** Timings were collected on a Pentium 4 machine (3055MHz, 8K L1 and 512K L2 cache, 1GB main memory) running Linux `2.4.20-31.9`. All experiments are fully automated and available online [3]. We report results based on version `DP_002/1.25` of the benchmark. It was executed using MetaOCaml version `308_alpha_020` bytecode compiler, Objective Caml version `3.08.0` native code compiler, and GNU's gcc version `2.95.3 20010315`, and Intel's icc version `8.0 Build 20031016Z` C compilers.

Table 1. Speedups and Break-Even Points from Offshoring to C

Name	Unstaged (ms)	Generate (ms)	Compile (ms)	Staged Run (μs)	Speedup	**Speedup'**	**BEP**
forward	0.20	1.1	34.	0.37	530x	**20.x**	**180**
gib	0.13	0.26	19.	0.12	990x	**5.3x**	**170**
ks	5.2	36.	8300.	1.4	3700x	**69.x**	**1600**
lcs	5.5	46.	6400.	5.1	1100x	**24.x**	**1100**
obst	4.2	26.	5300.	4.6	910x	**22.x**	**1300**
opt	3.6	66.	8700.	3.4	1100x	**44.x**	**2500**

Using GNU's `gcc`

Name	Unstaged (ms)	Generate (ms)	Compile (ms)	Staged Run (μs)	Speedup	**Speedup'**	**BEP**
forward	0.20	1.1	33.0	0.35	580x	**21.x**	**170**
gib	0.13	0.26	20.0	0.098	1200x	**6.4x**	**180**
ks	5.2	36.	8100.	1.5	3500x	**66.x**	**1600**
lcs	5.5	46.	6500.	5.3	1000x	**25.x**	**1200**
obst	4.2	26.	5400.	4.5	940x	**23.x**	**1300**
opt	3.6	66.	9300.	3.3	1100x	**46.x**	**2600**

Using Intel's `icc`

5.2 Comparing Offshoring with the OCaml Byte Code

Because of engineering issues with OCaml's support for dynamic loading, MetaOCaml currently extends only the bytecode compiler with staging constructs, but not the native code compiler. We therefore begin by considering the impact of offshoring in the bytecode setting.

Table 1 displays measurements for offshoring with both `gcc` and Intel's `icc`. The columns are computed as follows: **Unstaged** is the execution time of the unstaged version of a program. All times are averages, and are reported are in milliseconds (ms) unless otherwise stated. We measure execution times using MetaOCaml's standard library function `Trxtime.timenew`. This function repeatedly executes a program until the cumulative execution time exceeds 1 second and reports the number of iterations and the average execution time per iteration. **Generate** reports code generation times. Generation is considered the first stage in a two-stage computation. The second stage is called Staged Run and will be described shortly. **Compile** is the time needed to translate the generated program and compile it with `gcc` (or `icc` in the second table), and dynamically load the resulting binary. **Staged Run** reports the C program execution time, including marshalling. The binary is generated by calling the C compiler with the flags `-g -O2`. **Speedup** is computed as Unstaged divided by Staged Run. **Speedup'** is the ratio between Speedup with staging (without offshoring) and staging with offshoring. **BEP** is the break-even point, i.e., the number of executions for a piece of code after which the initial overhead of offshoring (generation and compilation) pays off. For example, we need at least 180 uses of `forward` compiled using `gcc` before it is cheaper to use the staged version than to just use the unstaged one.

In general, the results obtained for offshoring to C are encouraging. The ratio between speedups (**Speedup'**) is always greater than 1. But it should be

noted that the BEP's are higher than for non-offshored staged execution within MetaOCaml. This is primarily a result of the C compilation times being higher than OCaml bytecode compilation times.

5.3 Marshalling Overhead

To assess the impact of marshalling on the runtime performance, we compare the time to execute the benchmarks from within MetaOCaml against the time it took to execute the same functions in a C program with no marshalling.

Table 2 displays the numbers for the `gcc` and `icc` compilers. Each program is run multiple times externally using the same number of iterations obtained from the measurements for the **Unstaged Run** in Figure 1. The Unix `time` command is used to measure the time for each program. This time is then divided by the number of iterations to obtain an average execution time per program. The columns are computed as follows: **Marshalling Overhead(G)** and **Marshalling Overhead(I)** show the difference in microseconds between running each function from within MetaOCaml and running it externally using `gcc` and `icc` respectively. **Percentage(G)** and **Percentage(I)** show the marshalling overhead as a percentage of the total time spent on the marshalled computation.

In the `forward` example, we marshall a 7-element integer array. In `gib`, we marshall only two integers, and thus have a low marshalling overhead. But because the total computation time in this benchmark small, the marshalling overhead is high. In `ks`, we marshall a 32-element integer array, and in `lcs` benchmark, we marshall two 25- and 34-element character arrays, which accounts for the high marshalling overhead in these benchmarks. Since both these benchmarks perform a significant amount of computation, the proportion of time spent marshalling is, however, lower than `gib` or `forward`. Similarly, `obst` and `opt` have significant marshalling overheads since they must marshal 15-element floating-point and 18-element integer arrays.

While the percentages given by both `gcc` and `icc` are comparable, there is one benchmark `ks` which shows a large difference. This can be explained by the fact that `icc` was able to optimize much better than `gcc`, reducing computation time to the point that most of the time was spent in marshalling.

The data indicates that marshalling overhead is significant in this benchmark. We chose the current marshalling strategy for implementation simplicity, and

Table 2. Marshalling Overhead for `gcc` and Intel's `icc` Compiled Offshored Code

Name	Marshalling (G) Overhead (μs)	**Percentage (G)**	Marshalling (I) Overhead (μs)	**Percentage (I)**
forward	0.19	**50. %**	0.22	**63. %**
gib	0.11	**91. %**	0.089	**91. %**
ks	0.37	**26. %**	1.4	**96. %**
lcs	3.8	**74. %**	4.2	**79. %**
obst	0.61	**13. %**	1.5	**34. %**
opt	0.61	**17. %**	0.24	**7.4 %**

Table 3. Speed of Offshored vs. OCaml Native Compiled Code

Name	OCamlOpt (μs)	Tweaked (μs)	Best GCC (μs)	**Speedup**	**Speedup"**
forward	0.29	0.25	0.13	**2.2x**	**2.1x**
gib	0.017	0.0095	0.0096	**1.7x**	**0.95x**
ks	23.	0.61	1.0	**23.x**	**0.59x**
lcs	24.	3.1	1.2	**20.x**	**3.0x**
obst	33.	10.	2.7	**12.x**	**3.9x**
opt	24.	4.9	2.8	**8.3x**	**1.7x**

Using GNU's `gcc`

Name	OCamlOpt (μs)	Tweaked (μs)	Best ICC (μs)	**Speedup**	**Speedup"**
forward	0.29	0.25	0.13	**2.2x**	**1.8x**
gib	0.017	0.0095	0.0091	**1.8x**	**1.1x**
ks	23.	0.61	0.061	**380.x**	**10.x**
lcs	24.	3.1	1.1	**22.x**	**2.2x**
obst	33.	10.	2.9	**11.x**	**3.6x**
opt	24.	4.9	3.1	**7.8x**	**1.6x**

Using Intel's `icc`

not for efficiency. These results suggest that it would be useful to explore an alternative marshalling strategy that allows sharing of the same data-structures across the OCaml/C boundary.

5.4 Comparing Offshoring with the OCaml Native Compiler

One motivation for offshoring is that standard implementations of high-level languages are unlikely to compile automatically generated programs as effectively as implementations for lower-level languages such as C. The figures reported so far relate to the MetaOCaml bytecode compiler. How does offshoring perform against the native code compiler?

Table 3 displays measurements for execution times of the generated programs when executed using the native-code compiler for OCaml and the best runtime when running the result of their translation in C. The columns are computed as follows: **OCamlOpt** is the time for executing programs compiled with the `ocamlopt` compiler with the options `-unsafe -inline 50`. **Tweaked** are execution times for programs hand-optimized post-generation by the following transformations[2]: currying the arguments to functions, replacing the uses of the polymorphic OCaml operators `min` and `max` by their monomorphic versions, and moving array initializations outside the generated functions. The same compiler options as in **OCamlOpt** were used. **Best GCC** (**Best ICC**) is the execution time for generated and offshored code with the best performing optimization level (`-O[0-4]`). **Speedup** is the ratio of the **OCamlOpt** times to the **Best GCC** (**Best ICC**) times, and **Speedup"** is the ratio of the **Tweaked** execution times **Best GCC** (**Best ICC**).

Without hand-optimizing the generated code, offshoring always outperforms the native code compiler. After hand-optimizing the generated code, the situa-

[2] Thanks to Xavier Leroy for pointing out that these changes would improve the relative performance of the OCaml native code compiler.

tion is slightly different. In two cases, the OCaml native code compiler outperforms `gcc`. In fact, for the `ks` benchmark, the OCaml compiler does much better than `gcc`.

The tweaks that improved the relative performance of OCaml's native code compiler draw attention to an important issue in designing an offshoring transformation and in interpreting speedups such as the ones presented here: Speedups are sensitive to the particular representation that we chose in the base language, and the real goal is to give the programmer the ability to generate specific programs in the target language. Speedups reported here give one set of data-points exhibiting the potential of offshoring.

6 Further Opportunities for Offshoring

The particular offshoring translation presented here is useful for building multi-stage implementations of several standard algorithms. But there are other features of C that have distinct performance characteristics, and which the programmer might wish to gain access to. We expect that the translation presented here can be expanded fairly systematically. In particular, an offshoring translation can be viewed as an inverse of a total map from the target to the source language. The backwards map would be a kind of OCaml semantics for the constructs in the target language. With this insight, several features of C can be represented in OCaml as follows: Function pointers can be handled naturally in a higher order language, although the representative base language subset would have to be restricted to disallow occurrences of free variables in nested functions. Such free variables give rise to the need for closures when compiling functional languages, and our goal is not to compile, but rather to give the programmer access to the notion of function pointers in C. If the programmer wishes to implement closures in C, it can be done explicitly at the OCaml level. For control operators, we can use continuations, and more generally, a monadic style in the source language, but the source would have similar restrictions on free variables to ensure that no continuation requires closures to be (automatically) constructed in the target code. Targeting struct and union types should be relatively straightforward using OCaml's notion of datatypes. Dynamic memory management, bit manipulation, and pointer arithmetic can be supported by specially marked OCaml libraries that simulate operations on an explicit memory model in OCaml. Arbitrary dimension arrays are a natural extension of the two-dimensional representation that we have already addressed.

7 Conclusion

We have proposed the idea of implicitly heterogeneous MSP as a way of combining the benefits of the homogeneous and heterogeneous approaches. In particular, generators need not become coupled with the syntax of the target language, and the programmer need not be bound to the performance characteristics of the base language. To illustrate this approach, we target a subset of C suitable for

numerical computation, and take MetaOCaml as the base language. Experimental results indicate that this approach yields significantly better performance as compared to the OCaml bytecode compiler, and often better performance than the OCaml native code compiler. We have implemented this extension in MetaOCaml. A fully automated version of our performance measurement suite has been implemented and made available online [3]. An offshoring translation targeting FORTRAN is under development.

Acknowledgments. We would like to thank John Mellor-Crummey and Gregory Malecha, who read drafts of this paper and gave us several helpful suggestions. We would also like to thank the reviewers and the editors for their detailed and constructive comments. One of reviewers asked whether offshoring can be used to produce highly-optimized libraries such as those used in the SPIN model checker or in Andrew Goldberg's network optimization library. We thank the reviewer for the intriguing suggestion, and we hope that further experience with offshoring will allow us to answer this question.

References

1. Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Ml-like inference for classifiers. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
2. Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 14 edition, 1994.
3. Dynamic Programming Benchmarks. Available online from `http://www.metaocaml.org/examples/dp`, 2005.
4. Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming (extended version). Available online from: `http://www.cs.rice.edu/~taha/publications/preprints/2005-06-28-TR.pdf`, June 2005.
5. Conal Elliott, Sigbjørn Finne, and Oege de Moore. Compiling embedded languages. In *[17]*, pages 9–27, 2000.
6. Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
7. Robert Glück and Jesper Jørgensen. Multi-level specialization (extended abstract). In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *LNCS*, pages 326–337. Springer-Verlag, 1999.
8. Sam Kamin. Standard ML as a meta-programming language. Technical report, Univ. of Illinois Computer Science Dept, 1996. Available at `www-faculty.cs.uiuc.edu/~kamin/pubs/`.
9. Samuel N. Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components i: Source-level components. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 49–64, London, UK, 2000. Springer-Verlag.
10. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

11. Gregory Neverov and Paul Roe. *Towards a Fully-reflective Meta-programming Language*, volume 38 of *Conferences in Research and Practice in Information Technology*. ACS, Newcastle, Australia, 2005. Twenty-Eighth Australasian Computer Science Conference (ACSC2005).
12. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
13. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
14. Gerd Stolpmann. Dl- ad-hoc dynamic loading for OCaml. Available from `http://www.ocaml-programming.de/packages/documentation/dl/`.
15. Kedar Swadi, Walid Taha, and Oleg Kiselyov. Staging dynamic programming algorithms, April 2005. Unpublished manuscript, available from `http://www.cs.rice.edu/~taha/publications.html`.
16. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [12].
17. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
18. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, Boston, 2000. ACM Press.
19. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
20. Walid Taha and Tim Sheard. Mult-stage programming with explicit annotations. In *Proceedings of Symposium on Partial Evaluation and Semantics Based Program manipulation*, pages 203–217. ACM SIGPLAN, 1997.
21. David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

A Type System for the C Fragment

The type system for the target language is defined for a particular Σ that assigns types to constants and operators.

Expressions ($\Gamma \vdash e : t$)

$$\frac{}{\Gamma \vdash c : b}\,\texttt{Const}\ (c : b \in \Sigma) \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\,\texttt{Var} \qquad \frac{f^{(1)} : b\ f(b_1) \in \Sigma \quad \Gamma \vdash e : b_1}{\Gamma \vdash f^{(1)}\ (e) : b}\,\texttt{Op1}$$

$$\frac{f^{[2]} : b\ f^{[2]}(b_1, b_2) \in \Sigma \quad \Gamma \vdash e_1 : b_1 \quad \Gamma \vdash e_2 : b_2}{\Gamma \vdash e_1\ f^{[2]}\ e_2 : b}\,\texttt{Op2}$$

$$\frac{\Gamma(x) = b \quad (t_i)^{i \in \{\ldots n\}} \quad \{\Gamma \vdash e_i : t_i\}^{i \in \{\ldots n\}}}{\Gamma \vdash x\ (e_i)^{i \in \{\ldots n\}} : b}\,\texttt{FCall} \qquad \frac{\Gamma(x) = \mathsf{int}}{\Gamma \vdash x\texttt{++} : \mathsf{int}}\,\texttt{Inc} \qquad \frac{\Gamma(x) = \mathsf{int}}{\Gamma \vdash x\texttt{--} : \mathsf{int}}\,\texttt{Dec}$$

$$\frac{\Gamma(x) = b\,[\,] \quad \Gamma \vdash e : \mathsf{int}}{\Gamma \vdash x[e] : b}\,\texttt{Arr1} \qquad \frac{\Gamma(x) = *\,b[\,] \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash x[e_1][e_2] : b}\,\texttt{Arr2}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : t}\,\texttt{If}$$

Statements ($\Gamma \vdash s : t$)

To indicate which terms must or can include a return statement, we define a more general judgment $\Gamma \vdash s : r$, where r is defined as follows:

$$r ::= t \mid \bot$$

The r types indicate return contexts: if a statement is well-formed with respect to the return context $\bot$, no return statements are permitted in it. If the return context is t, the judgment ensures that the rightmost leaf of the statement is a return statement of type t. For example, the definition body of a function with return type t, must be typed with t as its return context.

$$\frac{\Gamma \vdash e : t_1}{\Gamma \vdash e : \bot}\texttt{EStat} \qquad \frac{\Gamma(x) = b \quad \Gamma \vdash e : b}{\Gamma \vdash x = e : \bot}\texttt{Assign} \qquad \frac{\Gamma(x) = b[\,] \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : b}{\Gamma \vdash x[e_1] = e_2 : \bot}\texttt{SetArr1}$$

$$\frac{\Gamma(x) = *b[\,] \quad \Gamma \vdash e_2 : \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_3 : b}{\Gamma \vdash x[e_1][e_2] = e_3 : \bot}\texttt{SetArr2} \qquad \frac{\Gamma \vdash e : \mathsf{int} \quad \Gamma \vdash s_1 : r \quad \Gamma \vdash s_2 : r}{\Gamma \vdash \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 : r}\texttt{IfS} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \mathsf{while}\ (e_1)\ s : \bot}\texttt{While}$$

$$\frac{\Gamma(x) = \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \mathsf{for}\ (x = e_1; e_2; x\texttt{++})\ s : \bot}\texttt{For} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{return}\ (e) : t}\texttt{Ret}$$

$$\frac{\Gamma \vdash c_n : \mathsf{int} \quad \{\Gamma \vdash s_i : \bot\}^{i\in\{\dots n\}} \quad \Gamma \vdash e : \mathsf{int} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \left\{ \begin{array}{l} \mathsf{switch}\ (e)\{ \\ \quad (\mathsf{case}\ c_i : s_i\ \mathsf{break};)^{i\in\{\dots n\}}\ \ \mathsf{default} : s\} \end{array} \right\} : \bot}\texttt{Sw} \qquad \frac{\left\{\Gamma \cup (d_j)^{j\in\{\dots m\}} \vdash s_i\right\}^{i\in\{\dots n-1\}} : \bot \quad \Gamma \cup (d_j)^{j\in\{\dots m\}} \vdash s_n : r}{\Gamma \vdash \{(d_j)^{j\in\{\dots m\}}; (s_n)^{i\in\{\dots n\}}\} : r}\texttt{Blk}$$

Function definition ($\Gamma \vdash f$)

$$\frac{\Gamma \cup (a_i)^{i\in\{\dots n\}} \vdash \{(d_j)^{j\in\{\dots m\}}; (s_k)^{k\in\{\dots z\}}\} : b}{\Gamma \vdash (b\ f\ \ (a_i)^{i\in\{\dots n\}}\ \ \{(d_j)^{j\in\{\dots m\}}; (s_k)^{k\in\{\dots z\}}\})}\texttt{TFun}$$

Program ($\Gamma \vdash \overline{g}$)

$$\frac{}{\Gamma \vdash \cdot}\texttt{Empty} \qquad \frac{\Gamma \vdash b\ f\ \ (a_i)^{i\in\{\dots n\}}\ \ s \quad \Gamma, b\ f\ \ (a_i)^{i\in\{\dots n\}} \vdash g}{\Gamma \vdash (b\ f\ (a_i)^{i\in\{\dots n\}}\ \ s); g}\texttt{ExtTop}$$

B Type System for the OCaml Fragment

We do not show the detailed definition of the typing judgment for the OCaml fragment due to lack of space (full definitions are available in the extended version [4]). Instead, we just note that the type system of the MetaOCaml fragment consists of three judgments: $\Gamma \vdash e : t$, for expressions ; $\Gamma \vdash d : \hat{t}$, for statements; $\Gamma \vdash \hat{s} : \hat{t}$, for programs.

C Offshoring Translation

This section formalizes the translation from the OCaml subset to C. OCaml types and expressions are translated directly into C expressions.

$\boxed{[\![\hat{t}]\!]}$ $\boxed{[\![\hat{e}]\!]}$

$$
\begin{array}{rcl}
[\![\texttt{int}]\!] & = & \texttt{int} \\
[\![\texttt{bool}]\!] & = & \texttt{int} \\
[\![\texttt{char}]\!] & = & \texttt{char} \\
[\![\texttt{float}]\!] & = & \texttt{double} \\
[\![b\ \texttt{array}]\!] & = & [\![b]\!]\ [\,] \\
[\![b\ \texttt{array array}]\!] & = & *[\![b]\!][\,]
\end{array}
\qquad
\begin{array}{rcl}
[\![\hat{c}]\!] & = & c \\
[\![\hat{x}]\!] & = & x \\
[\![\hat{x}(\hat{e_1}, \ldots, \hat{e_n})]\!] & = & x([\![\hat{e_1}]\!], \ldots, [\![\hat{e_n}]\!]) \\
[\![\hat{f}^{(1)}\,\hat{e}]\!] & = & [\![\hat{f}^{(1)}]\!][\![\hat{e}]\!] \\
[\![\hat{f}^{(2)}\,\hat{e_1}\,\hat{e_2}]\!] & = & [\![\hat{f}^{(2)}]\!][\![\hat{e_1}]\!][\![\hat{e_2}]\!] \\
[\![\hat{e_1}\,\hat{f}^{[2]}\,\hat{e_2}]\!] & = & [\![\hat{e_1}]\!][\![\hat{f}^{[2]}]\!][\![\hat{e_2}]\!] \\
[\![\texttt{if}\ \hat{e_1}\ \texttt{then}\ \hat{e_2}\ \texttt{else}\ \hat{e_3}]\!] & = & [\![\hat{e_1}]\!]?[\![\hat{e_2}]\!] : [\![\hat{e_3}]\!] \\
[\![!\hat{x}]\!] & = & x \\
[\![\hat{x}.(\hat{e})]\!] & = & x[[\![\hat{e}]\!]] \\
[\![\hat{x}.(\hat{e_1}).(\hat{e_2})]\!] & = & x[[\![\hat{e_1}]\!]][[\![\hat{e_2}]\!]]
\end{array}
$$

The statement subset of OCaml is translated to a pair $(\overline{d}, \overline{s})$, where $\overline{d}$ are declarations of variables that are bound in OCaml `let` expressions, and $\overline{s}$, the sequence of C statements that corresponds to OCaml statements. The translation for OCaml statements $\{\!|\hat{d}|\!\}$ is written with a *return context* which can be $\bot$ or $\top$.

$$\textit{Return context } a ::= \bot \mid \top$$

If the return context is $\bot$, the translation does not generate return statements at the leaves; on the other hand, if the return context is $\top$, bottoming out at a leaf OCaml expression produces the appropriate C return statement.

$\boxed{\{\!|\hat{d}|\!\}_a = (d, s)}$

$$\frac{}{\{\!|\hat{e}|\!\}_L = (\cdot, [\![\hat{e}]\!])} \qquad \frac{}{\{\!|\hat{e}|\!\}_R = (\cdot, \texttt{return}\ [\![\hat{e}]\!])} \qquad \frac{}{\{\!|\hat{x} := \hat{e}|\!\}_a = (\cdot, x = [\![\hat{e}]\!])} \qquad \frac{}{\begin{array}{c}\{\!|\hat{x}.(\hat{e_1}) \leftarrow \hat{e_2}|\!\}_a = \\ (\cdot, x[[\![\hat{e_1}]\!]] = [\![\hat{e_2}]\!])\end{array}}$$

$$\frac{}{\begin{array}{c}\{\!|\hat{x}.(\hat{e_1}).(\hat{e_2}) \leftarrow \hat{e_3}|\!\}_a = \\ (\cdot, x[[\![\hat{e_1}]\!]][[\![\hat{e_2}]\!]] = [\![\hat{e_3}]\!])\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\{\!|\left\{\begin{array}{l}\texttt{let}\ \hat{x} : \hat{t} = \hat{e} \\ \texttt{in}\ \hat{d}\end{array}\right\}|\!\}_a = ((([\![\hat{t}]\!]x; l), (x = [\![\hat{e}]\!]; s))}$$

$$\frac{\begin{array}{c}\{\!|\hat{d_1}|\!\}_L = (l_1, s_1) \\ \{\!|\hat{d_2}|\!\}_a = (l_2, s_2)\end{array}}{\{\!|\hat{d_1}; \hat{d_2}|\!\}_a = (l_1; l_2, s_1; s_2)} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\{\!|\texttt{let}\ \hat{x} : \texttt{ref}\ \hat{t} = \texttt{ref}\ \hat{c}\ \texttt{in}\ \hat{d}|\!\}_a = ((([\![\hat{t}]\!](c)x; l), (x = [\![c]\!]; s))}$$

$$\frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\begin{array}{c}\{\!|\left\{\begin{array}{l}\texttt{let} x : \hat{t}\ \texttt{array} = \\ \texttt{Array.make}\ \hat{c_1}\ \hat{c_2}\ \texttt{in}\ \hat{d}\end{array}\right\}|\!\}_a = \\ ([\![\hat{t}]\!]\ x[\,] = \{\overline{c_2}^{c_1\ times}\}, l\ ;\ s)\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s) \quad \overline{y_m}\ fresh}{\begin{array}{c}\{\!|\texttt{let}\ \hat{x} : \hat{t}\texttt{array}\ \texttt{array} = \texttt{Array.make\_matrix}\ \hat{e}\ \hat{c_2}\ \hat{c_3}\ \texttt{in}\ \hat{d}|\!\}_a = \\ \begin{array}{c}([\![\hat{t}]\!]\ y_m = \{\overline{[\![e]\!]^{c_2\ times}}\}^{c_1\ times}; \\ [\![\hat{t}]\!]\ *x[\,] = \{y_1, \ldots, y_{c_1}\}; l, s)\end{array}\end{array}}$$

$$\frac{\begin{array}{c}\{\!|\hat{d_1}|\!\}_a = (l_1, s_1) \\ \{\!|\hat{d_2}|\!\}_a = (l_2, s_2)\end{array}}{\begin{array}{c}\{\!|\texttt{if}\ (\hat{e})\ \texttt{then}\ \hat{d_1}\ \texttt{else}\ \hat{d_2}|\!\}_a = \\ (\cdot, \texttt{if}([\![\hat{e}]\!])\{l_1; s_1\}\ \texttt{else}\ \{l_2; s_2\})\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_L = (l, s)}{\begin{array}{c}\{\!|\texttt{while}\ (\hat{e})\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_a = \\ (\cdot, \texttt{while}([\![\hat{e}]\!])\{l; s\})\end{array}}$$

$$\frac{\{\!|\hat{d}|\!\}_L = (l, s)}{\begin{array}{c}\{\!|\texttt{for}\ \hat{x} = \hat{e_1}\ \texttt{to}\ \hat{e_2}\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_a = \\ (\texttt{int}\ x, \texttt{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{<=}s[\![\hat{e_2}]\!]; x\texttt{++})\{l; s\})\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_L = (l, s)}{\begin{array}{c}\{\!|\texttt{for}\ \hat{x} = \hat{e_1}\ \texttt{downto}\ \hat{e_2}\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_a \\ = (\texttt{int}\ x, \texttt{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{>=}[\![\hat{e_2}]\!]; x\texttt{--})\{l; s\})\end{array}}$$

$$\frac{\{\!|\hat{d_n}|\!\}_L\, n = (l_n, s_n) \quad \{\!|\hat{d}|\!\}_L = (l, s)}{\begin{array}{c}\{\!|\texttt{match}\ \hat{e}\ \texttt{with}\ \overline{\hat{c_n}\ \rightarrow\ \hat{d_n}}_n \mid \_ \rightarrow\ \hat{d}|\!\}_a = \\ \texttt{case}\ c_n : \{l_n; s_n; \texttt{break}\};\quad \texttt{default} : \{l; s\}\}\end{array}}$$

OCaml programs $\hat{s}$ are translated into a pair (g, l), where g is a sequence of function definitions and l is a sequence of variable declarations.

$\boxed{\langle\!|\, \hat{s} \,|\!\rangle = (g, l)}$

$$\frac{\langle\!|\, \hat{s} \,|\!\rangle = (g, l)}{\langle\!|\, \texttt{let}\ \hat{x} : \hat{t} = \hat{c}\ \texttt{in}\ \hat{s} \,|\!\rangle = (g, ([\![\hat{t}]\!]\ x\ =\ c; l))}$$

$$\frac{\langle\!|\, \hat{s} \,|\!\rangle = (g, l)}{\langle\!|\, \texttt{let}\ \hat{x} : \texttt{ref}\ \hat{t} = \texttt{ref}\ \hat{c}\ \texttt{in}\ \hat{s} \,|\!\rangle = (g,\ [\![\hat{t}]\!] x = c; l)}$$

$$\frac{\langle\!|\, \hat{s} \,|\!\rangle = (g, l)}{\langle\!|\, \left\{ \begin{array}{l} \texttt{let}\ \hat{x} : \hat{t}\ \texttt{array} = \\ \texttt{Array.make}\ \hat{c_1}\ \hat{c_2}\ \texttt{in}\ \hat{s} \end{array} \right\} \,|\!\rangle = (g,\ [\![\hat{t}]\!]\ x[\,] = \{\overline{c_2}^{\,c_1\ times}\}; l)}$$

$$\frac{\langle\!|\, \hat{d} \,|\!\rangle = (g, l) \qquad (y_i\ fresh)^{i \in \{1 \ldots c_1\}}}{\langle\!|\, \texttt{let}\ \hat{x} : \hat{t}\ \texttt{array array} = \texttt{Array.make\_matrix}\ \hat{e}\ \hat{c_2}\ \hat{c_3}\ \texttt{in}\ \hat{d} \,|\!\rangle_a = (g, \left([\![\hat{t}]\!]\ y_i = \{\overline{[\![e]\!]}^{c_2\ times}\} \right)^{i \in \{\ldots c1\}}; [\![\hat{t}]\!]\ *\ x[\,] = \{y_1, \ldots, y_{c_1}\}; l)}$$

$$\frac{\langle\!|\, \hat{s} \,|\!\rangle = (g, l) \quad \{\!|\hat{d}|\!\}_R = (l_d, s_d)}{\langle\!|\, \texttt{let}\ \hat{f}\ (\hat{x_i} : \hat{p_i})^{i \in \{\ldots n\}} : \hat{b} = \hat{d}\ \texttt{in}\ \hat{s} \,|\!\rangle = ([\![\hat{b}]\!] f\ ([\![p]\!]_i\ x_i)^{i \in \{\ldots n\}}\ \{l_d; s_d;\}; g, l)}$$

$$\frac{\{\!|\hat{d}|\!\}_R = (l, s)}{\langle\!|\, \lambda\ (\hat{x_i} : \hat{p_i})^{i \in \{\ldots n\}}\ .\ (\hat{d} : \hat{b}) \,|\!\rangle = (([\![\hat{b}]\!]\ \texttt{procedure}\ ([\![p_i]\!]\ x_i)^{i \in \{\ldots n\}}\ \{l; s\}), \cdot)}$$

Source-Level Optimization of Run-Time Program Generators*

Samuel Kamin, Barış Aktemur, and Philip Morton

University of Illinois at Urbana-Champaign, USA
{kamin, aktemur, pmorton}@cs.uiuc.edu

Abstract. We describe our efforts to use source-level rewriting to optimize run-time program generators written in Jumbo, a run-time program generation system for Java. Jumbo is a compiler written in *compositional* style, which brings the advantage that any program fragment can be abstracted out and compiled to an intermediate form. These forms can be put together at run-time to build complete programs. This principle provides a high level of flexibility in writing program generators. However, this comes at the price of inefficient run-time compilation. Using source-level transformations, we optimize the run-time generation of byte code from fragments, achieving speedups of 5–15%. We discuss the optimization process and give several examples.

1 Introduction

Jumbo[1,2,3,4] is a Java compiler with code quotation and anti-quotation for run-time program generation (RTPG). In this, it is similar to such systems as MetaML [5], MetaOCaml [6], 'C [7,8,9], and DynJava [10]. However, it has a unique design based on the principle that, if the static compiler is structured "compositionally," there need be only that one compiler — its back end can serve as the code-generating engine for RTPG. We have in the past [3] described the advantages of this approach, and will again briefly do so in Section 2. It does, however, possess one distinct disadvantage: inefficiency. This paper describes our on-going efforts to address this problem.

In each of the systems just mentioned, program generators are created by using a code quotation/anti-quotation syntax. For example, in Jumbo, the notation `$<while (x>0) 'Stmt(getBody())>$` indicates that, at run time, a while statement is to be generated with its body returned from the method call `getBody()`. Since the latter is not known at compile time, it is called a "hole." Quotation marks, `$<` and `>$`, can contain an entire compilation unit — an interface or list of classes — or a fragment as small as a single variable or constant. Quoted code cannot be compiled to Java virtual machine (JVM) code: either it has holes or it is not a full class and is therefore missing necessary context, such as field declarations. (Technically, it is legal to quote a complete compilation unit, without holes, but it is pointless, since it could be compiled statically.)

* Partial support for this work was received from NSF under grant CCR-0306221.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 293–308, 2005.

That fragments cannot be compiled all the way to bytecode does not mean they cannot be compiled at all. Consider the case of a quoted class definition with a hole where a method should go. In essence, we have a partial evaluation problem: The compiler has two inputs — the quoted class and the method — of which only the class is known at compile time. It is quite plausible that we might apply the compiler to the class and obtain a "residual compiler" that will receive the method and complete the compilation at run time. (The situation is symmetric in the arguments. When the compiler sees the quoted method, it has a similar problem, with two inputs — the method and its surrounding context — of which it sees only one.)

How can we partially evaluate a compiler applied to an incomplete fragment? The first point to note is that an ordinary compiler handles only compilation units; when presented with a smaller fragment, it gives a syntax error; partial evaluation cannot overcome that. The second point is that, given a compiler for a real language, even if we provided a compilation unit (with holes), it would be a practical impossibility to partially evaluate it mechanically.

In Jumbo, we address these problems in two ways. First, the Jumbo compiler is *compositional*. This means it is structured in such a way that small fragments are still meaningful to the compiler; they can be *partially* compiled, to an intermediate representation we call *Code*. The *Code* value of a compound fragment is a function solely of the *Code* values of its subfragments. Thus, in the example above, when the method definition is supplied at run time, it is supplied in its partially compiled form — not as source code or an abstract syntax tree (AST). This *Code* value is somehow placed inside the *Code* value for the class, and the result is compiled to JVM code.

Second, we have written a set of source-level transformations to optimize the compilation of fragments. These are the subject of this paper. In pursuing this strategy, we have also found that the compiler may need to be massaged to make it more susceptible to transformations, though we have yet much to learn about that process. The work is on-going; the results given here represent the current state of our compiler.

In the paper, we elaborate on each of the themes mentioned above. Section 2 explains what we wish to achieve with our system; to give a preview, it argues that the primary reason to insist on a single compiler is *not* to save development time, but rather to ensure a high level of *generality* in the tool we produce. Sections 3 and 4 discuss compositional compilation in general, and its use in Jumbo, respectively. Section 5 describes the analyses and transformations we have implemented and Section 6 gives examples and timing results. In Section 7, we discuss some of the difficulties presented by Java which have limited our success in optimization, and ways to overcome them. Section 8 gives our conclusions. Related work is noted throughout the paper, and is not segregated.

2 Trade-Offs in RTPG Systems

In previous work [3], we have argued that, in view of the many possible uses of program generation and our relatively modest understanding of those uses,

RTPG systems ought to be as general and flexible as possible. Generality has two meanings here: First, it refers to the richness of the language in which generated programs are expressed — the language inside quotations. Second, it refers to the programmer's freedom in dividing her program into fragments.

Is it legal to fill the hole in `$<int m () {‘(hole) return x;} >$` with the declaration `$<int x=10;>$`? How about filling `$<if (y==x) ‘(hole) else ... >$` with `$<break L;>$`? Is the position of the hole in this fragment legal: `$<try ... catch (‘(hole)) { ... }>$`? Can the hole in `$<‘(hole) class C { ... }>$` be filled with `$<import java.util.*;>$`? These are the kinds of questions we would ask to probe the generality, in the second sense, of an RTPG system.

Different systems make different trade-offs among the values of *generality*, *safety*, and *efficiency*. On the whole, safety and efficiency compete with generality. Disallowing the insertion of variable declarations, for example, allows the types of all variables used in a fragment to be known, and thereby promotes safety and efficiency, but it certainly constrains the programmer's ability to structure the program-generating process.

In most work on run-time code generation, speed and safety are primary concerns. We know of no system, other than Jumbo, in which the answers to all the questions asked above would be "yes." Consider the question of whether to allow a fragment to contain a declaration whose scope extends beyond that fragment. Partial evaluation-based systems [5,6,11,12] possess the "erasure property" — erasing quotation marks leaves a valid program which is equivalent to the original but is not staged. Thus, they follow ordinary scoping rules for declarations, and the generation process cannot introduce new declarations. Template-based approaches [11,13], which construct programs at run-time by combining precompiled fragments, are inherently limited to fragments that generate machine code; declarations produce no machine code. Other non-partial evaluation-based system [7,10] restrict the introduction of declarations to permit faster code generation.

The design of Jumbo gives precedence to generality, in both its meanings. By using the same compiler statically and dynamically, we ensure that the dynamic language is the same as the static one. And by giving each node in the abstract syntax tree its own semantics, and insisting that *any* node — even a declaration — can be left as a hole to be filled in at run time, we ensure the ability to divide up the program into almost arbitrary fragments.

Thus, by using a single compiler for both static and dynamic compilation, we lower development cost — there is no extra work beyond writing the one compiler — and get a system of great generality. On the other hand, we can then offer no safety guarantees, and suffer from inefficiency at run time. The latter is the problem we address in this paper.

3 Compositional Compilation

There would be nothing for us to do — we could achieve our goal trivially — if we were willing to carry the compiler with us wherever code was to be

generated, or to assume it existed everywhere. We could just emit source code and invoke the compiler from the running program. However, this approach is inherently inefficient, and more importantly, is extremely difficult to use in practice because of portability issues and the fundamental reliance on exporting source, which many organizations will not do.

Instead, Jumbo works by *partially* compiling each quoted fragment, producing a value of type *Code*. We will give the precise definition of *Code* shortly. First, we discuss the structure of our compiler.

The idea of compositionality is that the *Code* value to which any AST translates is a function only of the abstract syntax operator at its root and the values of each of its children. Thus, the compiler is really just a set of definitions of abstract syntax operators, but with arguments of type *Code* rather than *AST*. Examples are:

```
Code makeIfThen (Code cond, Code truebranch)
Code makeVariable(int flags, Type type, String name)
Code makeClass(int flags, String name, String supername,
               StringList implementees, CodeList members)
```

This is the essential difference between this structure and a conventional compilation structure: Instead of creating an AST and then generating JVM code while traversing it, the abstract syntax operators themselves contain the code to compile that syntactic construct.

A preprocessing step translates quoted fragments to abstract syntax operators, in the usual way. For example,[1]

```
Code safePointer (Code ptr, Code computation) {
   return $< if (‘Expr(ptr) == null)
               throw error();
             else ‘Stmt(computation) >$;
}
```

becomes (`0` is the code for binary operator "`==`")

```
Code safePointer (Code ptr, Code computation) {
   return makeStatements(
            makeIfThenElse(
              makeBinOp(0, ptr, nullConstant()),
              makeThrow(makeSelfInvocation("", "error", new List())),
              computation));
}
```

This program is now statically compiled — that is, as an ordinary Java program. The calls to the abstract syntax operations are part of the program and

[1] The syntactic category names are needed to allow parsing of holes within fragments. The holes are replaced by special names — `unknownExpr`, `unknownStmt`, etc. — before parsing. Zook et al. [14] describe a way to eliminate these using context-sensitive parsing, but we have not yet implemented their technique in Jumbo.

will be elaborated at run time, after the holes have been filled in. In particular, at run time, *Code* values will be provided for the arguments to `safePointer`, and the returned expression will be evaluated.

Eventually, this *Code* value will be placed inside the *Code* value for a compilation unit, and be ready for the final step of compilation — generating Java .class files containing JVM code. The method `void generate ()` performs this final step. Alternatively, `Object create (String classname)` calls `generate`, and then loads the class file and returns an object of the class. `generate` is for when Jumbo is used for off-line program generation, and `create` for true run-time program generation.

We have achieved our goal of allowing for partial compilation even for fragments of programs: the compilation of any fragment, *compile*(A), is its *Code* value, obtained by evaluating the expression to which the fragment is translated by the preprocessor. Holes are handled with no special effort — they are just expressions within this larger expression which do not happen to be explicit calls to abstract syntax operations. Mathematically, we can regard a fragment with a hole, $P[\cdot]$, as being compiled to a function from *Code* to *Code*:

$$\begin{aligned} &\mathit{compileWithHole}(\mathcal{P}[\cdot]) = \lambda C : \mathit{Code}.\ \mathit{compile}(\mathcal{P}[A]) \\ &\qquad \text{where } A \text{ is } \textit{any} \text{ fragment such that } \mathit{compile}(A) = C. \end{aligned}$$

Compositionality ensures that this function is well defined.

4 Jumbo

As discussed in [15,16], there are many choices for the *Code* type. A degenerate version of compositional compilation is to make *Code* be AST's, and let `generate` do all the work. In Jumbo, our goal is to put as much meaning as possible into *Code*, leaving `generate` very simple. The most natural way to do this is to make each *Code* value a function taking the compilation context (or "environment") to JVM code. This is how compositionality is achieved in defining abstract meanings of programs in denotational semantics [17], and it works just as well when "abstract meaning" is replaced by "compilation."

In Java, the situation is a bit more complicated, but for the most part we follow this idea. *Code* values are represented by objects having a single method, plus some additional information:

$$\mathit{Code} = \mathit{ExportedDefinitions} \times (\mathit{Environment} \rightarrow \mathit{ClosedCode})$$

$$\mathit{ExportedDefinitions} = (\mathit{ClassInfo} + \mathit{MethodInfo} + \mathit{FieldInfo})$$

$$\mathit{Environment} = \text{stack of } (\mathit{ClassInfo} + \mathit{MethodInfo} + \mathit{LocalInfo})$$

$$\mathit{ClosedCode} = \mathit{JVM\ code} \times \mathit{integer} \times \mathit{integer} \times \mathit{VarDecls} \times \mathit{Value}$$

The first component of *Code* is the declarations exported from the code fragment. The second is the function we have been referring to above, which we

call *eval*; it does the actual translation to JVM code. Exported declarations are just that: declarations that are in scope outside of this fragment. Based on the exported declarations of a class's members, the class can create a fairly complete record of its contents, and that record (a *ClassInfo*) will be its exported declaration. The *eval* method is given an environment containing all enclosing classes, methods, and variables, and then generates code. The two integers in *ClosedCode* give the next available location for local variables and the gensym seed, needed to assign unique names to anonymous classes. The *VarDecls* value carries the local variable declarations of that code fragment. The *Value* field gives the constant value of an expression, if it has one; the Java language definition [18, section 15.27] requires this.

In the implementation, `Code` is an abstract class with two methods: `DeclarationList getDecls ()`, and `ClosedCode eval (Environment)`.

The definition of *Code* is quite a delicate one, and we have gone through several iterations. The current definition is parsiminious in the sense of having as few components as we think is possible. We now explain briefly why this definition works. In Java, names fall under two scope rules: names defined within a method — local variables and inner classes — are in scope in statements that follow the declaration ("left-to-right" scope), while names defined in a class — fields and inner classes — are in scope everywhere within the class (with the exception that fields are not in scope in their own initializers). The exported definitions in *Code* are used to create the latter part of the environment; the environment passed into the *eval* function of the methods of a class contains all the fields and inner classes of that class. Names with left-to-right scope are passed along in the environment from one statement to the next, using the *VarDecls* in *ClosedCode*. Thus, the *eval* function for each statement gets an environment containing all the names in scope at that statement.

(Aside to Java experts: This definition is actually a little bit too parsiminous, in that it does not allow a proper treatment of free variables in inner classes. The rule about inner classes is that each variable captured by an inner class becomes a field of the inner class, and the constructors of the inner class must assign the variable to its corresponding field. The question is, how do we know which variables are actually used in an inner class? This information does not come from the exported definitions of the inner class, since references are not definitions, nor is it passed "left-to-right." We finesse this problem by assuming that all variables in scope in an inner class are referenced in that class. This gives a correct, but obviously non-optimal, implementation of inner classes. An earlier version of Jumbo had an additional "pass" — that is, another method in *Code* — whose purpose was to gather free variable references in classes; we removed it for reasons discussed in Section 7.)

So, our task comes down to this: In a Jumbo program, sections of quoted code become expressions of type *Code*. At run time, these expressions will be evaluated, producing a *Code* object whose `getDecls` and `eval` functions will then be invoked. We wish to optimize this entire process, but mainly the `eval` function of each *Code* value, since this is where most of the compilation occurs.

5 Source-Level Optimization of Java

In this section, we describe the optimizations we apply. These take the form of source-level transformations, including method inlining, constant propagation, and various simplifications.

At present, these optimizations are not all applied automatically. A number of transformations are "contractive" — simply put, they never make things worse — and they are applied repeatedly in a "clean up" process. Others — such as inlining — are potentially dangerous, in that they can lead to code expansion, and the system must be told to perform them. (The user interface highlights all inlinable methods and constructors, and the user clicks on the method name to inline it.) We intend to explore methods of automating the entire process in future research.

The transformations are mainly standard and will be described only briefly. We emphasize that all are valid transformations in Java. The idea is not to build an optimizer specific to our compiler, but to use the logic of Java to optimize it. On the other hand, the specific choice of analyses and transformations was made with knowledge of the compiler.

To simplify rewriting, we first normalize the code. There are three main parts of the normalization step:

FQCN: Converts every name to its fully qualified version. For instance, a field access `x` becomes `this.x`, and a field declaration `Code c;` becomes `uiuc.Jumbo.Compiler.Code c;`. (`uiuc` is our university's domain name, so it is the root of package names that reside here.)

For-While: Converts for-loops to while-loops. Also, each while-loop's condition is replaced by `true` and taken inside the loop. The flow goes out of the loop with a break-statement.

Flattening: Breaks complex expressions into simpler expressions. For instance, after this step, all the arguments going into a method call will be simple variables.

We can then apply the following rewrites. All must be applied "manually" — that is, by explicitly requesting the rewriting engine to apply them. However, Cleanup incorporates many of them in a fixpoint iteration; those are not normally invoked manually.

Inlining: Inlines a method invocation. Replaces return-statements of the inlined method with break-statements.

WhileUnroll: Unrolls the first iteration of a specified while loop.

AnonClassConvert: Converts anonymous classes to non-anonymous inner classes.

ConstructorInlining: This transformation is described below.

Unflatten: Transforms the flattened program to a form that is more readable.

Cleanup: Runs the following rewrites in a fixpoint iteration. Each can be invoked manually, but there is little reason to do so.

Untupling: Extracts a field from a newly created object.

UnusedDecl: Removes declarations that are never used.
UnusedScope: Removes scopes that have no semantic significance.
UnusedDef: Removes variable definitions that are not used.
UnusedReturn: Eliminates assignment of a method call when the assigned variable is not used. The method call must still be executed for its side effects.
IfReduction: Simplifies if-statements whose condition is a constant boolean.
Arithmetic: Simplifies constant-valued arithmetic and logic expressions.
UnusedBreak: Removes break statements that make no difference to the flow.
ConstantPropagation: Moves constant values through local variables.
CollapseSystemCalls: Collapses `intern` and `equals` calls made on `String`s.
ArrayLength: Replaces `array.length` expressions with the length, if available.
Switch: Reduces constant switch statements to the match.
CopyAssignment: Propagates redundant assignments of variables and literals.
UnusedObject: Removes object creation statements if they are never used and side-effect-free.
FieldValue: Propagates values through object fields assigned directly.
TightenType: Makes types more specific, if possible.
UnusedFieldAssign: Removes unused assignments to fields.
UnreachableCode: Removes code which is indicated to be unreachable by the flow analysis.
ObjectEquality: Replaces `(obj1 == obj2)` with `true`, and `(obj1 != obj2)` with `false`, if it can determine whether the two objects point to the same location; and vice versa.
PointlessCast: Removes cast expressions where the target of the cast is already the right type.
WhileReduction: Removes while statements which only have a `break` as the body and/or `false` as the condition.
InstanceOf: Attempts to resolve `instanceOf` expressions.
NullCheck: If it can prove that an object `o` is not null, then replaces `o != null` with `true` and `o == null` with `false`; and vice versa.

These rewriters use the information obtained from program analyses. The analyses are **Dominator**, **Flow**, **Use-Def** and **Alias**. The first three are standard. Our alias analysis is described in [19].

5.1 Constructor Inlining

Most of our transformations and analyses are strictly intra-procedural. This makes inlining very important for exposing opportunities for optimization. Constructors cannot be inlined like methods, because there is no notation to create

an uninitialized object in Java; this is an implicit effect of each constructor. (If we were optimizing JVM code instead of source, this would not be a problem.) We might try to use the zero-argument constructor for this purpose, but it might have an explicit definition that conflicts with the definition of the constructor we are attempting to inline. We solve this problem by adding *annotations*, containing the statements of the constructor, to object creation sites. Other rewriters then see the constructor code as though it was just an inlined method. The constructor itself, which resides in a separate class, cannot be optimized, but values propagated out of it can be used in the calling program. The annotations must be removed before the optimized program is written; for this reason, the annotations must have the property that they can be removed at any time and leave a program with the same meaning as when they were there.

6 Examples

We demonstrate the effect of our optimizations via three examples. The first is a complete (but small) class, without holes. The other two are the classic (in the field of program generation) exponentiation function, and a program to generate finite-state machines.

For each example, we show the original program, with quoted fragments. The latter will be preprocessed away and transformed to calls to abstract syntax operators, as described in Section 3. The resulting program is an ordinary Java program that will be compiled into JVM code and executed. At run time, the various *Code* values produced by these expressions will be brought together to form a *Code* value representing a class. A call to `generate` or `create` will turn this `Code` value into a Java .class file. In our examples, we are not executing the generated programs, since we are interested only in code generation time. In each test, we let the virtual machine "warm up" — load the Jumbo API, `java.lang`, and other classes — before executing the programs, then run each test 500 times. Our measurements exclude I/O time for outputting the .class file.

To obtain the optimized versions of the programs, each quoted fragment is optimized, in isolation, after it is preprocessed, using the rewritings described in the Section 5.

For each run — original or optimized — we measure the overall time, and we also measure the time spent in the method `Class.forName`. This method does the run-time look-up for names used but not defined in the program (for example, classes defined in imported packages). It consumes such a large portion of run-time compilation time — more than 50% in most cases — that its effect on speed-up is often substantial. Furthermore, these calls are impossible to eliminate by any static optimization, since the imports must be elaborated on the target machine (i.e. at run time). Since this cost is specific to Java, it is interesting to see what speed-up we would be getting if this cost could be ignored.

The tables in this section have two columns for each of three different Java virtual machines: Sun's HotSpot, Kaffe (an open source VM), and IBM's production VM. (HotSpot is included because it is the the most widely used virtual

machine, but none of the three is distinctly better than the others, nor is any of them the "definitive" virtual machine.) For each VM, we give the overall execution time and the execution time excluding `forName` calls; these are the "w" and "w/o" columns, respectively. The tables have three rows: unoptimized time, optimized time, and speed-up ((unoptimized time - optimized time) / unoptimized time).

The timings are in seconds. Tests were run on an AMD Duron 1GHz processor, with 790 MB of memory, running Debian Linux.

6.1 Simple Class

To get a kind of baseline, we show the results of optimizing a complete, but simple, class. The tests just invoke `generate` on this code:

```
$<
public class Temp {
  int x;

  int id() {
    return 12;
  }
}
>$
```

When presented with a complete class without holes, the rewriters ought to be able to reduce it to a very efficient form. However, the speedups are not as great as we would hope. (In the case of the IBM JVM, the rewriting actually produced a slow-down.) Reasons for this are discussed in Section 7.

Table 1. Run-time generation performance for the simple example

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	0.46	0.44	0.79	0.76	0.42	0.41
Rewritten	0.40	0.38	0.66	0.64	0.47	0.46
Speed-up	13.0%	13.6%	15.2%	15.8%	-11.9%	-12.2%

6.2 Exponent

The exponentiation function generator creates a function that computes x^n for given value of n. Table 2 gives the performance of the original and rewritten programs.

```
interface ExpClass
{ public int exponent(int x); }
```

```
public class Power {
  public static ExpClass getExp(int n) {
        Code r = $<1>$;
        for(int i = 0; i < n; i++){
            r = $<‘Expr(r) * x>$;
        }
        String cname = "Power"+n;
        Code expcl = $<
            public class ‘cname implements ExpClass {
                public int exponent(int x) {
                    return ‘Expr(r);
                }
            }
        >$;
        return (ExpClass)expcl.create(cname);
    }
}
```

Table 2. Run-time generation performance for the Exponentiation example

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	2.79	0.98	2.45	1.29	4.55	2.98
Rewritten	2.70	0.89	2.36	1.09	4.19	2.63
Speed-up	3.2%	9.2%	3.7%	15.5%	7.9%	11.7%

6.3 FSM

Another application of RTPG is generation of finite state machines (FSM). Table 3 gives the program generation timings for this example.[2]

The example is discussed in [3] and here we give its main class. Due to space considerations, we do not give the source of other classes. (`ArrayMonoList` is just a type of list; here it is used to collect all the cases in the switch statement that is the heart of the FSM implementation.)

```
public class FSM {
    String FSMclassname;
    State[] theFSM;
```

[2] It is notoriously difficult to understand the performance of Java virtual machines, and Table 3 is an example. The calls to `forName` are a large percentage of the execution time on all VMs. Furthermore, these calls are identical in optimized and unoptimized code. Yet speed-ups in two cases actually *decline* when `forName` is discounted. This is because, even though optimizations do not touch this method, it runs faster in the optimized than in the original code. We have, at present, no explanation for this behavior.

```
    FSM (String c, State[] M) { FSMclassname = c;  theFSM = M; }

    Code genFSMCode () {
        ArrayMonoList body = new ArrayMonoList();

        // Each state corresponds to a case in the switch statement
        for (int i=0; i<theFSM.length; i=i+1){
            body.addAll($<case ‘Int(i):
                          ‘Stmt(theFSM[i].genStateCode("ch"))
                          break; >$);
        }

        Code result =$<
            import java.util.*;

            public class ‘FSMclassname {
                static void runFSM (StringTokenizer in) {
                    int theState = 0;
                    while (true) {
                        char ch;
                        if (!in.hasMoreTokens()) return;
                        ch = in.nextToken().charAt(0);
                        switch (theState) {
                            ‘Case(body)
                            default: return;
                        }
                    }
                    return;
                }
                static void addToBuffer(char ch){ ...  }
                static void emitbuffer(){ ...  }
                public static void main (String[] args) {
                    String input = ...; // obtain input from console
                    runFSM(new StringTokenizer(input));
                }
            }>$;
        return result;
    }
}
```

The constructor of this class takes a finite-state machine description in the form of an array of states; the client sends the `genFSMCode` message to that object and then invokes `generate` on the result. The created class contains a main method that reads a string from the console and runs the client's FSM on it.

We haven't shown an FSM description due to space limitations, but to give a general idea, an FSM description is a set of states, and each state is a set of transitions. Here is the definition of a single transition.

```
new Transition(new Predicate1 (), 1, new Action2 ())
```

where

```
class Predicate1 implements Predicate {
    public Code pred (String ch) {
      return   $<('a' <= `ch && 'z' >= `ch)
           || ('A' <= `ch && 'Z' >= `ch )>$ ;
    }
}

class Action2 implements Action {
    public Code action(int s, String ch) {
        return $<addToBuffer(`ch);>$;
    }
}
```

This transition, if it sees a letter, goes from its current state to state 1 and puts the letter into the buffer.

Table 3. Run-time generation performance for the FSM example

	HotSpot		Kaffe		IBM	
	w	w/o	w	w/o	w	w/o
Original	13.10	4.93	14.01	8.82	8.92	3.89
Rewritten	12.25	4.76	13.48	7.78	8.37	3.70
Speed-up	6.5%	2.9%	3.9%	11.8%	6.2%	4.9%

7 Lessons Learned and Future Work

Compositional compilation can be applied to any language, yielding a compiler that supports run-time program generation (once the quotation/anti-quotation syntax is added). Each language will present different issues, both in construction of the compiler and in optimizing run-time program generation. Java is in some ways highly suitable for this treatment. Because it has no preprocessor and no optimization pass to speak of, most of the compiler consists of a translator from AST's to low-level code — the process to which compositionality applies most naturally. But in another sense, Java is *too dynamic*; some compilation steps must be performed dynamically that, in other languages, can be performed statically. Obviously, anything the must be done at run time cannot be optimized away. In this section we discuss why we have not gotten better speed-ups, and our future plans.

7.1 Optimization Problems

The major issue blocking rewriting is resolution of class names. The Java definition requires that these names be resolved on the target machine. Thus, for

example, the test to determine if a method override is legal — which must be done for every method — cannot be eliminated, because the superclass is available statically only in the rare case when it is defined in the quoted fragment itself.

Similarly, the normalization of class names (conversion of a short class name to a fully qualified class name) for variable, field, and method declarations must be done dynamically. This necessitates that the fields keeping track of type information be mutable: The objects containing those fields are the class and method objects created by `getDecls`, but normalized class names cannot be filled in until `eval` is called. Moreover, these objects are returned from `getDecls` to `generate`, so unless the fragment being optimized is in a place where the `generate` call can be inlined — which it usually is not — the class and method information have to be considered to have "escaped." Propagating information through mutable fields of objects that escape is very difficult.

One result is that the optimized code generator still contains type checks which we would initially have expected could be eliminated, such as a check for the validity of the return statement in `$<int foo() { return 5; }>$`.

Even if the fragment being optimized consists of a complete class, it is possible that the consumer of the fragment will compile it in a larger context: adding import statements, adding sibling classes, or making it an inner class. Not knowing this context causes more class name resolution problems. For example, if an enclosing class contains a field named "`java`", then "`java.lang.Object`" represents a series of field lookups, not a fully qualified class name, Having an explicit `create` or `generate` call available in the code being optimized resolves this difficulty, because it tells us that the fragment we see will not be placed in any larger context. However, as noted above, this will not normally be the case.

7.2 Next Steps

We have continually refined our compiler in two ways. One is reducing the number of "passes" — that is, the number of functions in *Code*. The idea is that putting more work in a single pass makes more information available locally; with multiple passes, each called from `generate`, the connection from one pass to another cannot be inferred except in those cases where we can see the `generate` call and inline it. As mentioned in Section 4, the current structure is as compact as we think is possible.

The other refinement is making the fields in the compiler's classes final. There is a bit more we can do along these lines.

More broadly, however, Java fundamentally limits optimizations because of the requirement to locate classes dynamically. This entails run-time calls to `forName`; in one case — the exponentiation example in HotSpot — `forName` consumes 65% of run-time compilation time. We have also noted above how dynamic class locating has a cascading effect: it requires that certain fields be mutable, which in turn diminishes our ability to statically determine their values.

It would be an interesting exercise to see what we could achieve if we assumed that imported classes could be looked up at compile time. But Jumbo

is a compiler for Java, not an idealized or subsetted version of Java, and we do not want to change that. Nor would this be a simple experiment: we have pointed out in this section how this property of Java has pervasive effects in the compiler; vacating this property would have correspondingly pervasive effects. So, our current thinking is that it may be time to apply our approach to a more conventional, less dynamic, language like C, and this is an avenue we are actively exploring.

8 Conclusions

We have shown how source-level optimizations can improve the performance of a program generation system based on the principle of compositional compilation.

The Jumbo compiler was first publicly released in 2003. We began the current study from (the newest version of) that compiler, but found that compositionality alone was not enough to permit optimization. We rewrote the compiler to be (a) *more* compositional — where the first definition of *Code* contained four functions, the current one has two — and (b) more functional in style, making greater use of final fields. It seems reasonable to us that, since RTPG can offer very significant performance advantages, the compilers to support it might be written so as to allow for more efficient code generation. In any case, in our future development of Jumbo, we will think of it as a process of co-design: writing optimizations that apply to the compiler, and modifying the compiler to make the optimizations applicable.

Though our speed-ups are still modest, we consider our results encouraging. There remain possibilities for further rewriting, even in Java, and we fully expect to see better results as we develop both the optimizations and the compiler. We are also exploring the application of our ideas to other, more static, languages.

Acknowledgements

We thank the reviewers, who provided numerous helpful comments on this paper.

References

1. Kamin, S., Clausen, L., Jarvis, A.: Jumbo: Run-time Code Generation for Java and its Applications. In: Proc. of the Intl. Symp. on Code Generation and Optimization (CGO '03), IEEE Computer Society (2003) 48–56
2. Clausen, L.: Optimizations In Distributed Run-time Compilation. PhD thesis, University of Illinois at Urbana-Champaign (2004)
3. Kamin, S.: Routine Run-time Code Generation. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), ACM Press (2003) 208–220 Also appeared in: SIGPLAN Notices, vol. 38 (2003), pp. 44-56.
4. Kamin, S.: Program Generation Considered Easy. In: Proc. of the 2004 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04), ACM Press (2004) 68–79

5. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
6. Taha, W., Calcagno, C., Leroy, X., Pizzi, E.: MetaOCaml. `http://www.metaocaml.org/`
7. Engler, D.R., Hsieh, W.C., Kaashoek, M.F.: 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '96), ACM Press (1996) 131–144
8. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: 'C and tcc: A Language and Compiler for Dynamic Code Generation. ACM Transactions on Programming Languages and Systems **21** (1999) 324–369
9. Poletto, M., Engler, D.R., Kaashoek, M.F.: tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In: Proc. of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97), ACM Press (1997) 109–121
10. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type Safe Dynamic Code Generation in Java. In: The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001). (2001)
11. Consel, C., Lawall, J.L., Meur, A.F.L.: A Tour of Tempo: A Program Specializer for the C Language. Sci. Comput. Program. **52** (2004) 341–370
12. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: an expressive annotation-directed dynamic compiler for C. Theoretical Computer Science **248** (2000) 147–199
13. Hornof, L., Jim, T.: Certifying compilation and run-time code generation. In: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 60–74
14. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ Programs with Meta-AspectJ. In Karsai, G., Visser, E., eds.: Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE 2004). Volume 3286 of Lecture Notes in Computer Science., Vancouver, Canada, Springer (2004) 1–18
15. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-1: Source-Level Components. In: Proc. of the First Intl. Symp. on Generative and Component-Based Software Engineering (GCSE '99), Springer-Verlag (2000) 49–64
16. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-2: Binary-Level Components. In: Proc. of the Intl. Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00), Springer-Verlag (2000) 28–50
17. Stoy, J.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977)
18. Gosling, J., Joy, B., Steele, G.: The Java Language Definition. Addison-Wesley (1996)
19. Morton, P.: Analyses and Rewrites for Optimizing Jumbo. Master's thesis, University of Illinois at Urbana-Champaign (2005)

Statically Safe Program Generation with SafeGen

Shan Shan Huang, David Zook, and Yannis Smaragdakis

College of Computing, Georgia Institute of Technology,
Atlanta, GA 30332, USA
{ssh, dzook, yannis}@cc.gatech.edu

Abstract. SafeGen is a meta-programming language for writing statically safe generators of Java programs. If a program generator written in SafeGen passes the checks of the SafeGen compiler, then the generator will only generate well-formed Java programs, for any generator input. In other words, statically checking the generator guarantees the correctness of any generated program, with respect to static checks commonly performed by a conventional compiler (including type safety, existence of a superclass, etc.). To achieve this guarantee, SafeGen supports only language primitives for reflection over an existing well-formed Java program, primitives for creating program fragments, and a restricted set of constructs for iteration, conditional actions, and name generation. SafeGen's static checking algorithm is a combination of traditional type checking for Java, and a series of calls to a theorem prover to check the validity of first-order logical sentences constructed to represent well-formedness properties of the generated program under all inputs. The approach has worked quite well in our tests, providing proofs for correct generators or pointing out interesting bugs.

1 Introduction

Program generators can play an important role in automating software engineering tasks. A large amount of research has concentrated on meta-programming tools for writing program generators more conveniently or safely [4,5,15,7,13,11,2,6,3,17,18]. Nevertheless, such tools have not enjoyed much practical adoption. Programming language designers typically find meta-programming to be too unwieldy and undisciplined to be added as a general-purpose language feature. Working programmers who routinely use and write generators seem to find that advanced meta-programming infrastructure adds very little to what they can do with simple, text-based tools. For instance, many tens of thousands of programmers worldwide use code templates in the text-based XDoclet tool [12] to generate code for interfacing with J2EE application servers.

If a sophisticated meta-programming tool is to become mainstream, it should offer significant value-added for the generator programmer, comparable to the value added by high-level programming languages over assembly programming.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 309–326, 2005.

In this paper, we explore one possible direction for adding such value. We present SafeGen: a meta-programming language that offers static guarantees on the correctness of the generator, yet is expressive enough for many practical applications. That is, a generator written in SafeGen is analyzed statically and its correctness is examined under *all* possible legal inputs, where the user specifies what constitutes a legal input. If the analysis succeeds, the generator is guaranteed to only produce well-formed Java code. This addresses a common problem in generator development and a major reason why meta-programming often appears too unwieldy and undisciplined: a generator may have bugs that cause it to produce illegal programs but only under certain inputs. Such bugs can stay undetected for a long time and may only be found by end users and not by the generator writer.

To achieve well-formedness guarantees, SafeGen has an easy-to-analyze language for describing generators. This offers restricted syntax for describing control flow, iteration, and name generation. Inputs to a SafeGen generator are limited to legal Java programs. That is, SafeGen generates programs by examining existing Java programs at a level comparable to that of Java reflection. All of SafeGen's reasoning is done in a logic that deals with reflective entities (e.g., methods of a class, argument types of a method, etc.), as opposed to, say, integer numbers. Intuitively, this makes SafeGen ideal for XDoclet-like [12] tasks. For instance, SafeGen is appropriate for going over an existing Java class and creating a delegator, or wrapper, or interface, or GUI class that will work correctly with the original class. In contrast, SafeGen is *not* appropriate for generation tasks such as creating specialized versions of the FFT transformation for specific matrix sizes and dimensions.

SafeGen statically checks the legality of code templates by combining traditional Java type checking algorithms with automated proofs of the validity of logical sentences. That is, SafeGen expresses the structure of the generator as a collection of first-order logic formulas, treated as axioms. Further axioms, also in first-order logic, encode standard properties of Java at the static checking level (e.g., the fact that a final class cannot be extended). Finally, correctness conditions of the generator are described as first-order logic conjectures. SafeGen uses an automated theorem prover, SPASS [16], to attempt to prove these correctness conditions under all inputs, based on the axioms.

SafeGen's contribution to the meta-programming research community is its novel approach of combining logic on reflexive properties of valid programs with program generation, to guarantee the legality of programs that are not generated until the run-time of the generator. This approach makes SafeGen the only meta-programming tool we know of that both guarantees at the compile time of the generator the type-correctness of the generated program, and allows generation of arbitrary pieces of code (potentially with references to free variables and unknown types). SafeGen also shows that despite the restriction on control flow and name generation, this approach still allows the expressiveness that is useful for many program generation needs. This general logic-based approach is not

limited to SafeGen's current target language, Java, but could be applied to other languages.

2 Motivation and Background

One can question whether static checking of a generator is a valuable feature. After all, once the generator is used, the generated program will be checked statically before it runs. So why try to catch these errors before the program is even generated? The answer is that static checking is not intended to detect errors in the generated program or even errors in the generator input, but errors in the generator itself. Although these errors will be detected at compile-time of the generated program, this is at least as late as the run-time of the generator. Thus, static legality checking for generators is analogous to static typing for regular programs. It is a desirable property because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator was tested). To see the problem in an example, consider a program generator that emits programs depending on two input-related conditions: (We use MAJ [18] syntax: code inside a quote, `‘[...]`, is generated. The unquote operator, `#[...]`, is used to splice the result of an evaluated expression inside quoted code.)

```
if (pred1()) emit( ‘[int i;] );
...
if (pred2()) emit( ‘[i++;] );
```

If, for some input, `pred2` does not imply `pred1`, the generator can emit the reference to variable `i` without having generated the definition of `i`. This is an error in the generator. However, it might not surface until after the generator writer has tested and widely deployed the generator. This error will then be detected by some random user. It should be the responsibility of a good meta-programming language to prevent such errors by statically examining the generator.

The problem of guaranteeing the well-formedness of generated programs is essentially a problem of analyzing the control-flow and data-flow of the generator. For instance, in the above code fragment, the question is whether there is a valid program path that reaches the second `emit` statement without passing through the first. Similarly, consider a generator that introduces two new names in the same lexical context:

```
emit( ‘[ int #[name1], #[name2]; ] );
```

For static well-formedness checking, we need to know that `name1` and `name2` do not hold the same value (or we will end up with an illegal duplicate variable definition in the generated program). This is a data-flow property.

We should note that an interesting special case of program generation already offers strong legality guarantees for generated programs. Specifically, multi-stage languages, such as MetaML[13], MetaOCaml[7] or MetaD[10] guarantee that the generated program is type-correct by statically checking the generator. In this

sense, multi-stage languages represent the state of the art in static safety checking of generators. Nevertheless, staging applies restrictions on the structure of the generator and prohibits the expression of code templates in arbitrary fragments. Both of our above code examples are not possible in a multi-stage language. In the first example, identifiers in generated code (e.g., `i`) cannot refer to generated variable definitions that are not in an enclosing lexical scope inside the generator text. This is a drawback, even if the final program is expressible in a multi-stage language: ideally, a good meta-programming language should allow its user to express a generator in the style the user finds most convenient. In the second example, it is not possible in a multi-stage language to have the name of a generated definition vary depending on generator input.(Concretely, in MetaOCaml syntax, we cannot write, `.<let .~name:int = 0 in .~name + .~name)>.`, since binding instances cannot be escaped. Similarly, we cannot escape a type, e.g., `.<let i:.~typename = 0 in i+i>.`)

These restrictions mean that multi-stage languages are ideal for program specialization where the entire code to specialize is available, but not program generation where the generated program may be partial and may need to cooperate with other parts whose structure is not known until generator run-time. For example, a common generation task for J2EE applications is to take as input an arbitrary Java class and produce a Java interface that contains all of the class's public methods [14]. In this case, there is no code to specialize that is statically known to the generator. If the generator is to reason about the well-formedness of its output, it needs to do so using abstract properties of yet-unknown program entities, such as "no two methods in the input class can have the same type signatures". This is exactly the kind of program generation that SafeGen intends to support.[1] From a technical standpoint, the problem is harder than multi-stage programming, since there are no restrictions as to how the control and data-flow of the generator can influence the contents of the generated program parts.

3 SafeGen Design

In this section we describe the main design of the SafeGen language. We first give a high-level overview of SafeGen and then present the language in detail.

3.1 Overview of the Approach

Before we discuss the specifics of the SafeGen language, we will offer a quick example of what SafeGen can do, which will hopefully illuminate the role of all the distinct language features described in detail in the next sections. As we have not yet defined all the elements of SafeGen syntax and functionality, we will appeal to the reader's intuition for our example.

[1] We expect that the general approach used in SafeGen could also apply to program specialization tasks. Nevertheless, as mentioned earlier, SafeGen's current input language and reasoning engine is limited to reflection-like properties, and cannot apply to, say, generating specialized numerical code for a given array size and dimensions.

A basic, but not too interesting, SafeGen generator is the following:

```
#defgen makeInterface (Class c) {
  interface I {
    #foreach(Method m : MethodOf(m,c)) { void #[m] (); }
  }
}
```

The elements of this definition are as follows. The generator is called `makeInterface`. It accepts a Java class as its argument. It generates an interface named `I` (this name may be modified at generator runtime if the generator is used multiple times in the same lexical context). For each method of the input class, the generator produces a void, no-argument method by the same name in the generated interface.

Although this generator is almost trivial, it is still challenging to determine automatically whether it will output a valid interface for every input class. For example, do all the declared methods have unique signatures? In its attempt to prove that the generated code is well-formed, SafeGen relies on three kinds of knowledge: assumptions about the input (in this example there are none other than the fact that it is a class), general knowledge of Java typing, and the assumption that the input comes from a well-formed Java program (e.g., the input class, `c`, has methods with legal names). SafeGen uses the SPASS theorem prover to attempt to prove well-formedness properties of the output under any possible input. All knowledge that SafeGen has about the program is expressed as first-order logic sentences. For instance, the following formula states that any two members (either classes or interfaces) in a well-formed Java package need to have different names. (We show here the formula in SPASS syntax in order to be concrete about the level of interfacing with the theorem prover.)

```
formula(forall([c1, c2],
  implies(and(equal(DeclaringPackage(c1),DeclaringPackage(c2)),
              equal(Name(c1), Name(c2))),
          equal(c1,c2))),
         MEMBERS_IN_PACKAGE_DIFF_NAME).
```

The well-formedness conjectures that SafeGen tries to prove are also expressed as logic sentences. For instance, the following is a conjecture that states that generated methods cannot have the same name and type signatures if they are in the same class.

```
forall([m1, m2],
  implies(and(method(m1),
              method(m2),
              equal(DeclaringClass(m1), DeclaringClass(m2)),
              equal(Name(m1), Name(m2)),
              equal(Formals(m1), Formals(m2))),
          equal(m1, m2)))
```

In fact, this conjecture cannot be proven for the above generator, `makeInterface`. All generated methods have the same signature and methods can have the same names, since the same method name can be overloaded in the input class, `c`. Therefore, in this example we see that the output is potentially ill-formed.

3.2 Language Design

We describe next the syntax and main concepts of the SafeGen language. The language is described through short examples, followed by a longer example in the end. For a more thorough exposition, the syntax is shown in Figure 2 in the Appendix.

Cursors. The two main concepts in the SafeGen language are those of a *cursor* and a *generator*. A cursor is a variable ranging over all entities satisfying a first-order logic formula over the input program. Thus, the input program is viewed as a collection of logical facts about its type declarations. For instance, a cursor expression in SafeGen would be:

```
Method m : MethodOf(m,c) & Public(m) & !Abstract(m)
```

This cursor, `m`, describes all non-abstract, public methods in class `c` (`c` is a cursor assumed to have been defined earlier). In general, the values of cursors are type-system-level entities in the input program (methods, arguments, classes, interfaces, etc.). The logic predicates used to build cursors can be viewed best as a reflection mechanism over Java programs. SafeGen has several predefined predicates that correspond to Java reflection information and the user can create new predicate symbols that represent arbitrary first-order logic formulas over the predefined predicates. Since the logical sub-language used to define cursors in SafeGen is a standard first-order logic, we postpone describing its specifics in detail until later in the paper.

Generators. A SafeGen generator is a way to express Java code fragments. Generators are defined with the `#defgen` primitive. For example, a trivial generator, always producing a constant piece of code, is:

```
#defgen trivialGen () {
   class C { public void meth() {} }
}
```

A generator can receive input parameters that are either cursors or predicates describing constraints on the inputs. For instance, the following defines a generator that accepts a single non-abstract class as argument. (The body of the generator is elided.)

```
#defgen myGen (Class c : !Abstract(c)) { ... }
```

Similarly, one can define a new predicate that constrains the input of the generator:

```
#defgen myGen (input(Class c) => !Abstract(c)) { ... }
```

The above line defines a new predicate, called `input`, that is used to describe properties of the generator input values—namely that they are non-abstract classes. Note that (unlike predicate definitions that we will see later) the "implies" (`=>`) operator is used for predicates defining generator inputs: the input is not *all* classes that are non-abstract, just some classes that are guaranteed to be non-abstract.

A SafeGen generator interfaces with the outside world through Java reflection entities and strings. For instance, a generator that takes a `Class` argument, as above, is implemented as a Java method that accepts a `java.lang.Class` object as argument.

The body of a generator (enclosed in `{...}` delimiters) can contain any legal Java syntax. This Java code is "quoted"—that is, it gets generated when the generator executes. Quoted code can also contain three SafeGen constructs that serve as "escapes": they direct the control and data-flow of the generator, allowing configuration of the generated code. These three SafeGen constructs are `#[...]` (pronounced "unquote"), `#foreach` and `#when`.

The `#[...]` operator is used for adding fragments of Java code inside a larger fragment. For instance, a generator can integrate the output of another. More interestingly, a generator can derive code fragments by applying several built-in functions on cursors. Available functions are: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`. Consider the example of the following generator:

```
#defgen myGen (Class c : !Abstract(c)) {
   #[c.Modifiers] class #[c.Name] {  }
}
```

This generates a new (empty) class with the same name and modifiers as the input class.

Functions `Name` and `Type` only generate one identifier, while `Formals` generates an array of $\langle ArgType, ArgName \rangle$ pairs, separated by commas. Functions `ArgNames`, `ArgTypes`, and `Modifiers` generate arrays of values, with `ArgNames`'s output separated by commas. Clearly, not all functions can be applied to all cursors. `Formals`, `ArgNames`, and `ArgTypes` can only be applied to `Method` cursors. SafeGen also allows the syntax `#[c]` on a cursor `c`. This is a shortcut for `#[c.Name]`.

The control flow of the generator is affected by primitives `#foreach` and `#when`, allowing iteration and conditional execution, respectively. SafeGen logic formulas determine iteration and conditional generation—thus, all iteration terminates and can only be over elements of the generator input.

The `#foreach` construct takes as argument a cursor definition. Inside the body of the `#foreach`, the cursor name can be used to refer to the current element in the range of the formula used to define the cursor. For instance, consider the following generator:

```
#defgen addFields (Class c) {
   #foreach ( Field f : FieldOf(f,c) ) { int #[f]; }
}
```

This creates a sequence of definitions of integer variables, each named after a field in the input class, `c`.

The `#when` construct's syntax is `#when ( LOGIC ) { CODE_TEMPLATE }`, optionally followed by `#else { CODE_TEMPLATE }`. That is, `#when` takes a logic formula as a parameter. If the formula evaluates to true at run-time, the first code template is generated. Otherwise, the code template following the `#else` is generated. In the example below, the argument to the generator is a set of Java interfaces (with no other constraints on them). If the set is not empty, then the "implements" clause gets generated, followed by all the names of interfaces. Otherwise, nothing gets generated.

```
#defgen maybeImplements ( input(Interface i) => true ) {
   #when ( exists (Interface in) : input(in) ) {
      implements #foreach(Interface i) { #[i] }
   }
}
```

Note that the above example also indicates that the generator's model ignores low-level separator tokens. I.e., our generators operate on abstract syntax trees, not parse trees. Thus, when the `#foreach` construct above generates multiple interface names, they get added to an AST. But when actual code is generated, they will be separated by commas, as Java requires.

User-Defined Predicates. For modularity and code reuse, SafeGen also allows definitions of new predicates both inside and outside the body of a generator. `#defpred` is used to give a name to a frequently used logic formula. The following example declares a predicate `myPred` that can be used in logic formulas, just like built-in predicates:

```
#defgen myGen ( ... ) {
   #defpred myPred ( Class c ) = Public(c) & !Final(c);  ...
}
```

Name Management and Hygiene. In the body of a generator, identifiers that correspond to generated definitions are hygienically renamed to avoid name conflicts. For instance, consider the following generator:

```
#defgen renameGen (input(Method m) => (m.Type = int) & noArg(m)) {
   #foreach( Method m: input(m) ) { int result = #[m](); }
}
```

(For convenience, the generator uses a predicate `noArg`, which we can define using `#defpred`. This constrains the input methods to accept no arguments.)

The result of the above generator will not be multiple definitions of variable `result`. Instead, at generation time, the actual variables generated will have

fresh names. Any references to these variables under the same cursor (or a cursor defined over a sub-range) will be consistently renamed to refer to the right variable. Since the renaming is only performed at the final output phase (i.e., when all generators have been called and the result is a complete Java compilation unit) SafeGen can tell which identifiers need renaming. Sometimes, a generator writer might indeed want to specify a name for a particular declaration, without renaming. In these cases, we provide the keyword `#name["..."]`. The identifier between quotes is generated as is.

Predicates, Cursors, and Logic in Detail. The logic underlying SafeGen is a sorted logic, with the basic sorts being: `Class`, `Interface`, `Method`, `Constructor`, `Field`, `Identifier`. Accordingly, all variables and constants in our domain are of one of these sorts. SafeGen does not provide any built-in constants. However, the user implicitly "creates" constants of the `Identifier` sort as needed. For example, if a user wishes to find all classes that implements `java.io.Serializable`, she writes the logical sentence:

```
forall (Class c) : (exists (Interface i) :
    ( InterfaceOf(i, c) & i.Name = "java.io.Serializable"))
```

`java.io.Serializable` is then declared as a constant in the domain during the compilation process.

The syntax for SafeGen logical sentences closely follows the syntax for first-order logic sentences (with the addition of sorts for declared variables). SafeGen provides logical operators `forall`, `exists`, `=`, `&`, `|` , `=>`, `!`, which correspond to all the operators available in first-order logic. The full list of available predicates and functions is shown in Figure 3 in the Appendix. For readers unfamiliar with first order logic syntax, please refer to Figure 2, rule `LOGIC` for details.

Example. We can now consider a non-trivial generator written in SafeGen. This is a realistic example, yet one that is short enough to study here and to use later for illustrating SafeGen's static checking process. The generator in Figure 1 takes a set of non-abstract classes as input and creates subclasses of the input classes with methods that just delegate to the superclasses' methods. (As explained earlier, the identifier `Delegator` is going to be renamed for each of the generated classes as to not induce name conflicts.)

3.3 Static Checking

We can now see how our approach can reason about a generator and guarantee that it produces well-formed programs under all inputs. Every well-formedness property of the output program is expressed as a logical formula. For instance, consider again our Section 2 example generator, for which we want to guarantee that a generated reference is always bound to a definition:

```
if (pred1()) emit( `[int i;] ); ... if (pred2()) emit( `[i++;] );
```

```
1. #defgen makeDelegator ( input(Class c) => !Abstract(c) ) {
2.    #foreach( Class c : input(c) ) {
3.       public class Delegator extends #[c] {
4.          #foreach(Method m : MethodOf(m, c) & !Private(m)) {
5.             #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
6.                return super.#[m](#[m.ArgNames]);
7.             }
8.          }
9.       }
10.   }
11. }
```

Fig. 1. A generator that generates a delegator class for an input class

The above example written in SafeGen is:

```
#when(logic_1) { int i; } ... #when(logic_2) { i++; }
```

where `logic_1` and `logic_2` are first-order logic formulas defined using built-in predicates and functions. Checking whether variable `i` is declared before use becomes checking the validity of the logical implication `logic_2` → `logic_1`. If the theorem prover proves validity, we know that under *any* input to the generator, the variable `i` would always be declared before it is used.

Other program well-formedness properties are also expressible in a similar fashion. Determining how to translate a given program property into a logical sentence is the role of the SafeGen implementation, described in the next section.

We should be explicit in that implementing checks for all well-formedness properties of Java programs is a heavy engineering task. SafeGen currently does not support all possible checks but we believe the omission is just a matter of engineering. [2] The currently supported checks in SafeGen are fairly representative in difficulty of the task and correspond to many valuable program correctness properties (e.g., method typechecking). Specifically, the currently fully supported tests are for the following properties.

- A declared super class exists.
- A declared super class is not `final`.
- Method argument types are valid.
- A returned value's type is compatible with the method return type.
- The return statement for a `void`-returning method has no argument.

[2] Any computable property can be expressed as the validity of a first-order logic formula. The only question is whether a theorem prover can reason about such properties effectively. For several yet-unsupported properties (i.e., properties for which SafeGen does not generate conjectures automatically) we have hand-produced logic formulas corresponding to example SafeGen programs and we have confirmed that we can reason about them in SPASS effectively. For instance, the conjecture in Section 3.1 was hand-produced, although our longer example in the Appendix (Figure 4) was automatically produced by the SafeGen compiler.

Notably missing checks include access control (e.g., no access to "private" variables outside class); checking for subtyping restrictions (e.g., a non-abstract class supplies definitions for all its superclass's abstract methods); checking for referring only to defined variables; checks for duplicate definitions; checking for correct declaration of exceptions; etc. We expect that many of them will be fully supported soon.

4 SafeGen Implementation

The most interesting part of the SafeGen implementation is the static checker. Therefore in this section we discuss how SafeGen produces axioms and proof obligations for a theorem prover, based on the structure of the SafeGen program.

4.1 SafeGen Static Checking

Although the SafeGen checking algorithm is not a traditional type-checker, it is easiest to present it in terms of type-checking, where both the names and the types of the various entities can depend on logic predicates.

SafeGen has two type-checking processes. One is type checking for the meta-language: legality of references to meta-variables, meta-level predicates, functions, and generators. (Meta-variables are either cursors or logic variables introduced by an `exists` or `forall` quantifier.) The second but much more complex one, is type checking for templated Java code. SafeGen's type system keeps two separate environments to support these two processes: the meta scope, for the generator, and the object scope, for the generated program.

Environment. A meta scope keeps track of meta level declarations: generators, predicates, and variables. A new meta scope is created by the following keywords: `#defgen`, `#defpred`, `#foreach`, `#when`, and quantifier keywords `forall` and `exists`. With the exception of `#when`, all of the keywords above create new meta-variable declarations. In addition to keeping track of declarations, `#foreach` and `#when` meta scopes are also associated with the logical sentences under which they are created. Each meta scope is linked to at most one parent meta scope. For example, in Figure 1, the meta scope created by `#foreach` on line 4 has the `#foreach` scope created on line 2 as a parent. The declarations in parent meta scopes are visible in the children scopes.

An object scope is much like a type environment for regular Java type checking. It contains symbol tables for types, variables, and methods. However, there are two unique elements of our object scope. First, all entries in the symbol table (e.g., names of variables or method declared in the scope, and the types these map to) may not be constants but dependent on a cursor over the input program. Second, each entry in the symbol tables has a link to a meta scope within which the entry is declared. For example, in Figure 1, class `Delegator`, declared on line 3, is an entry in the type table, with a link to the meta scope created on line 2, by `#foreach ( Class c : input(c))`. This meta scope in turn has a parent

meta scope corresponding to the `#defgen` in line 1. For an example of an object scope entry with an unknown name, consider the method declared on line 5 of Figure 1. The entry in the symbol table will contain the information that the method name is equal to the value of `m.Name` and the corresponding meta scope will be that defined by the `#foreach` on line 4 (with parent meta scopes those on lines 2 and 1). Only meta scopes created with `#defgen`, `#foreach`, `#when` can be linked from object scope entries.

Algorithm. SafeGen's type checking algorithm involves two phases. Phase I accomplishes the following two tasks:

1) Fully populate meta scopes and type check the meta language. Type checking the meta language is simply ensuring that a) every use of a meta-variable, predicate, function, or generator is defined, and b) if a meta variable is used as an argument to predicates, functions or generator calls, it has the correct type. For example, if meta-variables `m`, `c` are used in predicate `MethodOf(m, c)`, `m` should have a `Method` type, and `c` should have a `Class` or `Interface` type.

2) Collect type information in code templates. Object scopes are partially populated with only type information for declared types, their methods, fields, and inner types. No statements are inspected. There is no legality checking done in this phase. This step is analogous to a conventional type checking algorithm, where a first pass generates all the type information needed to type check the statements inside of method bodies and static initializers. After the object scopes are populated, we generate a logical representation of what is in the object scopes: a sentence describing the types available, their methods, fields, inner classes, etc. For the example in Figure 1, the initial segment of this sentence is:

```
forall([c],
  implies(and(Class(c), input(c)),
          exists([c'], and(Class(c'), Name(c')=Delegator, ...)))))
```

We call this sentence *fact*. It will be used in Phase II of the type checking algorithm, as described next.

Phase II is responsible for checking the type correctness of templated Java code. The algorithm resembles regular Java type checking in that it utilizes the symbol tables to look up information on variables, methods, and types. However, the algorithm is complicated by the use of meta-variables and functions in declarations and references. Therefore, SafeGen's type system combines the use of object scope symbol tables with the building of logical sentences using the meta scopes (i.e., the meta scope associated with the current object scope and all its parent meta scopes). For example, in Figure 1, we need to check whether the method call, `super.#[m](#[m.ArgNames])` on line 6 is a valid call. The first step is to look up the superclass of the current class using the symbol table. However, we find that `super` does not point to an actual class with its own symbol tables, but to a meta-variable, `#[c]`. In order to check whether `super.#[m](#[m.ArgNames])` is a valid call, we must construct a logical sentence to inquire: under all legal inputs to this generator (any class that is `!abstract`), and under the logical context

(encoded by the meta scope) in which this method call is used (namely, `#foreach(Class c:input(c)) {...#foreach(Method m:MethodOf(m,c))}`), does the class `#[c]` always have a method with name `#[m]`, and argument types the type of `#[m.ArgNames]`? This question is expressed as a logical sentence, *test*. The *test* sentence for the method call `super.#[m](#[m.ArgNames])` is shown in Figure 4 in the Appendix.

We then construct the sentence $fact \rightarrow test$, where *fact* was constructed in Phase I, as described earlier. *fact* needs to be the condition in the implication because it states the existence of classes and methods that *test* might refer to. Facts about the well-formedness of generator inputs are also part of the theorem prover input, supplied as axioms. We next feed this sentence to the theorem prover to test its validity. The full input to the theorem prover includes the logic definition (i.e., predicates, functions, sorts), axioms about Java, and the $fact \rightarrow test$ conjecture. This is typically many hundreds of lines long.

5 Discussion

Using the Theorem Prover. There are two approaches to using the theorem prover to verify the correctness properties of code templates. We could construct a large sentence that is the conjunction of all the type-correctness properties the templated code should preserve, and ask the prover whether these properties hold given the facts produced by the code templates. While this approach simplifies our language implementation by delegating all type checking duties to the theorem prover, it has a major disadvantage. The checking would be all-or-nothing and it would not produce very useful error messages to the users. When one of the properties in the conjunction fails to be valid due to a contradiction, all we receive from the theorem prover is a series of syntactic maneuvers that arrived at the contradiction. It is very difficult to decipher these messages to determine the exact property that failed. We can only inform the user that *somewhere* in their program, there is an error. The problem is exacerbated by spurious errors due to valid formulas that could not be proven: the user would be unable to tell that the error is spurious if we just reject the entire program.

Therefore, we have chosen a second approach. SafeGen's type checking algorithm is a combination of traditional Java type checking and calls to the theorem prover. We make calls to the theorem prover to check the validity of very specific properties. For example, when we are type-checking a class declaration, and we reach the declaration of a super class, we make two calls to the theorem prover. One is to check that the declared super class exists. Another is to check that the super class is a non-final class. This approach yields simpler logic formulas to prove. At the same time, we are able to produce very precise error messages to the user regarding exactly which property the code template failed to establish.

The one disadvantage of our approach is that we must make many calls to the theorem prover in the process of compiling just one generator. There might be a potential performance hit depending on how long the theorem prover takes to return answers. However, as discussed next, we have not yet found this to be a major cause of concern.

Experience. SafeGen is still work in progress. Nevertheless, we have experimented extensively with the checking process for formulas that correspond to SafeGen programs. In fact, we first chose example SafeGen programs and expressed in logic their properties that we wanted to check, before trying different theorem provers and eventually choosing SPASS.

The choice of theorem prover is largely orthogonal to the overall approach, and we may switch in the future. The overriding factor we used in choosing a theorem prover was its ability to arrive at a result without human guidance. We cannot expect the user of SafeGen to hand-tune the logic whenever the theorem prover fails. A theorem prover that fails to find either a definite proof of validity or a counterexample would cause SafeGen to produce lots of spurious warnings to users. After trying several (4) theorem provers, we chose SPASS because (in our tests) it demonstrated the best ability to terminate much of the time without human guidance. With our limited set of example validity tests, SPASS always finds a proof for the valid sentences. For sentences that are not valid, SPASS terminates with a decision roughly 50% of the time. It fails to terminate (during the several minutes we observed it) the other 50% of the time. This means that, for our examples, SafeGen issues no false positive errors. However, for half of the true type errors SafeGen reported, SafeGen was only able to report a "possible error", because SPASS did not terminate with a decision (i.e., a counterexample) that the sentence is not valid.

Because SafeGen makes a large number of calls to the theorem prover during type-checking, the performance of the theorem prover was a consideration, as well. So far, for the cases that SPASS was able to terminate, it terminates in under 1 second. This is hardly surprising: most of the properties we want to prove are quite shallow. For instance, for many type-checking tests, the types and meta scopes match exactly even though they are complex expressions involving cursors and logic predicates. Currently we set the time limit for each SPASS proof attempt at 3 seconds.

It is worth noting that our delegator example in Figure 1 has a bug that SafeGen readily detects: the superclass method is not always guaranteed to have a return type. If the return type of method `m`, called in line 6, is `void`, then the statement `return super.#[m](#[m.ArgNames])` is not legal. The user should instead use a `#when` clause, to detect whether the superclass method has a returnable result and if not to just call it without attempting to return its value.

Another result of our experiments with properties of sample SafeGen generators is that we tuned our logic to limit its expressiveness but maximize the number of proofs we can produce completely automatically. That is, when we find in our examples that a specific pattern causes consistent difficulties in reasoning, we remove the logic feature it depends on. For instance, transitivity is very hard to reason about. The superclass relation is transitive, but instead of specifying the transitivity fully in our logic axioms, we only expand it three levels. As a result, if the validity of a generator depends on a subtyping relation between classes more than 3 links away in the subtyping hierarchy, then our logic cannot express the proof and SafeGen will issue a spurious warning.

Big Picture: Soundness and Why a New Language? The SafeGen static checking algorithm is sound: if a generator is approved by SafeGen, it is guaranteed to be correct (with respect to the supported tests, of course—but with no fundamental reason why these tests cannot be all possible Java well-formedness tests). As in any static checking system, however, what matters most is not soundness but usefulness. After all, soundness is easy to achieve by just rejecting all programs. In the static checking arena, tools like ESC/Java [8] have garnered a lot of attention by trying to be useful, even though they are not sound.

We view the soundness argument as tied to another major decision, namely whether to support a hard-to-analyze programming language like Java as the meta-language, or to design a small, specialized language like SafeGen. If we were to implement our checking approach on a meta-programming system built on top of Java (such as our MAJ system [18]), we would certainly have sacrificed soundness to achieve usefulness. Java has several language constructs (including dynamic dispatch, aliasing and assignments, exceptions) that make it hard to be sound (i.e., guarantee correctness) while allowing a large percentage of the correct programs. Instead, our choice of creating a new language was largely so that we could be sound, yet useful. We believe that soundness is not a goal by itself, yet it is valuable in terms of user perception. Sound static checking mechanisms (such as type systems) are much more easily accepted by programmers than unsound tools (like `lint` or ESC/Java) because they feel more disciplined. At the same time, we have aimed at making SafeGen expressive enough for most program generation tasks that depend on reflection over existing programs.

Of course, SafeGen checking offers no guarantees of completeness: if we find no proof of the correctness of the generator, it is by no means certain that it is erroneous. Since first-order logic is undecidable, the proof process will not always terminate. We have examined the possibility of restricting our language to a broad but decidable fragment of first-order logic, such as the guarded fragment [1]. (In fact, SPASS, with the right choice of parameters is a decision procedure for the guarded fragment [9].) Nevertheless, we believe that this would limit significantly the expressiveness of our logic. Furthermore, it is not clear whether a guarantee of termination of the proof process with a decision is a very important property in practice, unless it is a guarantee of termination in a very short time, which seems impossible: such decision procedures typically have super-exponential complexity.

6 Conclusions

In this paper we presented SafeGen, a meta-programming language with the distinguishing feature that it offers powerful correctness guarantees for generators expressed in it. SafeGen statically checks its input to guarantee that only well-formed code will be generated at the generator's runtime. We demonstrated a novel approach that combines traditional static type checking with representing program correctness properties in logic. We believe that SafeGen is expressive and useful, even though its syntax is restricted so we can represent all program

correctness properties logically. We also believe that the approach of using logic to control and reason about code generation is one that extends beyond the implementation of SafeGen. It can be used for a different target language (from Java), and with a different logic (from one based on Java reflexive properties), suitable for other broad categories of generation needs.

Acknowledgments. This research was supported by the National Science Foundation under Grants No. CCR-0220248 and CCR-0238289.

References

1. H. Andreka, J. van Benthem, and I. Nemeti. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 31–42. ACM Press, 2001.
3. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, pages 187–197. IEEE Computer Society, 2003.
6. A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 10–18. ACM Press, 2002.
7. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2830, pages 57–76. Springer, 2003.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, June 2002.
9. H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *Logic in Computer Science conference (LICS)*, pages 295–304, 1999.
10. E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming (ICFP)*, pages 218–229, New York, NY, USA, 2002. ACM Press.
11. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.
12. A. Stevens et al. *XDoclet Web site, http://xdoclet.sourceforge.net/.*
13. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

14. E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.
15. E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2487, pages 299–315. Springer, 2002.
16. C. Weidenbach. SPASS: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.
17. D. Weise and R. F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.
18. D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In *Generative Programming and Component Engineering (GPCE) Conference*, volume 3286 of *Lecture Notes in Computer Science*. Springer, Oct. 2004.

Appendix

```
GENERATOR_DEF : "#defgen IDENT (" ( INPUT )? ") {" CODE_TEMPLATE "}" ;
INPUT         : CURSOR_DEC ("," INPUT )*
              | INPUT_PRED_DEC ("," INPUT )*  ;
CODE_TEMPLATE : JAVACODE
              | "#foreach (" CURSOR_DEC ") {" CODE_TEMPLATE "}"
              | "#when (" LOGIC ") {" CODE_TEMPLATE "}"
                ( "#else {" CODE_TEMPLATE "}" )?
              | GENERATOR_DEF
              | PRED_DEF  ;
CURSOR_DEC    : METATYPE IDENT ( ":" LOGIC )? ;
METATYPE      : "Class" | "Interface" | "Method" | "Constructor" | "Field";
INPUT_PRED_DEC: IDENT "(" ( PRED_ARGS )? ") =>" LOGIC ;
PRED_DEF      : "#defpred IDENT "(" ( PRED_ARGS )? ") =" LOGIC ;
PRED_ARGS     : METATYPE IDENT ("," METATYPE IDENT)* ;
JAVACODE      : Java syntax + "#[" + METAEXPR + "]" ;
METAEXPR      : IDENT ( "." META_FUN )*
              | IDENT "(" ( GEN_ARGS )? ")" ;
GEN_ARGS      : IDENT ( "," IDENT )* ;
META_FUN      : "Name" | "Type" | "Formals" | "ArgTypes" | "ArgNames" ;
LOGIC         : "forall" METATYPE IDENT : "(" LOGIC ")"
              | "exists" METATYPE IDENT : "(" LOGIC ")"
              | IDENT "=" IDENT
              | "!" LOGIC  |  LOGIC "&" LOGIC  |  LOGIC "|" LOGIC
              | LOGIC "=>" LOGIC ;
```

Fig. 2. SafeGen syntax

- Unary predicates: `Public`, `Private`, `Protected`, `Static`, `Final`, `Abstract`, `Transient`, `Strinctfp`, `Synchronized`, `Volatile`, `Native`
- Binary predicates: `PackageOf`, `ClassOf`, `InnerClassOf`, `InterfaceOf`, `SuperClassOf`, `ConstructorOf`, `MethodOf`, `FieldOf`, `ExceptionOf`, `ArgTypeOf`
- Functions: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`.

Fig. 3. Available predicates and functions in SafeGen

```
implies(
forall([c],
  implies(
   and(input(c), class(c)),
   exists([del],
     and(class(del),
         equal(Name(del), Delegator),
         forall([sc], equiv(equal(SuperClass(del), sc), equal(c, sc))),
         forall([m],
           implies(and(equal(DeclaringClass(m), c), method(m)),
                   exists([del_meth],
                     and(method(del_meth),
                         equal(DeclaringClass(del_meth), del),
                         equal(Name(m), Name(del_meth)),
                         equal(RetType(m), RetType(del_meth)),
                         equal(Formals(m), Formals(del_meth)))))))))),
forall([c],
  implies(and(input(c), class(c)),
  forall([m],
    implies(
     and(equal(DeclaringClass(m), c), method(m)),
     exists([meth],
       and(method(meth), equal(Name(meth), Name(m)),
           exists([sc],
             and(equal(DeclaringClass(meth), sc),
                 exists([c'],
                   and(equal(DeclaringClass(meth), c'),
                       equal(SuperClass(c'), sc),
                       equal(Name(sc'), Delegator))))),
          equal(Formals(meth), Formals(m)))))))))
```

Fig. 4. SPASS Conjecture for type-validity of "super" call in example

A Type System for Reflective Program Generators

Dirk Draheim[1], Christof Lutteroth[2], and Gerald Weber[2]

[1] Institute of Computer Science,
Freie Universität Berlin,
Takustr.9, 14195 Berlin, Germany
draheim@acm.org
[2] Department of Computer Science,
The University of Auckland,
38 Princes Street, Auckland 1020, New Zealand
lutteroth@cs.auckland.ac.nz, g.weber@cs.auckland.ac.nz

Abstract. In this paper we describe a type system for a generative mechanism that generalizes the concept of generic types by combining it with a controlled form of reflection. This mechanism makes many code generation tasks possible for which generic types alone would be insufficient. The power of code generation features are carefully balanced with their safety, so that we are able to perform static type checks on generator code. This leads to a generalized notion of type safety for generators.

1 Introduction

Generators are a cornerstone of today's software engineering, especially in the area of enterprise application development [1]. There exists a large variety of tools for the generation of database interfaces, GUIs and compilers, and even CASE tools can be subsumed under the notion of generators. Besides these very specialized examples of code generation technology, many systems have been developed that offer a more generic approach toward code generation. Some of these systems allow the user to extend a programming language with new constructs which trigger the generation of customized code.

In many cases it is not easy for a user to develop own code generators, even when using systems that support this explicitly. The user has to have knowledge about how a generator receives its parameters, how code is represented and processed, how code is emitted, and how a generator is deployed. The answers to these questions vary greatly from technology to technology. Code generation is a sensitive area because it depends on parameters, and the usual data structure involved, a syntax tree, is not trivial. A generator may work well most of the time but can potentially fail with some rare actual parameters, and an error may not be obvious but express itself in some slightly malformed parts of generated code. Using generators always bears the risk of introducing hard to find bugs, while a good generator has the potential to provide an economic and solid solution to

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 327–341, 2005.

a common problem. Complexity in the development of code generators leads to generators that are more error-prone.

In this paper we show how the concept of code generators can be made accessible to the user directly in object-oriented languages and how a type system can be extended to take generators into account. The aim is to make generators part of a program and not of the compiler while retaining the safety properties of a typed language. No internal knowledge of the compiler should be required, and the generation process should be transparent for the user. Placing generators into the language itself instead of into a compiler affects the language syntax as well as its semantics and safety; the challenge lies in integrating the new constructs syntactically without interfering with existing semantics. Typed languages usually offer a high degree of safety through the use of type systems, and type checkers are able to detect many potential execution errors statically. With the new concept of generators, however, new types of potential execution errors are introduced, namely those that happen when code generation produces ill-typed code. Consequently, code generation poses new challenges to type systems.

In Sect. 2 we introduce the Genoupe language, which integrates code generators into the C# language, by looking at source code examples. We also discuss its general applicability to different problems. Section 3 presents the novelties of Genoupe's type system and discusses some malformed examples of Genoupe code that cannot be given a correct type. Section 4 looks at related work and explains how Genoupe is different to similar approaches. The paper concludes with Sect. 5.

2 Object-Oriented Programming with Parameterized Generators: The Genoupe Language

Our concept for the integration of generators into object-oriented programming is called Genoupe. It was developed from the language Factory [2], which integrated reflective generators into Java, and implements a similar but strongly revised concept for C#. Genoupe introduces a syntax that is reminiscent of that of generic types, although it is not limited to classes or interfaces. Like for generic types the template paradigm is used, but in contrast to simple genericity, the template can contain generator code written in a special compile-time level language. This sublanguage is kept in an imperative style and along the lines of the C# language itself, so that a C# programmer will intuitively understand its meaning. Also the type system is analogous to the runtime one, but simpler for ordinary types, since we usually do not need as many features here for generation as we usually want for runtime code. With respect to generated types the type system gets somewhat more sophisticated, and we need a whole set of essentially new type rules. However, this is well worth it because, as we will see in Sects. 3 and 4, the new type system makes it possible to detect parts of a generator that can potentially generate malformed code, in contrast to just detecting code that is malformed itself.

In the Genoupe language a generator can be embedded into the source code like an ordinary type definition. Source code files written in the Genoupe language have the name suffix `.genoupe` and are compiled to ordinary C# source files with the same name but `.cs` suffix (see Fig. 1). Each time a generator is applied with new arguments, new types with unique names are created. If a generator is applied more than once with the same arguments in a compilation run, the corresponding code is generated only once.

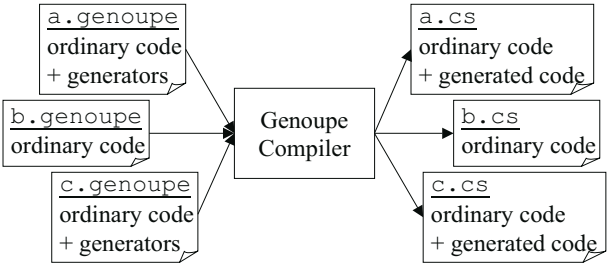


Fig. 1. The Genoupe compilation process

In a generator actual type parameters can be accessed through so called *generator variables*. These are variables that, in contrast to runtime variables, hold objects at generation-time and make them accessible in the generator code. Analogous to the parameters in an ordinary method, each declared generator parameter creates a generator variable, which can be used in *generator expressions*. A generator expression describes a values that is used at generation-time, just as an ordinary expression describes a value that is used at runtime. It is very similar to an ordinary C# expression in the sense that most generator expressions are valid C# expressions. One speciality of generator expressions is that, with the same values assigned to the generator variables, two structurally equal generator expressions describe the same value. We do not have non-deterministic effects like, e.g., random values, which are not needed in code generators. As we will see in Sect. 3.1, this will help us to rule out some potential generation errors statically.

Usually generator expressions are used to introspect type parameters and extract or construct the information that is needed for intercession, i.e., information that represents code that should be made part of the generator output. In order to make the value of a generator expression part of the generated code, the generator expression is enclosed in `@` characters and placed into the code template at a position where the entity that is represented by the expression's value is allowed to occur. If we want, for example, to generate a certain type in a declaration of a generated class, we would create a generator expression that evaluates to a `Type` object representing the desired type. This generator expression would be placed, enclosed in `@` characters, at the position in the source code where we would normally place a type name. At generation time all generator expressions are evaluated and substituted by the code represented by their values.

That is, if we had a generator expression of type `Type`, i.e., one that evaluated to a `Type` object, Genoupe would substitute the generator expression by the name of the type represented by the type object in the generated code. Genoupe makes use of the standard C# metaobject protocol, so that it is obvious in most cases which type represents which language entity.

In the following subsections we will consider some simple examples of Genoupe source code, which will point up how Genoupe can be used. Some applications for Genoupe, e.g., the generation of interfaces like GUIs or APIs, are not discussed here. Information on those and further examples can be found in [3,4].

2.1 Parametric Polymorphism

One of the simplest applications for Genoupe is parametric polymorphism. The following generic stack generator has a single parameter `T` of type `Type` and generates a stack class for elements of type `T`:

```
1  public class Stack(Type T)
2  {
3      private Stack s = new Stack();
4
5      public void push(@T@ x) {
6          s.push(x);
7      }
8
9      public @T@ pop() {
10         return (@T@) s.pop();
11     }
12 }
```

The generator parameter declaration in line 1 looks a bit similar to a method declaration, and like in a method declaration, a generator can have an arbitrary number of parameters with arbitrary type. In lines 5, 9 and 10 we insert generator expressions containing only the generator parameter in order to generate correct type declarations and type casts.

2.2 Class Extensions

Genoupe can be used for the generation of useful extensions. In contrast to ordinary inheritance mechanisms, which also extend classes, a generator can adapt the extension it generates to the class that is extended. This makes it possible to address static crosscutting concerns [5].

The following code snippet shows a generator that takes a class `T` and an array of field names `FNames` for that class. It generates a subclass of `T` that extends it by a new method `Randomize` that assigns random values to the fields of `T`. This can be useful, for example, for the generation of test data.

```
1  public class Randomizeable(Type T, String[] FNames) : @T@
2  {
3      public void Randomize() {
4          Random r = new Random();
5          @if(FNames!=null) {
6              @foreach(FName in FNames) {
7                  @const F = T.GetField(FName);
8                  @if(F.FieldType==Double)
9                      this.@F.Name@ = r.NextDouble();
10                 else @if(F.FieldType==Boolean)
11                     this.@F.Name@ = (r.NextDouble()>=0.5);
12                 // ...handle other data types...
13             }
14         } else {
15             @foreach(Field in T.GetFields()) {
16                 // ...generate assignments for all fields...
17         }   }
18 }   }
```

In line 5 we see the `@if` control construct of the generator language for conditional generation. It checks if an array of field names has been given at all, and only then the `FNames` array is used. In line 6 we see the `@foreach` construct, which is used for iterative generation. Its only difference to the `foreach` construct of C# is that the static type of the iterator variable needs not to be declared. In line 7 we define a new generator variable with a constant value, which is just syntactic sugar for our convenience. In the following lines, depending on the type of the respective field, we generate a statement that assigns to the field a compatible random value. The field's identifier is generated with a corresponding generator expression of type `String`. In the else-clause of the outermost `@if`, which is analogous to the aforementioned code, we handle the case that an array of field names was not given by generating code that assigns random values to all fields of `T`.

2.3 Proxies and Wrappers

A common pattern for modifying the behavior of existing classes or bridging incompatibility is the use of proxies [6] and wrappers. With Genoupe both of these can be generated automatically, which makes it possible to address dynamic crosscutting concerns [5].

The following class generator takes a type parameter `T` and creates a subtype of `T` that overrides and wraps `T`'s methods. A class generated by this generator behaves like `T` but logs all method calls and exits, which can be useful for debugging purposes.

```
1  public class Logger(Type T) : @T@
2  {
3      public String Log = new String();
```

```
4
5       @foreach(M in T.GetMethods()) {
6           @const Pars = M.GetParameters();
7
8           public override @M.ReturnType@ @M.Name@
9           (@foreach(P in Pars) { @P.ParameterType@ @P.Name@ })
10          {
11              Log += @new Literal(M.Name)@+" called.\n";
12              base.@M.Name@(@foreach(P in Pars) { @P.Name@ });
13              Log += @new Literal(M.Name)@+" exiting.\n";
14          }
15  }   }
```

In lines 8 and 9 we use generator expressions to generate the signature of each of `T`'s public methods. A list of method parameter declarations is generated by iterating over all the parameters and generating each parameter declaration individually. The same approach is used in line 12 in order to generate the list of arguments for a method call. The `Literal` objects constructed in lines 11 and 13 represent generated string literals, opposed to generated identifiers.

3 Generator Type Safety

When dealing with metaprograms, i.e., programs that process other programs or themselves in some suitable representation, a whole set of new sources of execution errors comes into play. *Generation errors* in generators are those parts of the generator program that can potentially generate malformed code, which in turn may cause execution errors when executed. Of course, we also want our generators to be free of execution errors themselves. In addition to normal type systems, which can only detect potential forbidden errors in the code that is type checked, we need a new kind of type system that can also detect parts in generators that can potentially generate ill-typed code. This requirement leads to a new notion of type safety, which we want to call *generator type safety*. It is the property of a generator not to be able to generate ill-typed code, i.e., code that may cause a forbidden execution error. If a generator is not generator type safe, it contains one or more *generator type errors*, i.e., parts in the generator code that are responsible for the generation of ill-typed code. We call a type system that can detect generator type errors a *generator type system*.

Before we describe the generator type system of Genoupe in the next section, let us look at examples of malformed generators that can potentially generate ill-typed code. The following generator generates a class with a single field:

```
1   class C(Type T)
2   {
3       @T@ x = 1;
4   }
```

The fact that `x` is assigned a numerical value restricts its possible type. The type parameter `T` however is not subject to any such restriction. This is clearly a generator type error that leads to some arguments producing type-correct code and others not.

The next example demonstrates another issue of type compatibility.

```
1   class C(T istype Component)
2   {
3       @T@ x = new Button();
4   }
```

The Genoupe keyword `istype` makes it possible to set a bound for type parameters, i.e., parameters of type `Type`. Line 1 signifies that parameter `T` is a type parameter and that all possible arguments represent types that are either class `Component` itself or one of its subclasses. In the generator body we define a member variable `x` with type `T`, to which we assign a `Button` object. `Button` is a subclass of `Component`, but what if `T` is a subclass of `Component` but not compatible to `Button`, i.e., not either `Button` itself or one of its superclasses? The generated code is type correct iff `T` is `Button` or one of its superclasses.

The following example is a class generator that has a string parameter `ID`. As the name suggests, the string is used to generate the identifier of a local variable in a method.

```
1   class C(String ID)
2   {
3       void m() {
4           int @ID@ = 1;
5           x++;
6       }
7   }
```

In line 5 we increment a variable `x`. Since there are no other variable definitions in the generator, `x` must be defined in the preceding line where the identifier of a variable is generated by a generator expression. If the generator is given the argument `"x"`, the generated code works just fine, otherwise it is ill-typed. This is also known as the problem of *inadvertent capture* [7].

The next generator contains a conditional generation.

```
1   class C(String X)
2   {
3       @if(X.Equals("hello")) {
4           @T@ y = "world";
5       }
6
7       void m() {
8           Console.WriteLine(y);
9       }
10  }
```

The definition of the member variable `y` is only generated when `"hello"` is the string argument in `X`. Again, we have cases where this generates an error and others where it does not.

Our last example illustrates a generator type error that can occur in iterative generation.

```
1  class C(Type S, Type T)
2  {
3      @foreach(F in S.GetFields()) {
4          @F.FieldType@ @F.FieldName@;
5      }
6
7      void m() {
8          @foreach(F in T.GetFields()) {
9              Console.WriteLine(this.@F.FieldName@);
10         }
11     }
12 }
```

The first generative iteration replicates the field definitions of type parameter `S`. The second one in method `m` generates statements that access and print the values of fields as defined in type parameter `T`. Clearly this can only work if `T` contains fields with identical name for all the field definitions in `T`, which is of course the case when `S` and `T` are bound to the same type.

All these generator type errors also occur in real generators, and usually they occur in a subtler way that makes them much harder to find. Such errors are typically introduced, for example, when applying inconsistent changes: one part of a generator is changed without adjusting other parts accordingly that are affected by that change.

Note that the Genoupe language has another property which makes its generators safer than those in many other languages: if all the methods we use in generator code terminate and we do not use generators recursively, which is usually unnecessary, a generator is guaranteed to terminate. This is because our looping construct, the @foreach, iterates over collections without modifying them, and the collections contain of course only a finite number of elements. In C++ templates, for example, we must use recursion when we want to repeat something arbitrarily often. C++ templates can potentially recourse endlessly, and only a limited recursion-depth prevents this [8]. In other technologies which use a Turing-complete language for metaobject manipulation, like CLOS [9], OpenC++ [10] or Jasper [11], generators potentially do not terminate as well.

3.1 The Genoupe Type System

In order to detect generator type errors, we developed a generator type system which is compatible with and extends the type system of the host language C#. Its notation is similar to the one used in [12]. It consists of rules with judgments about the correctness of certain program parts in their pre- and postconditions,

and only the programs that can be derived by those rules are considered type correct. In some respects, however, our type system deviates from the way in which type systems of object-oriented languages usually work. We use an environment Γ, which keeps track not only of the signatures of declared runtime variables but also of the signatures of generator variables. The signature of a runtime variable can contain generator expressions because its identifier and type may be generated by them. For handling conditional and iterative generation of declarations correctly, definitions that are generated conditionally or iteratively have special signatures, and Γ is also used to store additional facts about the code portion that is being type-checked.

Rather than delivering a complete description of the type system, this paper focuses on explaining the main concepts by looking at some exemplary type rules. These rules can be found in Table 1, and we will go through them one after another. Rule $[Env\ Var]$ describes how the signature of a generated variable can be included into Γ. The two judgments in the precondition state that we need a correct generator expression of type `String` for the variable's identifier, and a correct generator expression of type `Type` for the variable's type. The `::` symbol associates a generator expression with its type. In the postcondition the new environment is a conjunction of the old Γ and the new signature. The `:` symbol associates the identifier of a variable with its type. Rule $[Env\ then]$ allows us to register in Γ that a generator expression *Gexpr* evaluates to true. The generator expression must be of type `Boolean` and the opposite, i.e., that *Gexpr* evaluates to false, must not be registered in Γ already. As the name of the rule suggests, this rule is used for type-checking in the then-clause of an `@if` construct, where the generator expression describing the condition of the `@if` is known to be true. Analogous to this, rule $[Env\ loop]$ allows us to register in Γ that an iterator variable of a `@foreach` contains an element of a particular collection, which is the collection over which is iterated.

Rule $[Def\ Var]$ describes how a variable definition can be generated with suitable generator expressions and what its signature looks like. The $\therefore$ symbol associates a signature to a definition. A signature is a set of facts that describe a definition. Rule $[Def\ @if]$ describes the conditional generation of definitions. In the second and third line of the precondition, we see that the facts *Gexpr* and $\neg$*Gexpr* are included in the environment when we demand that the declarations D_1 and D_2 have the signatures Sig_1 and Sig_2, respectively. Consequently, the judgment in the postcondition states the correctness of an `@if` with D_1 in the then- and D_2 in the else-clause. The signature of the $@if$, which becomes part of the environment during type-checking, has two parts: one describing the signature of the generated definition in the case that the condition is true and one describing the signature of the generated definition when its not. Rule $[Def\ @foreach]$ describes the iterative generation of definitions. In the second judgment of the precondition we demand that D is a correct definition with signature Sig. The environment states that ID is an iterator variable which contains an element of the collection described by *Gexpr*. The signature of the resulting `@foreach` is again a special one: it signifies that for any generator vari-

able X, with X being an element of some collection described by $Gexpr$, there is a signature that looks like the signature of D, only that each occurrence of ID in that signature is substituted by X.

The rules [$Expr\ Var\ 1$], [$Expr\ Var\ 2$] and [$Expr\ Var\ 3$] will hopefully clarify why we need these unusual elements in Γ. They all specify how we can use a generated variable in a generated expression. Rule [$Expr\ Var\ 1$] states that if there is a generated variable declared in Γ, we can generate an expression that uses it by generating its identifier with a corresponding generator expression. Rule [$Expr\ Var\ 2$] describes under which circumstances a variable can be used that has been generated in the then-clause of a conditional generation: it can be used if Γ states that $Gexpr$, the condition under which the variable was generated, is true. Analogous to this rule, there is also one for using a variable that has been generated in the else-clause of an `@if`. Finally, rule [$Expr\ Var\ 3$] handles the usage of variables that have been generated in a `@foreach`. Such a variable can be used if Γ states that the usage of the variable is in the body of a `@foreach` loop that loops over a collection described by the same generator expression as the collection of the loop in which the variable was defined. This means that the collections of the two loops are equivalent. In the loop in which we generate code that uses the variable, the iterator variable may have a different identifier. Therefore we substitute the X in the variable's signature by the ID of this loop's iterator variable.

3.2 Limitations

Like most type systems, the Genoupe type system is restrictive: it forbids not only programs that are obviously incorrect but also many others which do not contain generator type errors. In the rules for the `@if`, for example, we require that a conditionally generated variable must be used in the body of a conditional with equivalent condition. Logically it would be enough, though, to require that the condition of the defining conditional *implies* the condition of the conditional in which the variable is used. Analogously, if variables are generated in a `@foreach`, it would be sufficient to demand that they are used in a loop that iterates over a *subset* of the collection in the defining iteration. Because the underlying problems are undecidable, we did not try to solve them, although it would be possible to address these issues using approaches from logical programming like, for example, constraint solving and model checking. Note that this is a popular way for type systems to deal with issues that restrict the way a language is used but do not really limit its applicability: C# and Java, for example, do not really check whether a method with a non-void return type returns a value; they merely check if a superset of possible execution paths returns a value.

The possibility to generate arbitrary identifiers with generator expressions brings about lexical problems: a generated identifier might be malformed, e.g., it might clash with a keyword, or might not be unique. Both these problems could only be solved if we restricted the way identifiers can be generated. But if we did that, we would lose flexibility and potentially the ability to produce clear human-readable names, and the language would become more complicated.

Table 1. Exemplary type rules of the Genoupe generator type system

$$[Env\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{String} \quad \Gamma \vdash Gexpr_2 :: \texttt{Type} \quad Gexpr_1 \notin \mathrm{Dom}(\Gamma)}{\Gamma \cup \{Gexpr_1 : Gexpr_2\} \vdash \diamond}$$

$$[Env\ then] \quad \frac{\Gamma \vdash Gexpr :: \texttt{Boolean} \quad (\neg Gexpr) \notin \Gamma}{\Gamma \cup \{Gexpr\} \vdash \diamond}$$

$$[Env\ loop] \quad \frac{\Gamma \vdash Gexpr :: \texttt{ICollection}}{\Gamma \cup \{ID \in Gexpr\} \vdash \diamond}$$

$$[Def\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{Type} \quad \Gamma \vdash Gexpr_2 :: \texttt{String}}{\Gamma \vdash @Gexpr_1@\ @Gexpr_2@; \therefore \{Gexpr_2 : Gexpr_1\}}$$

$$[Def\ @if] \quad \frac{\begin{array}{c}\Gamma \vdash Gexpr :: \texttt{Boolean} \\ \Gamma \cup Sig_1 \cup \{Gexpr\} \vdash D_1 \therefore Sig_1 \\ \Gamma \cup Sig_2 \cup \{\neg Gexpr\} \vdash D_2 \therefore Sig_2\end{array}}{\begin{array}{l}\Gamma \vdash @\mathrm{if}(Gexpr)\ \{\ D_1\ \}\ \mathrm{else}\ \{\ D_2\ \} \\ \therefore \{Gexpr \rightarrow Sig_1,\ \neg Gexpr \rightarrow Sig_2\}\end{array}}$$

$$[Def\ @foreach] \quad \frac{\Gamma \vdash Gexpr :: \texttt{ICollection} \quad \Gamma \cup Sig \cup \{ID \in Gexpr\} \vdash D \therefore Sig}{\Gamma \vdash @\mathrm{foreach}(ID\ in\ Gexpr)\ \{\ D\ \} \therefore \{\forall X \in Gexpr.Sig[X/ID]\}}$$

$$[Expr\ Var\ 1] \quad \frac{(Gexpr_1 : Gexpr_2) \in \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 2] \quad \frac{\{Gexpr,\ Gexpr \rightarrow Gexpr_1 : Gexpr_2\} \subseteq \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 3] \quad \frac{\begin{array}{c}\{ID \in Gexpr,\ \forall X \in Gexpr.(Gexpr'_1 : Gexpr'_2)\} \subseteq \Gamma \\ (Gexpr'_1 : Gexpr'_2)[ID/X] = (Gexpr_1 : Gexpr_2)\end{array}}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

The more freedom we allow for the generation of identifiers, the more complex a collision detection scheme would have to be in order to avoid this problem. We decided not to implement any such restriction or detection scheme and take the risk of lexical collisions, which is inherent when working with a textual source code representation. The responsibility for handling the generation of identifiers carefully lies with the programmer of a generator, for whom this is usually unproblematic.

4 Related Work

Genoupe is an extension of *genericity* or parametric polymorphism found, for example, in ADA or Java [13,14]. With parametric polymorphism it is possible to program components that are uniformly reusable for many types. However, these generic type parameterization mechanisms are at the same time type abstraction mechanisms: the construction of the type cannot be exploited in the

parameterized software component – at most it can be exploited up to a bound, known as bounded parametric polymorphism. Therefore it is useful for container libraries, e.g., C++ Standard Template Libraries, but it is not as powerful as Genoupe.

The original *C++ template mechanism* does not allow for the enforcement of properties for actual type parameters as, for example, supported by the notion of bounded parametric polymorphism [12,15]. Ad-hoc solutions to provide some level of concept checking for C++ templates, like specialized macros [16] and static interfaces [17], has been generalized by the introspection library approach in [18]. This approach targets user-customized checks for both compile-time adaptation and diagnostics.

The *new C++ templates* standard allows in principle Turing-complete metaprogramming with static and dynamic reflection in C++ [19], sufficient, e.g., for an interface generator for a relational database [20]. It is still less powerful than Genoupe; for example, it is not possible to generate function names dependent on a parameter. It does not support any static notion of generator type safety; type-checks are done with the ordinary C++ type system. Furthermore, a template metaprogram may not terminate. The Turing-completeness makes it impossible to analyze the generating templates exhaustively.

Aspect oriented programming aims at handling of crosscutting concerns in programs. AspectJ [5] is a Java extension for aspect oriented programming, which offers two approaches: dynamic and static crosscutting. Crosscutting does not help us with type-dependent generative problems, e.g., the implementation of a transparent data-access layer. Static crosscutting allows to extend the signature of classes and interfaces, but not in an adaptive manner: we can add a new method to a class from within an aspect – so-called member introduction – but still the method has to be specified literally and cannot be made dependent on some parameter. The generative approach to aspect-oriented programming in [21] characterizes certain uniform patterns that arise in using the aspect oriented style of inverting functional decomposition as amenable to be handled by the incremental computation approach. Based on this insight the approach establishes a behavioral semantics for generative aspect-oriented features that are oriented towards finite differencing [22].

The concept of *runtime reflection* dates back to Lisp [23] and has been subject of major interest in the functional programming community. The combination of parametric polymorphism with reflective features in Generic Haskell [24,25] benefits from the theoretical well-understood type-system of the host language. In the context of the object-oriented functional programming language CLOS [26,9], a mature metaobject protocol has been elaborated. In [27] CLOS is used to prove the value of metaprogramming by embedding representations of common object-oriented design patterns [6] into programs. Multistage programming [28,29] is an approach that focuses on runtime program generation and execution. The programmer is supported by constructs for partial evaluation and program specialization, whereas several properties of runtime generation can already be ensured statically. An implementation of the multistage programming approach is pro-

vided on top of the object-based functional programming language O'Caml [30]. The language Metaphor [31] results from extending the subset of an object-oriented language like C# or Java by the multistage constructs of the functional programming language MetaML [28,29], i.e., a construct for building representations of expressions, a construct for splicing code and a construct for running staged evaluated code. With its multi-staged language design Metaphor achieves type-safe generation of code that makes use of the reflection system of the base language.

Jasper [11] is a reflective syntax processor for Java. It provides mechanisms for *static reflection*. It does not follow the template approach; instead it allows for metaprogramming through the extension/modification of the syntax processor itself [32] – an architecture that is known as *open compiler*. It supports universal metaprogramming and is as such more powerful, but less understood.

5 Conclusion

Genoupe implements a concept for generative programming that integrates reflection by means of a metalanguage into a template mechanism reminiscent of genericity. It can be used to solve common problems of generative programming and offers advantages compared to other languages with respect to the degree of integration of the runtime and the metalanguage and safety:

- Genoupe places the concept of generators into the language instead of relying on an external tool driven approach, thus minimizing the interface to the user and avoiding potential errors.
- It integrates well with an object-oriented host language and can be seen as a generalization of genericity. It uses similar syntax for runtime and generator code, which makes it easy to use and understand.
- A wide range of common applications of generative programming can be addressed.
- Genoupe offers an particular high degree of static safety for reflection by means of a type system that is able to detect generator type errors.

More information about Genoupe and implementations of the Genoupe system can be found on our project web site, `http://www.genoupe.formcharts.org/`.

References

1. Draheim, D., Weber, G.: Form-Oriented Analysis - A New Methodology to Model Form-Based Applications. Springer (2004)
2. Draheim, D., Lutteroth, C., Weber, G.: Factory: Statically Type-Safe Integration of Genericity and Reflection. In: Proceedings of the 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS (2003)
3. Draheim, D., Lutteroth, C., Weber, G.: Integrating Code Generators into the C# Language. In: Proceedings of ICITA 2005: The 3rd International Conference on Information Technology and Applications, IEEE Press (2005) to appear.

4. Draheim, D., Lutteroth, C., Weber, G.: Generative Programming for C#. ACM SIGPLAN Notices (2005) to appear.
5. Kiczales, G.: An overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming. LNCS 2072, Budapest, Hungary (2001) 18–22
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
7. Kohlbecker, E., Friedman, D., Felleisen, M., Duba, B.: Hygienic Macro Expansion. In Gabriel, R., ed.: Proceedings of the ACM SIGPLAN Conference on LISP and Functional Programming, ACM Press (1986) 151–181
8. Czarnecki, K., Eisenecker, U.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (2000)
9. R. G. Gabriel, D. G. Bobrow, J.L.W.: Object Oriented Programming - The CLOS perspective. The MIT Press, Cambridge, MA (1993)
10. Chiba, S.: A Metaobject Protocol for C++. In: OOPSLA 1995 - Proceedings of the 10 th Conference on Object-Oriented Programming Systems, Languages and Programming. SIGPLAN Notices, ACM Press (1995) 285–299
11. Nizhegorodov, D.: Jasper: Type-Safe MOP-Based Language Extensions and Reflective Template Processing in Java. In: Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends, ACM Press (2000)
12. Cardelli, L.: Type Systems. In: Handbook of Computer Science and Engineering. CRC Press (1997)
13. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: OOPSLA 1998 - Conference on Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices, ACM Press (1998) 183–200
14. Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.E., Stoutamire, D., Thorup, K., Wadler, P.: Adding Generics to the Java Programming Language: Participant Draft Specification. Technical report, SUN Microsystems Ltd. (2001)
15. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
16. Siek, J., Lumsdaine, A.: Concept Checking: Binding Parametric Polymorphism in C++. In: First Workshop on C++ Template Programming, NETOBJECTDAYS 2000. (2000)
17. McNamara, B., Smaragdakis, Y.: Static Interfaces in C++. In: First Workshop on C++ Template Programming, NETOBJECTDAYS 2000. (2000)
18. Zólyomi, I., Porkoláb, Z.: Towards a General Template Introspection Library. In: GPCE 2004 - The 3rd International Conference on Generative Programming and Component Engineering. LNCS 3286, Springer (2004) 266–282
19. Attardi, G., Cisternino, A.: Reflection Support by Means of Template Metaprogramming. In: 3rd International Conference on Generative and Component-Based Software Engineering. LNCS 2186 (2001)
20. Attardi, G., Cisternino, A.: Template Metaprogramming an Object Interface to Relational Tables. In: REFLECTION 2001 - The 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. LNCS 2192 (2001)
21. Smith, D.R.: A Generative Approach to Aspect-Oriented Programming. In: GPCE 2004 - The 3rd International Conference on Generative Programming and Component Engineering. LNCS 3286, Springer (2004) 39–54
22. Paige, R., Koenig, S.: Finite Differencing of Computable Expressions. ACM Trans. Program. Lang. Syst **4** (1982) 402–454

23. Smith, B.C.: Reflection and semantics in LISP. In: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1984) 23–35
24. Hinze, R., Jeuring, J.: Generic Haskell: Practice and Theory. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. LNCS 2793, Springer (2003) 1–56
25. Hinze, R., Jeuring, J.: Generic Haskell: Applications. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. LNCS 2793, Springer (2003) 57–97
26. Kiczales, G., des Rivières, J.: The Art of the Metaobject Protocol. MIT Press (1991)
27. von Dincklage, D.: Making Patterns Explicit with Metaprogramming. In Pfenning, F., Smaragdakis, Y., eds.: GPCE 2003 - The 2nd International Conference Generative Programming and Component Engineering. LNCS 2830, Springer (2003) 287–306
28. Taha, W., Sheard, T.: Multi-Stage Programming with Explicit Annotations. In: PEPM 1997 - Partial Evaluation and Semantics-Based Program Manipulation. SIPLAN Notices, ACM Press (1997) 203–217
29. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
30. Leroy, X., Doligez, D., Garrigue, J., Rmy, D., Vouillon, J.: The Objective Caml system release 3.08 - Documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique (2004)
31. Neverov, G., Roe, P.: Metaphor: A Multi-stage, Object-Oriented Programming Language. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE'04). LNCS 3286, Springer (2004) 168–185
32. Draheim, D., Lutteroth, C., Weber, G.: An Analytical Comparison of Generative Programming Technologies. Technical Report B-04-02, Institute of Computer Science, Freie Universität Berlin (2004)

Sorting Out the Relationships Between Pairs of Iterators, Values, and References

Krister Åhlander

Department of Information Technology, Uppsala University, Uppsala, Sweden
krister.ahlander@it.uu.se

Abstract. Motivated by a wish to sort an array A while simultaneously permuting another array B, iteration over *array pairs* (A, B) is considered.

Traditional solutions to this problem require an adaption of either the algorithm or of the data structure. The generic programming approach described in this paper involves the construction of an iterator adaptor: an iterator pair. The different approaches are implemented in C++ and compared with respect to flexibility and performance.

Our design is also compared with another iterator-based design. When examining our solution, we identify the relationship between a reference type and a value type as an independent abstraction. We find that a valid "reference type" to a value type `T` is not necessarily `T&`. The reference pair developed in this paper serves as an example of a reference type which refers to a standard value pair without being a standard reference.

Our understanding of the relationships between iterator pairs, value pairs, and reference pairs, makes our design simpler than the alternative. It is argued that a recognition of these relationships is useful in many other generic programming contexts as well.

Keywords: C++, iterators, reference types.

1 Introduction

A colleague of ours encountered the following sorting problem while developing the commercial finite element software FemLab 3 [1].

> **Motivating problem:** Consider two huge arrays A and B, where, for each index i, the numbers A_i and B_i are related. The task is to sort A while maintaining the inter-array relationship.

There are three fundamentally different approaches to this problem.

Algorithm-based Adapt the algorithm. Based on any sorting algorithm, it is a simple matter to write a special-purpose sorting algorithm which is dedicated to solving this particular situation.

Data structure-based Reorganize the data structure. Instead of having a pair of arrays, data could be organized as an array of pairs. It is then trivial to accomplish the task by reusing a general-purpose sorting algorithm such as C++ `std::sort`.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 342–356, 2005.

Iterator-based Reuse via generic programming (GP) as pioneered by Stepanov and Lee in the design of STL, the C++ standard template library. Construct an iterator adaptor which makes it possible to reuse `std::sort` without redesigning the data structure.

For the developers of FemLab 3, it was not feasible to change the data structure. It would have required a major redesign of the implementation, and it is also likely that it would have degraded the performance of other parts of the program. Regarding the iterator-based approach, my colleague said that he considered the third alternative for a while, but settled for the more pragmatic first alternative.

The motivation behind the present paper is to discuss the third alternative, the iterator-based GP approach. The key idea is to develop a pair iterator, a concept similar to the `zip_iterator` found in Boost [2]. One such solution has previously been developed by Williams [3]. Independently, we designed an alternative implementation. We compare the iterator-based versions with the other approaches with respect to flexibility and performance. The iterator-based approach is of course superior with respect to flexibility, but when it comes to performance we find that the other approaches are faster.

However, the main issue which we want to discuss concerns the insights drawn from comparing the iterator-based designs with each other. For random access iterators, Williams introduces `OwningRefPair` as a value type in order to properly handle dereferencing. In our approach, we introduce pairs of references as a convenient way to accomplish the same task. When studying the C++ standard in more detail, we realize that our solution is not even standard compliant: the standard requires that a random access iterator with value type `T` *should* have a reference type `T&` [4–Ch. 24.1]. Similar restrictions remain also in the discussions of the new iterator types [5]. However, we argue that the assumption that a value type and a reference type always are related in this way is unnecessarily strict. Our notion of a reference pair which *behaves* as a reference to a pair of values illustrates that a value type `T` may well have different reference types, not only `T&`.

Our presentation is organized as follows. Section 2 reviews the basics of C++ iterators. Section 3 recapitulates Williams' implementation. Our approach is presented in Section 4. Section 5 compares the GP approaches with the traditional approaches, particularly with respect to performance. Section 6 elaborates further on the relationships between iterator types, value types and reference types. Finally, Section 7 summarizes our findings.

2 Review of C++ Iterators

In order to make the paper accessible to a broader audience, we commence with a brief review of C++ iterators, containers and algorithms. STL was proposed by Stepanov and Lee in 1993, and soon thereafter adopted by the C++ standard. See, e.g., the preface of [6] for an account of the historical background. Even

though STL has its limitatations, see, e.g., [5], it has proven to be a very useful tool for C++ programmers today.

STL provides generic containers and algorithms for many common programming tasks. The containers and algorithms communicate via iterators. A key feature of STL is the use of C++ templates. This makes the design type safe and thus more robust, as well as fast, because the template instantiation allows for efficient code optimization in link time.

Iterators have different semantics, depending on their category. The iterator categories defined by STL are input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators. The most basic of these iterator categories are input and output iterators, which provide input and output access, respectively, as well as forward traversal. Forward iterators support both input and output access, but are still limited to forward traversal. Bidirectional iterators also support backward traversal, whereas random access iterators, in addition to the requirements above, support random access.

The different categories allow the algorithms of STL to put different requirements on the iterators they use. The `std::copy` algorithm, for example, requires only input and output iterators, whereas for example the `std::sort` algorithm expects random access iterators.

Different containers provide iterators of different categories. An STL `vector`, for instance, provides random access iterators, whereas an STL `list` only provides bidirectional iterators. But STL containers are not the only C++ mechanisms that can be manipulated via iterators. The design of the iterator's interface has been carefully crafted to allow other data structures to be accessed through iterators. An input stream, for example, can be adapted and used as an input iterator, and a standard pointer does, in fact, qualify as a random access iterator.

As a simple example which illustrates the last point, let us consider an array of hundred time stamps (hours and minutes) which is to be sorted. Assume that the array is declared by

```
typedef std::pair<int,int> TimeStamp; // hour and minute
TimeStamp times[100];                 // array of time stamps
```

After proper initialization, it is then a simple matter to sort the times by a call to the standard sorting routine:

```
std::sort(times,times+100);
```

There are, of course, lots of things going on behind the scene. The call to the generic `sort` is instantiated with pointers of type `TimeStamp*`, but the algorithm needs to know more information about the underlying data types, for example the corresponding value type and reference type. In this case, these types are `TimeStamp` and `TimeStamp&`, respectively, and this is figured out by using the auxiliary class `iterator_traits`. Another detail is that this call to `std::sort` expects `operator<` to be used for comparing two arguments of type `const TimeStamp&`. Since a time stamp is nothing but a standard pair instantiated with two integers, this operator is provided automatically, assuming a lexicographical ordering of the first and second members of the pair.

Table 1. The common category for a pair of iterators is deduced from the categories of the underlying iterators. Note that an input iterator and an output iterator can not be paired.

	input	output	forward	bidirect.	random
input	input	N/A	input	input	input
output	N/A	output	output	output	output
forward	input	output	forward	forward	forward
bidirect.	input	output	forward	bidirect.	bidirect.
random	input	output	forward	bidirect.	random

It is, as already mentioned, not the purpose of the present section to explain STL in detail, but rather to review some of the concepts needed to discuss generic approaches to our motivating problem. We also notice that the data structure-based solution sketched in the introduction is very similar to the example of sorting time stamps above. However, it is not always possible or viable to restructure data in this way, which motivates our search for other approaches.

3 Pairing Off Iterators

In this section, we briefly summarize the solution reported by Williams [3]. Interestingly, his motivation for studying the problem was the same as ours, but beside the required support for random access iterators he also handles the other iterator categories. An auxiliary class, `CommonCategory`, is used to find the appropriate iterator category according to Table 1.

Based on the common category of the iterators, the value type and the reference type of the pair iterator, which he calls `PairIt`, are chosen. Consider two iterators `I1` and `I2` whose value types are `T1` and `T2`, respectively. Using various auxiliary classes, Williams creates the value types and the reference types of a pair iterator `PairIt<I1,I2>` as follows.

- If the pair iterator is an input iterator, the value type is a plain pair, `std::pair<T1,T2>`, and the reference type is `pair<T1,T2>&`. On dereferencing, the underlying iterators are dereferenced and an internal pair is created which holds the result. A reference to the internal value is returned.
- If the pair iterator is an output iterator, Williams follows the standard which prescribes that the value type should be `void`[1] The reference type, however, needs to be something different, since assignments to an object of this type should forward the assignment to the underlying iterators. Williams solves this task by introducing an `OutputPair<I1,I2>` class, which holds references to the underlying iterators.
- Williams recognizes that the "biggest headache" is to support forward, bidirectional, and random access iterators. Here, the standard prescribes that

[1] This requirement is unnecessarily strict and a relaxation has consequently been proposed [7–Article 445].

if the value type is `T`, the reference type should be `T&`. Williams resolves these requirements by introducing `OwningRefPair<T1,T2>` as the value type, and consequently `OwningRefPair<T1,T2>&` as the reference type. The class `OwningRefPair<T1,T2>` contains `T1& first` and `T2& second` as public members. These members may refer to external data, or to data kept in an internal buffer. The handling of the internal buffer requires a utility class `RawMem`, in order to avoid the overhead of dynamic memory allocation.

In addition to figuring out the appropriate types to export by the iterator and the handling of dereferencing, the iterator should of course also provide means of traversal etc. These operations are all fairly straight-forward in the literal sense of the word, since they just forward the appropriate call to the underlying iterators. Therefore, we will not discuss the remaining operations further here.

In order to address the motivating problem, it is not necessary to know all the details behind the scene. Listing 1 illustrates how the pair iterator could be used to solve the problem at hand.

After the first call to `std::sort`, `A` is sorted and the elements of `B` are in the order 2, 3, 4, 1, since the inter-array relationship is maintained. Notice that,

```
// A comparison function
bool compFirst(const std::pair<int,int> &l,
               const std::pair<int,int> &r) {
  return l.first < r.first;
}

// A comparison functor
struct CompSecond {
  template<typename T1, typename T2>
  bool operator()(const T1 &l, const T2 &r) {
    return l.second < r.second;
  }
};

int main() {
  // Two "huge" arrays
  int A[] = { 8,5,6,7 }, B[] = { 1,2,3,4 };

  // An iterator pair for the pair of arrays
  IteratorPair<int*, int*> p(A,B);

  // Reuse standard C++ sort
  std::sort(p,p+4,compFirst);
  std::sort(p,p+4,CompSecond());
}
```

Listing 1: The iterator pair concept offers a convenient way to sort a pair of arrays while maintaining an inter-array relationship

since B originally was sorted, B now contains an encoding of the permutation used to sort A. After the second call to std::sort, both arrays are restored into their original state, since the functor used in this call compares with respect to values in the second array.

Williams also observes that the example's usage of a comparison function, as in the first call to sort, requires unnecessary conversions between the value type of the pair iterator, which is OwningRefPair<int,int>, and the standard pair. Therefore, a functor such as Comp with a parameterized binary predicate should be used instead, as in the second call. In Section 4.2, we present another way of handling the comparisons.

4 An Alternative Design

When solving the motivating problem, we arrived at a design where the key idea of an iterator pair is similar to Williams' solution. However, we focused only on random access iterators, and the comparisons between our designs therefore apply to this category only.

The biggest difference between Williams' design and ours concerns the value type and the reference type: We argue that the natural value type for an iterator pair should be a standard pair, and the reference type should be a pair of references:

```
template <class Iterator1,  class Iterator2>
class IteratorPair
{
public:
  typedef typename iterator_traits<Iterator1>::value_type T1;
  typedef typename iterator_traits<Iterator2>::value_type T2;
  typedef typename iterator_traits<Iterator1>::reference R1;
  typedef typename iterator_traits<Iterator2>::reference R2;
  typedef pair<T1,T2>                          value_type;
  typedef const ReferencePair<R1,R2>           reference;
  // ...
};
```

It is not possible to use std::pair<T1&,T2&>, since it would call for the instantiation of the types T1&& and T2&&, and references to references are not allowed. Therefore, we designed the ReferencePair<R1,R2>.

4.1 Reference Pairs and Reference Traits

Reference pairs are developed in parallel with iterator pairs. To obtain appropriate types related to a reference, we find it convenient to also introduce reference traits, see Listing 2. The Reference traits class is similar to the standard iterator traits, and exports a value type and a reference type. In addition, inspired by [8], reference traits also export a parameter type. Listing 3 shows how the reference pair uses the reference traits. Note the usage of Parameter in the constructor. For instance, consider dereferencing of an iterator of type

```
IteratorPair<int*,IteratorPair<double*,char*> >
```

which leads to a reference of type

```
const ReferencePair<int&, const ReferencePair<double&,char&> >
```

The constructor of this type would have `int&` as its first parameter, which is the same as its `R1` type. Its second parameter, however, would be

```
const ReferencePair<double&,char&>&
```

which is not the same as `R2`, but a reference to `R2` instead.

We also point out that the default constructor is private, since a reference pair always should refer to something. Moreover, the assignment is marked as `const`, since the references are not altered, only what they refer to. The importance of these details are emphasized in Section 6.

```
// ReferenceTrait provide typedefs for a reference.
template<typename R>
struct ReferenceTrait {
  typedef typename R::Value     Value;
  typedef typename R::Reference Reference;
  typedef typename R::Parameter Parameter;
};

// For standard reference to type T, default typedefs are provided
template<typename T>
struct ReferenceTrait<T&>
{
  typedef T  Value;
  typedef T& Reference;
  typedef T& Parameter;
};
```

Listing 2: The reference trait is similar to the iterator trait

4.2 Sorting with `operator<`

The classes sketched above can be used for the motivating problem, exactly as in Listing 1. The only change necessary is to use our `IteratorPair` instead of the class `PairIt`, developed by Williams. It is natural to ask, though, if `operator<` which is defined on `std::pair` can be used when sorting a pair of arrays, cf. the example in Section 2. The answer is no, at least not without some extra machinery. The reason is that the sorting algorithm typically makes comparisons between `pivot` which is of value type, and the object `*it` returned upon dereferencing, and this is of reference type. It would require an implicit conversion from `RefererencePair<T1,T2>` to `pair<T1,T2>`, but this is not resolved by the template overloading mechanism [4–Section 14.8.3]. Therefore, we provide parameterized comparison operators between pairs and reference pairs:

```
template<typename R1, typename R2>
struct ReferencePair {
  typedef typename ReferenceTrait<R1>::Value Value1;
  typedef typename ReferenceTrait<R2>::Value Value2;
  typedef typename ReferenceTrait<R1>::Parameter Parameter1;
  typedef typename ReferenceTrait<R2>::Parameter Parameter2;
  typedef pair<Value1,Value2>                          Value;
  typedef ReferencePair<R1,R2>                         Reference;
  typedef const ReferencePair<Parameter1,Parameter2>& Parameter;

  R1 first; R2 second;

  ReferencePair(Parameter1 a,  Parameter2 b)
    : first(a), second(b) { }

  ReferencePair(const ReferencePair& x)
    : first(x.first), second(x.second) { }

  const ReferencePair & operator=(const ReferencePair & x) const {
    first = x.first; second = x.second;
  }

  const ReferencePair & operator=(const Value & x) const {
    first = x.first; second = x.second;
  }

  operator Value () const {
    return  Value (first, second);
  }
  //...
};
```

Listing 3: The reference pair uses the reference trait

```
template<typename T1, typename T2, typename R1, typename R2>
inline bool operator<(const pair<T1, T2>&, const ReferencePair<R1, R2>&);
```

In addition, we provide overloaded comparisons between reference pairs. The extra operators introduced offers a working solution to our problem, without the unnecessary construction of temporary `pair` objects which implicit conversions requires. However, the usage of a C++ functor as in Listing 1 is probably more elegant, since it does not clutter the name space with several overloaded comparison operators.

5 Comparing the Approaches

We have summarized two different iterator-based versions which solve the motivating problem, but how do these compare with the other approaches discussed in Section 1 and with each other?

Below, we first discuss a few obvious advantages with the GP approach. Next, we present performance measurements.

5.1 Advantages of the Iterator-Based Approach

The GP paradigm allows new data structures to reuse algorithms. The following code snippet illustrates how three arrays are permuted simultaneously. The call to `std::sort` will sort the vector `v1` and permute the others accordingly.

```
int arr1[] = { 2,4,1,3 };   vector<int> v1(arr1,arr1+4);
double arr2[] = { 3.1, 2.2, 5.5, 0.1 }; char arr3[] = "RAND";

typedef vector<int>::iterator iterator;
typedef IteratorPair<iterator,IteratorPair<double*,char*> > MyIterator;
MyIterator p(v1.begin(),IteratorPair<double*,char*>(arr2,arr3));

sort(p,p+4);
```

Traditional approaches would have required either a new sorting routine, or a restructuring of data. In either case, we find that the GP approach implies less programming work.

The GP paradigm allows new algorithms to reuse data structures. In our example, the iterator pair and the auxiliary classes were developed in order to reuse the standard `std::sort` for the data structure of the motivating problem. Thanks to the GP paradigm, we do not only solve this task, but we also get the possibility to use the pairs of arrays with other algorithms as a bonus.

As a trivial example, the following code snippet illustrates how the `std::copy` algorithm may be used in order to copy the values in the pair of arrays, A and B, into an array of pairs, C.

```
int A[] = { 2,4,1,3 }; int B[] = { 20,40,10,30 };

typedef IteratorPair<int*, int*> MyIteratorPair;
typedef pair<int, int> MyValue;

MyIteratorPair AB(A,B);
MyValue C[4];

copy(AB,AB+4,C);
```

This additional power of the GP approach is not achieved with traditional approaches.

5.2 Performance

The iterator-based approach clearly has major advantages, but can it compete with respect to performance? To investigate this, we carried out several experiments. Our performance experiments were carried out on a SUN UltraSPARC-IIIi using the GNU compiler version 3.4.3 for Solaris 2.9, with optimization

flag `-O`. For all our experiments, we present minimum time measurements of 5 consecutive iterations, in order to decrease random effects such as irregular work load etc.

To begin with, we had of course to implement the traditional approaches as well. The data-based approach was trivial to implement, see Section 2. We needed to provide comparison functors though, since we do not want the second member of the pair to affect the comparison. The algorithmic-based approach required more work, since we had to be careful to use the same underlying sorting algorithm in all experiments. Based on the library `std::sort` routine, we developed a special version according to the algorithm-based approach. This routine takes three random access iterators as parameters, indicating the beginning and the end of the first array, which holds the keys, and the beginning of the second array, which holds the related data. The g++ "introspective" `sort` routine is a variation of quick sort, but it uses a limit on the recursion depth to avoid the possibility of $\mathcal{O}(N^2)$ complexity [9]. If the recursion limit is reached, the algorithm switches to heap sort, thereby guaranteeing a worst case complexity of $\mathcal{O}(N \log N)$. In our experiments, we use arrays with random data, and it is therefore very unlikely that the the recursion limit is encountered. To simplify the development of our special sorting routine, we decided not to implement the full introspective sorting algorithm, but only the quick sort recursion. In order to make the comparisons fair, we discarded the very few experiments where the recursion depth was reached.

Our first experiment investigates the performance when addressing the original, motivating problem. For different array sizes, we create a pair of arrays with random data. We measure the time to permute both arrays such that the first array becomes sorted, while maintaing the inter-array relationship. Table 2 shows the time measurements in milliseconds for the special sorting routine (Algo), for standard sort of an array of pairs (Data), for our own GP implementation (GP 1), and for Williams GP implementation (GP 2). The results are quite discouraging, since the traditional sorting routines outperform the iterator-based versions. We note though, that our GP version seems more efficient than Williams', probably because of the extra memory handling present in the `OwningRefPair` class.

Since the iterator-based approaches do incur some overhead, we had expected somewhat worse performance for the GP versions, but we were surprised that the degradation was as large as it was. In order to estimate an acceptable—or at least a not easily avoidable—level of performance degradation, we studied the performance drop when using the standard `std::reverse_iterator`, see Table 3. The iterators used are `double*` (Forward), `reverse_iterator<double*>` (Reverse), and finally a reverse iterator of a reverse iterator (Reverse^2). Since the performance degradation is considerable also in this case, we find that we, at least with this compiler, may have to accept the performance degradation of the pair iterators too. A plausible explanation for the performance difference is that the optimizer may keep plain pointers in registers, when they are passed as parameters to functions, whereas this is not possible for more complex types

Table 2. Time measurements (milliseconds) when sorting pairs of arrays of different sizes (N), using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation

N	Algo	Data	GP 1	GP 2
100000	40	50	110	130
200000	110	120	220	270
300000	170	180	350	480
400000	230	260	520	640
500000	300	320	640	830

Table 3. Time measurements (milliseconds) when sorting arrays of different sizes (N), using forward iterators, reverse iterators, and reverse reverse iterators

N	Forward	Reverse	Reverse^2
100000	30	60	50
200000	70	100	110
300000	110	160	170
400000	150	220	220
500000	200	280	260

Table 4. Time measurements (milliseconds) when sorting pairs of arrays of integer keys and images, using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation. The fastest method, Perm, uses the permutation obtained from sorting the keys.

N	Algo	Data	GP 1	GP 2	Perm
1000	890	1080	1130	1080	100
2000	2010	2490	2530	1810	220
3000	3080	2950	3900	3710	320
4000	4180	6310	3580	3250	290
5000	3440	5310	4690	5490	550

Table 5. Time measurements (milliseconds) when sorting pairs of arrays of images, using the algorithmic and data-based approaches as well as the iterator-based approaches, where GP 1 is our implementation

N	Algo	Data	GP 1	GP 2
100	450	480	480	250
200	1040	1060	1150	610
300	1600	1670	1760	1090
400	2440	2450	2490	1380
500	3000	3200	3130	1900

such as pair iterators. This explanation is supported by the fact that Reverse^2 is not much worse than Reverse.

We have found that the special sorting routines are faster than the GP approaches for sorting pairs of simple types, but how do they perform if we sort

arrays of more complex data? As an example, consider the sorting of pairs of integer keys and images, where each image is 256×256 characters. Table 4 shows the results for this case. The special sort routine is—apart from the permutation based sorting algorithm discussed below—still fastest, but the difference is smaller. One reason for this should be the extra overhead of copying images. In this case, we also find that Williams' implementation performs better than ours. We believe that this is due to his handling of temporary storage. For small data types, it incurs some overhead, but this overhead seems to pay off if the underlying data types consume more memory. This trend is even more significant when we sort pairs of arrays of images, see Table 5. In this case we use images both as keys and as data, and the (somewhat artificial) comparison operator compares two images with respect to their total brightness. In this experiment, the GP2 implementation is actually the winner.

Finally, we remark that it is a simple matter, see Listing 1, to use the pair iterator and standard sort in order to obtain the actual permutation when sorting a set of keys. If the objective is to sort an array of keys and a corresponding array of memory consuming images, it is thus easy to first sort only the keys and obtain the permutation, and then use the permutation to sort the images. With this approach we need much fewer calls to image assignment, an operation which is quite time consuming. As seen in the last column of Table 4, this solution is much faster than any of the other approaches.

6 A Discussion on References

When comparing the design of the two GP approaches, we note that our solution is not standard compliant, since the reference type of the iterator pair is not a standard C++ reference to the value type. However, we do not think that it is a mere coincidence that we are able to solve the problem at hand. Our implementation works since the reference pair we use *behaves* as a reference. Our example illustrates that the assumption that a value type `T` only has one valid reference type `T&` is more strict than it has to be. This is similar to the more well-known situation that there may be different pointer types—also known as random access iterators—which refer to the same value type.

Sections 8.3.2 and 8.5.3 of [4] discuss references in detail. To keep the discussion simple, we ignore `const` and `volatile` qualifiers here and suggest that the key properties of a reference to type `T` are the following.

1. A reference must be initialized by an object of type `T`.
2. Changes to the reference affect only the object being referenced.
3. The reference cannot be changed to refer to another object.
4. The reference can be converted to the value type.
5. The value type can be deduced from the reference.

Thus, it is possible to define a *valid reference type* to type `T` as any type whose objects meet these criteria. We then find that the value type `pair<T1,T2>` admits not only the usual `pair<T1,T2>&` but also `const ReferencePair<T1&,T2&>` as a valid reference type.

As demonstrated by the time measurements in the previous section, our usage of `ReferencePair` as the reference type to a pair iterator seems to perform better, at least for simple value types, than the alternative implemented by Williams. We also think that our design is simpler, since it is not necessary to introduce an auxiliary class `RawMem` for handling an internal buffer. In addition to these arguments, we would like to motivate the soundness of our design by the following argument.

Let `I1` and `I2` be two iterator types, whose value types and reference types are `T1`, `R1` and `T2`, `R2`, respectively, and assume that the value type of `R1` is `T1` and the value type of `R2` is `T2`. Let us now consider the pairing `Pair`[2] of two types to construct a new type. Thus, we may construct `Pair<I1,I2>`, `Pair<T1,T2>`, and `Pair<R1,R2>` as new types. We now argue that relationships between types should be preserved, in such a way that `Pair<I1,I2>` should have `Pair<T1,T2>` as value type and `Pair<R1,R2>` should be its reference type. Moreover, `Pair<R1,R2>` should have `Pair<T1,T2>` as its value type. This argument is similar to the formal definition of a functor, which maps objects (in our case types) to objects, and morphisms (relationships between types) to morphisms. The point we make is that `IteratorPair<I1,I2>`, the standard `std::pair<T1,T2>`, and `ReferencePair<R1,R2>` preserve these relationships.

We therefore find the recognition of the reference relation as an entity in its own right well motivated. This extra level of indirection may also be useful in other contexts. As a simple example (cf. the discussion in [10–Ch. 2.4]), we may use the reference concept in order to write a generic `Swap` (or, perhaps a `ref_swap`) routine, as shown in Listing 4. The template meta function `ValueOfType` is here used to deduce the appropriate value type corresponding to a valid reference. It corresponds to the type transformation `remove_reference` found in the Boost metaprogramming library. Note that this version of `Swap` also handles swapping of two objects of the same value type referred to by different reference types. This situation could also be resolved by parameter overloading. This approach is however not very practical, since the number of overloaded `swap` functions grow exponentially with the number of valid reference types to a given value type.

7 Conclusions

We have investigated different approaches to the problem of sorting a pair of arrays. We find that the algorithm-based version performs best, but it requires the development of a dedicated sorting routine for this particular situation. Running time is of course only one software metric, and we stress that flexibility, robustness, and programmer time in many situations are more important. The data-based version also performs well and it is much easier to implement, if standard generic tools are used. The disadvantage is that the data-structures have to

[2] We deliberately use capital P here, in order to distinguish this concept from `std::pair`.

be restructured, which often means that this approach is inadequate. Therefore, we advocate the iterator-based approach.

We present the design of two different iterator-based approaches, ours, and an implementation by Williams [3]. We acknowledge that his implementation is more comprehensive, since he addresses all iterator categories, not only the

```
template<typename V>
struct ValueOfType {
  typedef V type;
};

template<typename V>
struct ValueOfType<V&> {
  typedef V type;
};

template<typename R1, typename R2>
struct ValueOfType<const ReferencePair<R1,R2> > {
  typedef pair< typename ValueOfType<R1>::type,
                typename ValueOfType<R2>::type > type;
};

template<typename R1, typename R2>
inline void
Swap(R1& a, R2& b)
{
  const typename ValueOfType<R1>::type tmp = a;
  a = b;
  b = tmp;
}

int main {
  IteratorPair<int*, int*> ip1, ip2;
  pair<int,int> vp3;

  // ...

  Swap( *ip1, *ip2 );
  Swap( *ip1, vp3 );
  Swap( vp3, *ip1 );
  Swap( (*ip2).first, ip2->second );
}
```

Listing 4: The `Swap` routine is generic with respect to valid references. The main program illustrates four different calls to `Swap`. The first call to `Swap` shows that `ReferencePair` behaves as a reference. The second and third calls illustrate calls where different reference types are used to swap the same value type. The last call simply swaps two integers

random access iterator category required to solve the motivating problem. However, we find that our approach to treating random access iterators is simpler, because we do not need to introduce an auxiliary class for managing an internal memory buffer.

We notice, though, that our implementation is not standard compliant. The reason for this is that the standard tacitly assumes that there is a one-to-one correspondance between value types and reference types. We believe that this is too strict. In our opinion, there may be several different types which can refer to the same value type, and this is the insight which simplifies our design. We think that this notion should be useful also in other situations where the objective is to develop generic solutions.

Acknowledgements

I thank Stefan Engblom for bringing my attention to the motivating problem. I also thank Adis Hodzic for valuable comments on the manuscript.

References

1. Comsol homepage, `http://www.comsol.com/`.
2. Boost MPL homepage, `http://www.boost.org/libs/mpl/doc/index.html`.
3. Williams, A.: Pairing off iterators. Overload (2001) Avaliable at `http://web.onetel.com/~anthony_w/cplusplus/pair_iterators.pdf`, 2005-04-07.
4. The C++ Standard, Incorporating Technical Corrigendum No. 1. 2 edn. John Wiley & Sons, Ltd (2003)
5. Abrahams, D., Siek, J., Witt, T.: New iterator concepts (2003) `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1550.htm`.
6. Musser, D., Saini, A.: STL Tutorial and Reference Guide. Addison-Wesley (1996)
7. C++ standard library active issues list (revision 35) (2005) `http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html` 2005-03-04.
8. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
9. Musser, D.: Introspective sorting and selection algorithms. Software–Practice and Experience **8** (1997) 983–993
10. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and beyond. Addison-Wesley (2005)

Preprocessing Eden with Template Haskell

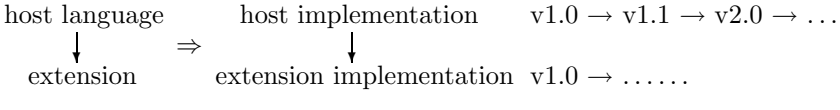
Steffen Priebe*

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik,
Hans-Meerwein-Straße, D-35032 Marburg, Germany
priebe@mathematik.uni-marburg.de

Abstract. Extending a programming language by new language constructs often implies extending its compiler by additional machinery. To reduce the complex interweaving of compiler and extension implementations we present a simple and modular concept of lifting the often needed additional preprocessing out of the base compiler implementation. Avoiding the introduction of standalone tools, this preprocessor framework for extensions of Haskell is designed as a separate portable library of monadic preprocessing functions based on Template Haskell. Additional preprocessing passes expressed in this framework can then much easier be carried along the series of ever advancing base compiler versions. Taking Eden, a parallel programming extension of Haskell, as an example we show that besides achieving improved portability and reusability pass code sizes can be reduced considerably.

1 Introduction

Suppose you have designed a new, fancy domain-specific extension of Haskell [1], possibly an extension for providing easy data base access, for interfacing to foreign languages, for providing faster computations by exploiting parallelism, or for mobile computing. How do you implement your extension? Developing a whole new compiler would mean reinventing the wheel, while extending an existing Haskell compiler most often produces a deep entanglement of compiler interiors with domain-specific implementation parts. That would result in reduced portability and maintainability, which is especially problematic if the base compiler is subject to frequent version changes which usually have to be reproduced:



Having all your implementation extensions contained in a traditional library seems like a nice idea. But a mere collection of subroutines would not be powerful enough, since such extensions usually also need changes to the runtime system and new optimisation passes run by the compiler. Following the idea of *active libraries* [2] one solution to this could be to use meta-programming tools to separate the domain-specific implementation from the compiler while

* Supported by a PhD grant of *Evangelisches Studienwerk Villigst e.V.*

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 357–372, 2005.

expressing it as a library. This envisions having a compiler for a base language and extensions of that language implemented as a set of attached libraries.

Eden [3] is such a domain-specific extension of Haskell which introduces constructs for parallel programming. Its first implementation consisted of a largely modified *GHC* [4] which was very hard to maintain over GHC version changes. To avoid these problems parts of the Eden-specific implementation have recently been pulled out [5,6] of the GHC. Among others one thing that still remains to be separated from the GHC is the preprocessing machinery for analysing and transforming Eden code which still resides amidst GHC's original code. This can also often be the case for other domain-specific extensions.

Besides writing a library one could also suggest building a standalone tool for preprocessing the sourcecode which can then be fed into the compiler manually. Disadvantages include: the introduction of an extra tool which needs additional care, manual and possibly erroneous operation by the user, the inability to use compiler facilities (functionality needs to be rewritten) and compile-time information and therefore probably a large reimplementation of compiler machinery. Much more promising instead is a preprocessor library which remains separate but can be glued on top of the compiler. The same has recently been done for the foreign function interface *Greencard* by making the transition from an external tool [7] to an integrated (active) library [8].

Contributions. In this paper we will use compile-time meta-programming facilities provided by *Template Haskell* [9] to implement a series of pre-GHC preprocessing passes on domain-specific source code. The preprocessor code will be automatically inserted into the source program and run prior to the compilation of the actual program, meaning that an enriched program is created which preprocesses itself. We will show how the new implementation (forming an active library) can be separated from the GHC implementation, while only a small hook for the invocation of the preprocessor needs to remain. The scheme shown is generally applicable and in no way tied to the Eden implementation. Advantages include a simplified addition of preprocessing passes, shorter and more concise pass code, and enhanced maintainability. The scheme will be used to rewrite and shorten existing passes and to express a simple kind of generic parallel programming in Eden.

Plan of the paper. After introducing Template Haskell in the next section we describe the preprocessor itself (Sect. 3), its pass construction mechanism (Sect. 4), and its main internal loop (Sect. 5). In Sect. 6 we apply the whole scheme to the parallel functional language Eden. Finally Sect. 7 outlines related work and Sect. 8 concludes.

2 The Tool

Active libraries can be built using *Template Haskell* [9], which is a typesafe compile-time meta-programming extension of Haskell built into the GHC. Basically Template Haskell introduces a second layer of execution by allowing to label Haskell expressions as "to be executed by interpretation at compile-time". This

means that one can insert Haskell code that is meant to be run at compile-time into regular Haskell code:

The result of this *splice* expression then has to be Haskell code described in an abstract syntax which will replace the splice and be embedded as runtime Haskell into the surrounding code. This corresponds to classical macro expansion, except that the newly generated Haskell code will also be successively typechecked. This is possible because splices are expanded by the GHCi interpreter during the typechecking phase.

A set of data structures forming an abstract syntax of Haskell is defined to be able to handle Haskell code as a value. There are types for patterns (`Pat`), expressions (`Exp`), declarations (`Dec`), types (`Type`) and so on. Depending on its position the code inside a splice has type `Q Exp` (part of a bigger expression) or `Q [Dec]` (top-level declarations). The `Q` monad is introduced among other things for encapsulating the generation of fresh names. A simple case expression for instance could then be described like this:

```
(CaseE (AppE (VarE "f") (VarE "x")) [...]) :: Exp

[| case (f x) of ... |] :: Q Exp
```

The first row shows pure abstract syntax while the second row uses the *quasi-quotation* `[|.|]` for automatic transformation of user-legible code into abstract syntax. Quasi-quotation occurs inside a splice. Both versions can be used although the first one is more expressive than the second. The same can be done for declarations using `[d|.|]`. A binary tree declaration could then look like this:

```
[DataD [] "Tree" ["a"]
  [NormalC "Leaf" [(NotStrict,VarT "a")], NormalC "Node" ...] []
] :: [Dec]

[d| data Tree a = Leaf a | Node ... |] :: Q [Dec]
```

Example: Consider writing a function `select` which returns the ith value of an n-tuple such that `$(select 2 4) (a,b,c,d)` returns `b` (see [9]):

```
select :: Int -> Int -> Q Exp
select i n = [| \ x -> $(return (CaseE (VarE "x") [alt])) |]
  where alt  = Match pat rhs []
        vars = ["v"++(show j) | j <- [1..n]]
        pat  = TupP (map VarP vars)
        rhs  = NormalB (VarE (vars !! (i-1)))
```

A lambda abstraction is generated which contains a `case` for deconstructing the tuple into its components v_1 to v_n before v_i is selected and returned. Note the nesting of another splice into the quasi-quotation.

Quasi-quotation cannot only be used for the simple construction of code but also for its deconstruction as the contained code will just be translated into abstract syntax. Therefore we can write

```
do { abssyn_case <- [| case (f x) of ... |]; ... }
```

to extract the abstract syntax representation of the `case` expression which can then be used inside the `Q` monad and be spliced back in modified form. The same applies to declarations which can be decomposed by the `[d|.|]` operator.

In summary Template Haskell makes it possible to write Haskell programs which modify themselves at compile-time, which is exactly what we need.

3 The Preprocessor

Having introduced our meta-programming tool we will now turn to building the preprocessor. What we want to achieve is a preprocessor which takes a program, applies a series of preprocessing steps to it and places back the result. We want the preprocessor implementation to be separated from the base compiler implementation but at the same time have it glued closely enough to it to avoid getting an extra tool.

3.1 Overview

We have seen that the quasi-quotation mechanism of Template Haskell can also be used to deconstruct declarations and expressions. Then why not let it embrace and decompose a whole program? This suggests the following basic scheme for preprocessing code with Template Haskell:

1. Textually embrace the given source code with quasi-quotes to get an abstract syntax representation of the source code.
2. Let predefined preprocessing machinery work on the extracted code given in abstract syntax (provided by the base compiler's parser).
3. Surround all this with a splice which delifts the modified source from abstract syntax back to regular Haskell code.

How can we operationally integrate that into a compiler? Simply by modifying the compiler to textually insert two small predefined code blocks into the source program before anything else happens. The source program would then carry around the preprocessing code for modifying itself through the compilation. The preprocessor then gets triggered before other regular compilation stages are started.

In Fig. 1 the transition of regular source code to an identical code with embedded preprocessing is shown. The two additional preprocessing code blocks contain `import` statements to import the Template Haskell module, a `Stager` module which contains the actual preprocessor functions, and a `Tools` module. User imports remain untouched. The remaining code is contained inside quasi-quotation brackets. After extraction into an abstract syntax representation `ds` it

has to be decided which preprocessing passes will be run on the code. External flags are read by `getFlags` and fed into `buildPasses` which will build a list of preprocessing passes. `doLoop` then runs these passes on `ds` producing the modified code `ds'`. After the loop has ended an announcement is printed that all the following messages belong to the underlying Haskell compiler. At last, the modified code `ds'` is returned and reinserted by the surrounding splice.

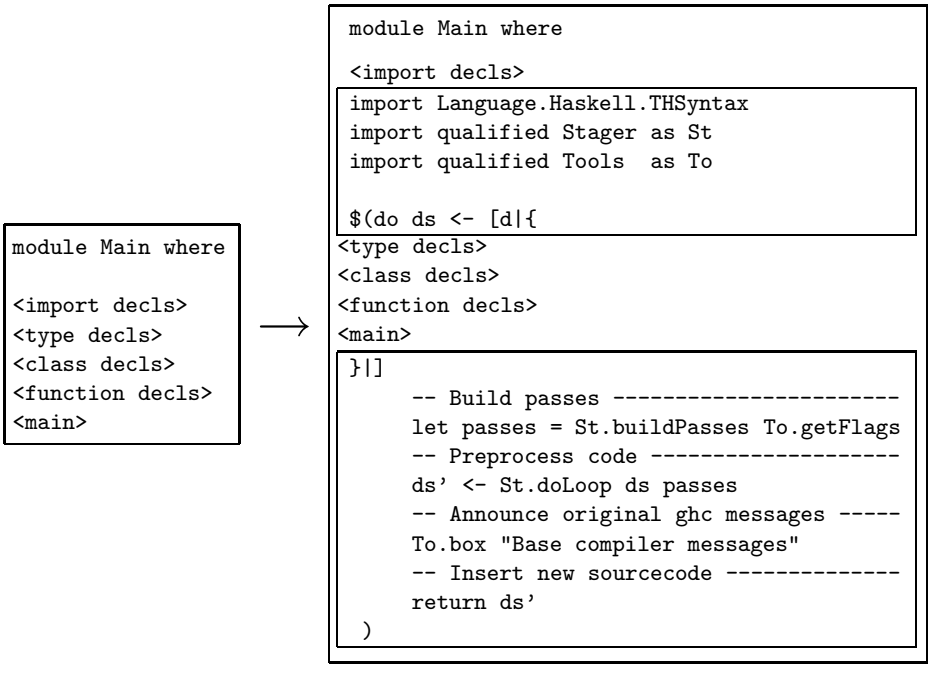


Fig. 1. Transition from input source code to code with embedded preprocessor

This solution permits the smooth integration of the preprocessor into the compiler implementation with only slight modifications while achieving technically a complete separation. In the following subsection we will discuss implementation aspects as well as advantages and drawbacks of the approach.

3.2 Technical Details

Figure 2 gives an overview of the workflow inside the GHC. The stages can be roughly divided into the sections analysis, transformation and synthesis. Until now additional analysis and transformation passes were usually placed in the transformation section and worked on the Core syntax representation, which is a subset of Haskell plus explicit type annotations and resembles the polymorphic lambda calculus. Due to complex functions and data structures for handling Core, inserting an additional pass is a complex task. And with desugaring already

having taken place it is additionally very hard to issue meaningful comments if necessary because the reference back to the original code is lost. A better place

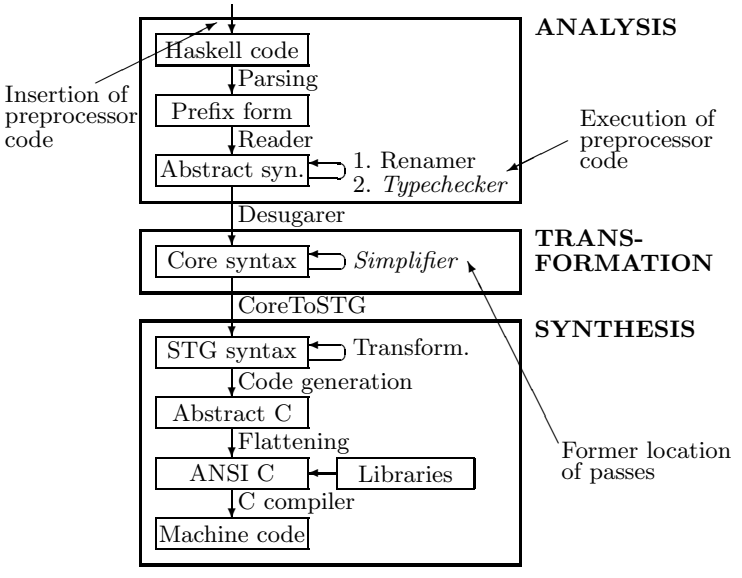


Fig. 2. GHC workflow diagram with changes for preprocessor

for domain-specific optimisation passes is the analysis section where the full source code is still available and for example error messages of the preprocessor can contain much more accurate position descriptions. For domain-specific extensions it is especially important that their high-level constructs have not already been boiled down to less meaningful constructs. Our new preprocessor code will be inserted into the source program before any other GHC stage has started.

But at the moment there are also three drawbacks to the taken approach which partially are related to Template Haskell and may be removed in a future version:

1. Quasi-quotation cannot handle source code which completely relies on layout. Braces and semicolons have to be used sometimes.
2. Regenerating the modified code destroys the internal location tracking of GHC. Regular errors discovered by GHC will not contain a line number but only the message `<at compiler-generated code>`.
3. Since the preprocessor acts outside the base compiler's passes, it has no access to informations (like types) gained by the base compiler. If these are required for an earlier pass, they have to be provided.

At the bottom line there is a clear trade-off: On the one hand we are able to work on richer syntax, while on the other hand we have to face technical restrictions.

4 The Passes

We have seen how the preprocessor is attached to the source program and how it is run. But what should it do when it is run and how can that be specified?

4.1 Stateful Monadic AST Traversal

Quasi-quotation delivers the program to be preprocessed as an abstract syntax tree (AST) of type `[Dec]`. Therefore we define a general type class `Traverser` (see Fig. 3), which contains functions for the recursive traversal of that tree. For each part of the mutually-recursive abstract syntax (`[Dec]`, `Dec`, `Exp`, ...) the class contains a corresponding transformation function (`tDecs`, `tDec`, `tExp`, ...). The `tMain` function is the central starting point which eventually calls `tDecs`. By default all these functions are defined to return the identical syntax tree. As later each instance will represent a preprocessor pass, selected functions will be overloaded to examine or modify the tree.

```
class (MonadState m s) => Traverser m s where
  -- Functions ------------------------------------------------------
  tName  :: m s -> String
  tMain  :: m s -> [Dec] -> Q ([Dec], [Pass])

  tDecs  :: [Dec]  -> m [Dec]          tDec  :: Dec    -> m Dec
  tDecs' :: [Dec]  -> m [Dec]          tDec' :: Dec    -> m Dec

  tBody  :: Body   -> m Body           tExp  :: Exp    -> m Exp
  tBody' :: Body   -> m Body           tExp' :: Exp    -> m Exp  ...

  -- Defaults -------------------------------------------------------
  tName _            = "Identity"
  ...
  tExp               = tExp'
  tExp' (VarE vname) = VarE vname                  -- variable
  tExp' (AppE e1 e2) = do e1' <- tExp e1           -- application
                          e2' <- tExp e2
                          return (AppE e1' e2')
  tExp' (LetE ds e)  = do ds' <- mapM tDec ds      -- let
                          e'  <- tExp e
                          return (LetE ds' e')                ...
```

Fig. 3. Monadic stateful traversal class `Traverser` (layout due to space limits)

To avoid extensive redefinitions when overloading only one alternative of a transformation function, each traversal function is split into two layers. For example the function `tExp` is accompanied by a second function `tExp'`. By default

`tExp'` is defined to continue the AST traversal without making any changes itself while `tExp` is defined to immediately call `tExp'`. Within a `Traverser` instance, one overloads `tExp` to implement modifications of an alternative of `Exp` (for example `LetE`) and refers to the default `tExp'` definitions for all other alternatives by also declaring `tExp x = tExp' x`.

For most preprocessing passes an internal state is needed. Therefore the whole traversal is based on a predefined state monad `(ST s) s` (see [10], [11]) which is shown in detail in Fig. 4. Functions for reading and changing the state are predefined and can be used in any traversal function, which are all defined monadically. The function `tMain` starts the pass by calling `tDecs` with the monad-specific starting state; it returns the resulting code and additional passes created by the current pass for subsequent execution and wraps up both in Template Haskell's `Q` monad.

```
data ST s a = ST (s -> (a, s))

class (Monad m) => MonadState m s | m -> s where
   get    ::              m s                 -- read state
   put    ::         s -> m ()                -- write state
   update :: (s -> s) -> m s                  -- update state by function
   get = update id

instance Monad (ST s) where
  (ST m) >>= f  = ST (\s -> let { (v, s') = m s; ST m' = f v } in m' s')
  return v      = ST (\s -> (v, s))

instance MonadState (ST s) s where
   put s    = ST (\_ -> ((), s  ))
   update f = ST (\s -> (s,  f s))
```

Fig. 4. State monad definition

This completes the definitions needed to define a new preprocessor pass. In total, the typical process of designing a pass would be to:

1. define a datatype `s` for representing the internal state of the pass
2. define an instance `Traverser (ST s) s`, overload functions to implement the pass, use `get`/`put`/`update` inside these to work with the internal state

In Sect. 6.2 two example passes will be discussed in detail.

4.2 Global Actions

Many preprocessing passes only modify the abstract syntax tree very locally. For instance for a typical `let` transformation in Haskell only the `LetE` alternative of the transformation function `tExp` would have to be overloaded. On the other

hand in the same passes one often needs to insert actions on the internal state which are triggered on *every* part of the syntax tree. In the `let` transformation one would for instance like to determine *where* each transformation happened; this output of the form:

```
in function 'f', in alternative '(TreeNode l r)', in 'case': let
```

has to be collected from the place where the `let` transformation occured back up to the AST root. This would result in the need to overload almost each transformation function and `update` the pass state by location information. To avoid that, we extend the `Traverser` class as shown in Fig. 5. As an example, only the extension of `tExp` for the alternative `AppE` is shown; all other cases are extended in the same way. Before descending into a branch of a source tree, the func-

```
class (MonadState m s) => Traverser m s where
  -- Functions ------------------------------------------------------
  ...
  tActionPre  :: AbsSyn -> m ()         tActionPost :: AbsSyn -> m ()

  -- Defaults -------------------------------------------------------
  tActionPre  _ = return ()             tActionPost _ = return ()
  tExp' e@(AppE e1 e2) = do tActionPre (AS4 e)
                            e1' <- tExp e1
                            tActionPost (AS4 (AppE e1' e2 ))
                            tActionPre  (AS4 (AppE e1' e2 ))
                            e2' <- tExp e2
                            tActionPost (AS4 (AppE e1' e2'))
                            return (AppE e1' e2')                ...
```

Fig. 5. `Traverser` class extended by global actions (layout due to space limits)

tion `tActionPre` is evaluated; after finishing the branch traversal `tActionPost` is called. (`AbsSyn` is defined as a collection of `Dec`, `Exp`, etc. to avoid having to define `tAction` functions for each part of abstract syntax.) Both are by default defined to leave the state unaltered, but can be overloaded to house a *set* of global actions which is run before and after entering a node of the syntax tree. A single action is run by `runAction` (see Fig. 6). As each action usually acts only on one part of the state, this action has to be lifted to the full state by applying the identity function to the remaining parts. The lifted function can then be used to update the full state. An action set is run by sequencing a list of `runAction` calls in an overloaded version of `tActionPre` or `tActionPost`.

All this allows us to define global actions separately and to attach selected ones to a preprocessor pass. Predefined actions include: collecting position in code, keeping an indentation level for output, accessing predecessor nodes in the AST, and others. See Sect. 6.2 for an example on how to attach action sets to a pass.

```
runAction :: (MonadState m b) =>
             AbsSyn -> ( AbsSyn->(a->a), (a->a)->(b->b) ) -> m ()
runAction x (action, lift) = do { update (lift (action x)); return () }
```

Fig. 6. Lift action into full state, run action, update state

5 The Loop

We have seen in the two previous sections how the preprocessor is started and how a preprocessing pass can be coded. But how can we set up and execute a series of passes? We start by defining a list of passes (see Fig. 7), which in this case contains a pass for the automatic derivation of class instances and a simplification pass.

```
type Passes = [Pass]
data Pass   = forall m s . (Traverser m s) => Pass (m s)

passes :: Passes           -- Example
passes = [Pass ( return (Derive  ([],[])) :: (ST Derive)   Derive   ),
          Pass ( return (Simplify []    ) :: (ST Simplify) Simplify )]
```

Fig. 7. Definition of passes

Preprocessing is started by a call to `doLoop` (see Fig. 1 and Fig. 8) which applies the passes `ps` to the given code `sc` by calling `doPasses`. The total number of performed passes is shown. `doPasses` successively applies each pass by first announcing its start and then calling its specific version of `tMain`.

```
doLoop :: [Dec] -> Passes -> Q [Dec]
doLoop sc ps = do (sc',n) <- doPasses sc 0 ps
                  To.box ((show n) ++ " pass(es) performed.")
                  return sc'

doPasses :: [Dec] -> Int -> Passes -> Q ([Dec], Int)
doPasses sc n []            = return (sc, n)
doPasses sc n ((Pass p):ps) =
  do To.box ("Pass " ++ (show n) ++ ": " ++ (tName p))
     (sc', newps) <- tMain p sc
     doPasses sc' (n+1) (newps ++ ps)
```

Fig. 8. Loop functions `doLoop` and `doPasses`

New passes created by the current pass are executed right after the the current pass has ended. This could for example be used to tidy up code by

invoking a simplification pass after finishing an inlining pass. Note that there is no automatic bail out mechanism, which means that the pass designer has to make sure that the pass list terminates.

6 The Application

As a case study we are going to describe the application of the preprocessing scheme to the implementation of the parallel functional language *Eden*. After introducing the language and presenting two useful passes we will use the preprocessor to provide a simple kind of static generic parallel programming.

6.1 Eden

In few words: *Eden* [3] extends Haskell with means for relocating the evaluation of a function application to another network node, enabling the evaluation of several expressions in parallel. A function embedded in a *process abstraction* by applying

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
```

can be run in parallel to the continuing evaluation of its parent expression on another processor by applying to its arguments a special application operator

```
(#) :: (Trans a, Trans b) => Process a b -> a -> b
```

The `Trans` context ensures that for both argument and result types corresponding low-level sending and receiving functions exist. This part of the Eden implementation has already been separated from the GHC implementation as far as possible and expressed as a Haskell module. Being defined in Haskell, Eden language constructs are represented within Template Haskell's abstract syntax tree as normal Haskell functions. `(#)` is implemented via the auxiliary function:

```
createProcess :: (Trans a,Trans b) => Process a b -> a -> Lift b
```

Its result is lifted into the data structure `data Lift a = Lift a` to keep local demand (delivers real parallelism) instead of waiting for the process to return (distributed sequentialism) because of waiting for the result (see [12]).

6.2 Two Passes for Eden

For controlling process creation order and steering evaluation depth in Eden one frequently needs functions for controlling evaluation of data structures based on the `seq` library function [12]. In Eden as well as in Glasgow Parallel Haskell [13] these functions are united in a type class called `NFData`. Instances of this class usually have to be derived by hand, a tedious and error-prone task. Figure 9 shows a pass which derives instances of `NFData` for every `data` declaration found (and selected base types like tuples) and adds it to the program. Functions `xi1`, `xi2`, and `xi3` denote different evaluation depths; the most common `xi2` (also known as `rnf` or `deepSeq`) demands the *spine* of a data structure which means that the recursive structure is forced without touching anything else.

```
newtype DeriveNFData = NFData [Dec]  -- State: List of new instances

instance Traverser (ST DeriveNFData) DeriveNFData where
  tName _    = "Derive NFData"
  tMain p sc = do let startstate           = NFData []
                  let ST f                 = tDecs (sc ++ basetypes)
                  let (_, NFData insts)    = f startstate
                  return (sc ++ insts, [])
  tDec d@(DataD c name vars cons ders) =
    do tActionPre (AS1 d)
       update (\(NFData insts) -> NFData (insts ++ inst))
       tActionPost (AS1 d)
       return d
    where inst  = [InstanceD ctxt iName decs]
          ctxt  = infer_context "NFData" d
          iName = AppT (ConT "NFData") typ
          typ   = foldl (\z v -> AppT z (VarT v)) (ConT name) vars
          decs  = [FunD "xi1" [derive_xi1      con | con <- cons]] ++
                  [FunD "xi2" [derive_xi2 name con | con <- cons]] ++
                  [FunD "xi3" [derive_xi3      con | con <- cons]]
          derive_xi1 :: Con -> Clause
          derive_xi1 (NormalC cName cElems) =
            (Clause [ConP cName pats] (NormalB rhs) [])
            where pats  = [WildP | _ <- [1..(length cElems)]]
                  rhs   = TupE []
          ...
  tDec x = tDec' x
```

Fig. 9. Outline of `NFData` instance derivation

For every `data` declaration a corresponding instance `inst` of `NFData` is derived and saved in the pass state; after finishing the pass, this list of instances is added to the original source code `sc`. The type context `ctxt` of the instance is determined by a separate function `infer_context`, which solves a context equation. After defining the instance name its three functions `xi1`, `xi2`, and `xi3` are generated and inserted as a list of declarations. For each of these functions and each constructor alternative a corresponding function `derive_xi{1,2,3}` is called to produce code handling the alternative.

The second example is a transformation pass shown in Fig. 10. The *eager transformation* [3] is a `let` transformation exercising more demand on let-bound process applications providing earlier process creation and thus better parallelism. Until now this transformation was implemented as a transformation of Core syntax. By expressing it as a preprocessor pass the code size has been reduced from around 300 lines to less than 30 lines, mostly by saving the effort of rewriting compiler front-end mechanisms. As the transformation itself is very similar to the previous example, we will only show the main functions and one

```
newtype Eager = Eager (Int, ([String],[String]))

instance Traverser (ST Eager) Eager where
  tName _    = "Eager transformation"
  tMain p sc =
    do let startstate = Eager (0, ([],[]))
       let ST f = tDecs sc
       let (sc', Eager (count,(positions,_))) = f startstate
       To.printToScreen positions
       return (sc', [])
  tExp e@(LetE ds e1) =
    do ds' <- mapM tDec ds
       e1' <- tExp e1
       ...  -- perform transformation, generate ds'' and e''
       let e' = (LetE ds'' e'')
       tActionPost (AS4 e')
       return e'
    where ...
  tExp x             = tExp' x
  tActionPost x      =
    sequence_ [ runAction x (aPos, liftPos) ]
    where liftPos :: (([String],[String]) -> ([String],[String])) ->
                     (Eager -> Eager)
          liftPos f (Eager (c, ps)) = Eager (c, f ps)
```

Fig. 10. Outline of eager transformation for Eden

attached global action `aPos` for saving the position where a transformation has been carried out (see Sect. 4.2).

The action `aPos` (see Fig. 11) itself adds textual information about the position of the traversal to the internal state. As the transformation can be triggered many times during a traversal, we need to be able to keep a list of position descriptions as a list of strings. When for example a function is encountered, a corresponding position description is added as a prefix to every position description kept in the state. We add it as a prefix since the action is triggered during an upwards traversal.

```
  aPos :: AbsSyn -> (([String],[String]) -> ([String],[String]))
  aPos (AS1 (FunD name cs)) =
    \(fin,act) -> (fin, aPosAdd ("in function "++name++", ") act)   ...
  aPos _ = id
```

Fig. 11. Excerpt of action for position tracking

6.3 Generic Programming in Eden

Generic programming [14] is all about being able to define functions which work not only for arbitrary element types (parametric polymorphism) but for arbitrary data structures (structural polymorphism). Imagine generalising the `length` function on lists to a generic `size` function. The generic version can then be specialised to show data type specific behaviour. Using the preprocessor we can model a generic parallel map function working on data structures of kind $(* \to *)$. Figure 12 first shows a generic `zipWith` class, instances of which will be automatically generated for base types and all types occuring in the source program by a deriving pass. Then the generic parallel map is defined which takes a structure of processes and a structure of values, zips these together, exerts enough demand to create all processes in parallel by applying `xi2`, and delifts the resulting structure. This function is very useful in Eden and eliminates the need to write that function over and over again for different data structures.

```
class GZipWith1 t where
  gZipWith1 :: (a -> b -> c) -> t a -> t b -> t c

gParMap1 :: (NFData (t (Lift b)), GZipWith1 t, Functor t,
             Trans a, Trans b) =>
            t (Process a b) -> t a -> t b
gParMap1 ps vs = let papps = gZipWith1 createProcess ps vs
                 in (xi2 papps) `seq` (fmap deLift papps)
```

Fig. 12. Generic parallel map in Eden

7 Related Work

The work presented has been inspired by the work of Lynagh on Template Haskell [15,16]. Both papers discuss ways of manipulating Haskell code which are extended to a separate framework in this paper. Seefried et al. investigate in [17] an approach similar to ours: Template Haskell is used to implement and optimise an extension of a host language.

Also related to this work is the approach by Peyton Jones et al. in [18], where so-called RULES pragmas can be inserted into the program which describe source-to-source transformations executed by the compiler. The difference to our approach is that these are restricted to function applications and cannot work on expressions or whole programs. Another alternative [19] is developed by Tolmach et al. which aims at defining a common *external Core syntax* format to help developers write external optimisation passes for the GHC.

The idea of "separation of concerns" has been carried on by Veldhuizen et al. in their work on *active libraries* [2,20]. This relates to our approach of separating host compiler and extension implementations by pulling the latter into a separate meta-programmed library.

Norell and Jansson [21] introduce generic programming by generating functions for translating every data type into a general representation on which functions can be defined generically. They also plan to use Template Haskell to express this code generation as a GHC extension. The "boilerplate" approach [22] is another possibility of generic programming and has recently been integrated into the GHC: Often nonrelevant traversal code has to be written to walk through a data structure until a function can be applied to a certain part of that structure. This work can be saved by using the automatically generated generic `gmapT` function which traverses arbitrary data structures. Function application is only triggered if the right part of the structure is encountered.

8 Conclusion

We have presented a monadic preprocessing scheme for extensions of the functional language Haskell implemented on top of the Glasgow Haskell Compiler. Being neither an extra tool nor a complicated hack of the underlying compiler our approach is both general and portable: Following the idea of active libraries it has been implemented via meta-programming in Template Haskell and designed as a separate library. Each pass is defined by creating an instance of a monadic class over the state of the pass. Actions global to the whole abstract syntax tree are predefined separately and can easily be attached to a pass. This library, together with a set of passes, can be ported from one base compiler version to the next with only minimal compiler modifications. Our approach is therefore especially useful for compiler writers as it makes their preprocessor implementation much more straightforward to port and to maintain. As an example we have applied this scheme to the parallel Haskell dialect Eden where we observed a considerable code size reduction and improved portability.

Future work. Future work areas include more preprocessing passes for Eden like analysis and runtime information passes. The main focus however will be a deeper investigation of generic parallel programming for Eden.

Acknowledgements. The author would like to thank Rita Loogen and Jost Berthold for discussions on the topic, Holger Gast for pointing out the work of Todd Veldhuizen on Active Libraries, and anonymous referees for their helpful comments.

References

1. Peyton Jones, S., et al.: Haskell 98: A Non-strict, Purely Functional Language (2003) See: `http://www.haskell.org/definition`.
2. Czarnecki, K., et al.: Generative Programming and Active Libraries. In: Dagstuhl Seminar on Generic Programming. Volume 1766 of LNCS. (1998)
3. Loogen, R., Ortega-Mallén, Y., Peña, R.: Parallel Functional Programming in Eden. Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming, to appear (2004)

4. Peyton Jones, S., et al.: The Glorious Glasgow Haskell Compilation System, Version 6.4 (2005) Available at: `http://www.haskell.org/ghc`.
5. Berthold, J., Klusik, U., Loogen, R., Priebe, S., Weskamp, N.: High-level Process Control in Eden. In Kosch, H., et al., eds.: EuroPar 2003 – Parallel Processing. Volume 2790 of LNCS., Klagenfurt, Austria (2003)
6. Berthold, J.: Towards a generalised runtime environment for parallel haskells. In Bubak, M., et al., eds.: Computational Science — ICCS'04. Volume 3038 of LNCS., Krakow, Poland, Springer-Verlag (2004) 297ff
7. Nordin, T., Peyton Jones, S.L.: Green card: a foreign-language interface for Haskell. In: Proceedings of the Haskell Workshop, Amsterdam, Netherlands (1997)
8. Reid, A.: Template Greencard. In: Proceedings of 15th International Workshop on the Implementation of Functional Languages (IFL 2003), Edinburgh (2003)
9. Sheard, T., Peyton Jones, S.: Template Meta-programming for Haskell. In: Haskell Workshop 2002, ACM Press (2002)
10. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science. Volume 2. (1992) 461–493
11. Jones, M.P.: Functional Programming with Overloading and Higher-Order Polymorphism. In: LNCS 925, 1st International School on Advanced Functional Programming, Båstad, Sweden. (1995)
12. Klusik, U., Loogen, R., Priebe, S.: Controlling Parallelism and Data Distribution in Eden. In: Trends in Functional Programming (Selected papers of the Second Scottish Functional Programming Workshop). Volume 2., Intellect (2000) 53–64
13. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. In Kluge, W., ed.: Workshop on the Implementation of Functional Languages, Bonn, Germany, Universität Kiel (1996)
14. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic Programming – An Introduction –. In Swierstra, S.D., Henriques, P.R., Oliveira, J.N., eds.: Advanced Functional Programming, LNCS 1608. Springer-Verlag (1999) 28–115
15. Lynagh, I.: Template Haskell: A report from the field. Unpublished. Available from the author's web page. (2003)
16. Lynagh, I.: Unrolling and simplifying expressions with Template Haskell. Unpublished. Available from the author's web page. (2003)
17. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell. In: Third International Conference on Generative Programming and Component Engineering, Springer-Verlag (2004) 186–205
18. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In: Haskell Workshop. (2001)
19. Tolmach, A., et al.: An External Representation for the GHC Core Language (2001) (Draft for GHC 5.02 documentation).
20. Veldhuizen, T., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98). (1998)
21. Norell, U., Jansson, P.: Prototyping Generic Programming in Template Haskell. In: Proc. of the 7th Int. Conf. on Mathematics of Program Construction. (2004)
22. Lämmel, R., Peyton Jones, S.: Scrap your Boilerplate: A Practical Design Pattern for Generic Programming. In: Proceedings of ACM Sigplan Types in Language Design and Implementation (TLDI). (2003)

Syntactic Abstraction in Component Interfaces

Ryan Culpepper[1], Scott Owens[2], and Matthew Flatt[2]

[1] Northeastern University
ryanc@ccs.neu.edu
[2] University of Utah
{sowens, mflatt}@cs.utah.edu

Abstract. In this paper, we show how to combine a component system and a macro system. A component system separates the definition of a program fragment from the statements that link it, enabling independent compilation of the fragment. A macro system, in contrast, relies on explicit links among fragments that import macros, since macro expansion must happen at compile time. Our combination places macro definitions inside component signatures, thereby permitting macro expansion at compile time, while still allowing independent compilation and linking for the run-time part of components.

1 Introduction

Good programmers factor large software projects into smaller components or modules. Each module addresses a specific concern, and a program consists of a network of cooperating modules. First-order module systems provide name management, encapsulation, and control over separate compilation [1]. However, first-order module systems use *internal linkage*, in which modules refer directly to other modules.

A module system can support component programming [2] by separating module definition from linking. Thus, components use *external linkage*, in which a component refers indirectly to other components through a parameterization mechanism. Additionally, a component must be compilable and deployable by itself, apart from any linkages that use the component. In analogy to separate compilation, we call this property *independent compilation*. A single independently compiled component is therefore re-usable in various situations, linked with a variety of other components. Although many module systems [3,4,5] support component-style parameterization, we concentrate here on a system designed expressly for component programming: *Units* [6].

Units and other component systems allow a component to import and export values and types, but not macros. Macro support is desirable because macros allow the definition of domain-specific language extensions, and components may benefit from these extensions. Because Scheme [7] supports sophisticated, lexically-scoped macros [8], implementors have devised module systems that support the import and export of macros [5,9,10,11,12], but these module systems do not support component-style parameterization with independent compilation.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 373–388, 2005.

This paper explains how to integrate macros and components while maintaining the desirable properties of both. In particular, our macros respect the lexical scope of the program and our components can be compiled before linking.

Section 2 introduces an example of a component-based program in PLT Scheme, our implementation substrate. Section 3 explains how the use of macros improves the program and introduces the macro system. Section 4 shows how we combine macros and components, and Section 5 shows more uses of macros in components. Section 6 discusses related work, and Section 7 concludes.

2 Programming with Units

Units are software components with explicit import and export interfaces. These interfaces serve as the canonical mechanism for designing systems and communicating documentation. Furthermore, since units are externally linked, they can be independently compiled.

Using the PLT Scheme unit system, programmers can organize programs as networks of components. Units are heavily used in the major applications distributed with PLT Scheme, including DrScheme and the PLT web server.

2.1 The Unit System

Signatures are the interfaces that connect units. A signature specifies a set of bindings that a unit may either import or export. A unit specifies one signature that lists its exported bindings, but it can specify many signatures listing imported bindings to support importing from multiple different units.

Signatures are defined using the **define-signature** form:

(**define-signature** signature-id
  (variable-id*))

The **unit/sig** expression specifies an atomic unit as follows:

(**unit/sig** (**import** signature-id*) (**export** signature-id)
  definition-or-expression$^+$)

The export signature indicates which definitions in the unit body are exported, and the import signatures indicate what variables are bound in the unit body. Unit expressions need not be closed. Like procedures, unit values close over the unit expression's free variables.

Units are *externally linked*; that is, a unit expression cannot refer specifically to the contents of another unit. Thus, compilation of a unit body does not require knowledge of any other unit, so units are independently compilable. Unit compilation depends only on signatures to determine import and export variables. These variables are compiled to use an indirection that supports linking.

Programs use a separate linking form called **compound-unit/sig** to link units together, satisfying each unit's import signatures:

```
(compound-unit/sig
  (import (tag : signature)*)
  (link (tag : signature (unit-expr tag*))+)
  (export (var tag : identifier)*))
```

The tags correspond to the nodes of the linkage graph, and the lists of tags in the link clauses specify the edges of the graph. The result is another unit whose imports are specified by the **import** clause and whose export signature is computed from the variables listed in the **export** clause.

The **invoke-unit/sig** form invokes a unit with no imports:

```
(invoke-unit/sig unit-expr)
```

An invocation evaluates all definitions and expressions in the unit's (atomic or compound) body in order.

2.2 An Example

Throughout the rest of this paper, we use the example of a hotel registration system that uses a database for persistent storage. The business logic consists of the following code:[1]

```
(define-signature db-sig (query make-select-cmd))
(define-signature hotel-reg-sig (get-reservation get-empty-rooms))

(define hotel-reg-unit
  (unit/sig (import db-sig) (export hotel-reg-sig)
    ;; get-reservation : string date → reservation
    (define (get-reservation name date)
      (———— (query (make-select-cmd 'reservations
                                    (list 'room 'rate 'duration)
                                    (list (cons 'name name) (cons 'date date)))
                   ————)))
    ;; get-empty-rooms : date → (list-of (cons string number))
    (define (get-empty-rooms date) ————)))
```

The third definition binds *hotel-reg-unit* to a unit satisfying the *hotel-reg-sig* interface. It must be linked with a database unit that exports the *db-sig* interface before it can be used.

The *hotel-reg-unit* component can be linked with any database component that exports the *db-sig* interface. In turn, the *hotel-reg-unit* unit provides the functionality represented by the *hotel-reg-sig* signature. This functionality may be used by any number of different components in the system, such as a graphical user interface for finding reservations and vacancies.

Assembling these components—the database code, the business logic, and the user-interface code—creates a complete program:

```
(define hotel-program-unit
  (compound-unit/sig
    (import)
```

[1] In the code fragments, we use ———— to indicate elided code.

```
    (link [HOTEL-DB : db-sig (PrestigeInc-db-unit)]
          [HOTEL-REG : hotel-reg-sig (hotel-reg-unit HOTEL-DB)]
          [GUI : hotel-gui-sig (hotel-gui-unit HOTEL-REG)])
    (export)))

(invoke-unit/sig hotel-program-unit)
```

The hotel's programmers can also write a web interface and assemble a second view for the registration software:

```
(define hotel-servlet-unit
  (compound-unit/sig
    (import (SERVER : servlet-import-sig)
    (link [HOTEL-DB : db-sig (GargantuWare-db-unit)]
          [HOTEL-REG : hotel-reg-sig (hotel-reg-unit HOTEL-DB)]
          [WEB-UI : servlet-sig (hotel-webui-unit SERVER HOTEL-DB)])
    (export))))
```

The web server would then link *hotel-servlet-unit* against a unit providing controlled access to the web server's functionality and invoke the resulting unit.

Signatures not only control the linking process; they also guide programmers in writing the components that implement and use the interfaces they represent. The brief signature definition of *db-sig*, however, is insufficient as a guide to implementing or using the the database unit. The true interface consists of not only the set of imported names, but also the types of those names, descriptions of their meanings, and advice on their use. Figure 1 shows an improved description of the *db-sig* interface.

The comments in this revised signature specify function headers and contain a usage example for *query*, a complex function. The example shows how to use *query* to select two fields, *f1* and *f2*, from the table named *tab1*. Only the rows where field *f3* is zero are processed. The argument function extracts the fields from *fieldset* by position and concatenates them. The call to *query* returns the accumulated list of concatenated strings.

The example exposes the awkwardness of the unadorned *query* function. The results come back in a list of 3-tuples (also represented with lists) that must be unpacked by position. This fragile relationship breaks when fields are reordered, inserted, or deleted. The programmer should be able to refer to field values by name rather than by position, and the database library should provide an abstraction that manages the connection between field names and their offsets in the result tuples. Ideally, the field names would be variables in the result-handling code. Creating a variable binding based on arguments is beyond the power of procedural abstraction, but it is a standard use of macros.

3 Programming with Macros

A declarative **select** form, specifically designed for expressing database queries, is more robust and less awkward to use than the *query* function. It can express the example query from Fig. 1 as follows:

```
(define-signature db-sig
  [;; query : select-cmd ((list-of string) → α) → (list-of α)
   ;; Applies the procedure to the each record returned, accumulating the final result.
   query

   ;; a constraint is (pair-of symbol {string|number|boolean})

   ;; make-select-cmd : (list-of symbol) symbol (list-of constraint) → select-cmd
   make-select-cmd

   ;; Example: to return a list of the concatenations of fields
   ;; 'f1' and 'f2' in table 'tab1' where 'f3' equals 0:
   ;;   (query (make-select-cmd (list 'f1 'f2) 'tab1 (list (cons 'f3 0)))
   ;;            (lambda (fieldset)
   ;;              (string-append (list-ref fieldset 0) ":" (list-ref fieldset 1))))
   ])
```

Lines with ";;" are comments.

Fig. 1. Database interface

```
;; Example: to return a list of the concatenations of fields
;; 'f1' and 'f2' in table 'tab1' where 'f3' equals 0:
(select [(f1 f2) tab1 with (f3 = 0)]
  (string-append f1 ":" f2))
```

The **select** form is implemented with a macro that compiles a **select** expression into a call to the *query* function. In Scheme, a **define-syntax** expression binds a compile-time function as a macro.

```
(define-syntax (macro-name stx)
  macro-body)
```

A macro takes in and produces annotated s-expressions called *syntax objects*. In a macro body, the **syntax-case** form matches a syntax object against patterns and binds *pattern variables*, and the #' operator creates a syntax object from a *template*. Each pattern variable used in a template refers to the portion of the input expression matched by that variable. The postfix ellipsis operator in a pattern matches the previous pattern zero or more times (similar to Kleene star); in a template it repeats the previous template for every match.

Figure 2 presents the **select** macro in the syntax-case [8] macro system. For the above **select** example, this macro generates code equivalent to the *query* example in Fig. 1.

The **select** macro uses *field*, *table*, *f1*, *v2*, and *body* as pattern variables, but **with** and = are matched as literals. The macro uses the name *field* both as a symbol (by putting it inside a **quote** form) passed to *make-select-cmd* and as a variable name. Because *body* is placed within that **lambda** expression, *field*

```
(define-syntax (select stx)
  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     ;; table, all field, and all f1 are identifiers
     ;; each v2 can be any expression
     ;; body is an expression
     #'(query (make-select-cmd (list (quote field) ...)
                               (quote table)
                               (list (cons (quote f1) v2) ...))
              (lambda (fieldset)
                (apply (lambda (field ...) body)
                       fieldset)))]))
```

Fig. 2. A simple **select** macro

is bound in *body*, and the evaluation of *body* is delayed until the procedure is applied.

The macro expander invokes the **select** macro when it encounters an expression of the form (**select** ————). The macro's parameter, here *stx*, is bound to the **select** expression. Macro expansion replaces the **select** expression with the result of the **select** macro and continues processing the program.

3.1 Respecting Lexical Scope

Unlike macros in LISP or C, Scheme's macros respect lexical scoping.[2] Variable references introduced in a macro's template are bound in the environment of the macro's definition (i.e. to the lexically apparent binding occurrence), instead of the environment of the macro's use. Hence, the macro's user can use local variables that coincide with those chosen by the macro implementer without altering the macro's behavior. In addition, templated identifiers that become binding occurrences never capture references received through the macro's argument. This property protects the meaning of the macro user's code from the macro definition's choice of temporary variables.

For example, consider the definition of the **select** macro in Fig. 2. The template contains a use of the *query* variable. Since the macro is defined at the top-level, the expanded code always refers to the *top-level* variable of that name, even if **select** is used in a context where *query* is shadowed by a local binding.

Scheme's macro system stores information about the lexical context of identifiers in the syntax objects that macros manipulate. The following superscript annotations are intended to illustrate the information that the syntax objects carry:

[2] Historically, a lexically-scoped macro system has also been called *hygienic* [13] and *referentially transparent* [14].

```
(define query^top ————)
(define-syntax (select^top stx)
  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     #'(query^top (make-select-cmd^top (list^top (quote^top field) ...) ————)
                  ————)]))
```

Thus, the following code

```
(let ([query^loc "What is the meaning of life?"])
  (select^top [(f1 f2) tab1 with (f3 = 0)]
    (string-append^top f1 ":" f2)))
```

expands into the following:

```
(let ([query^loc "What is the meaning of life?"])
  (query^top (make-select-cmd^top (list^top (quote^top f1) (quote^top f2)) ————)
             ————))
```

The macro system uses the lexical context information to ensure that the use of $query^{top}$ is not bound by $query^{loc}$, but refers to the top-level binding.

The **select** macro also relies on the other guarantee of lexically-scoped macro expansion. The macro expander uses another form of annotation on syntax objects to track identifiers introduced by macros. When those identifiers become binding occurrences, such as *fieldset* in the template of **select**, they bind only uses also generated by the same macro expansion. Consider this use of the **select** macro:

```
(let ([fieldset (lambda (new-value) (set! last-field-seen new-value))])
  (select [(f1 f2) tab1 with (f3 = 0)]
    (fieldset (+ f1 f2))))
```

This expands into the following:

```
(let ([fieldset (lambda (new-value) (set! last-field-seen new-value))])
  (query ————
         (lambda (fieldset_1)
           (apply (lambda (f1 f2) (fieldset (+ f1 f2)))
                  fieldset_1))))
```

Again, the binding of $fieldset_1$ does not capture the use of *fieldset*.

These two guarantees made by lexically-scoped macro systems are crucial to writing and using reliable macros. Without static knowledge of the binding structure of macros, reasoning about them becomes impossible.

However, consider the implications for programming with units. A macro such as **select** which uses the *query* function must be defined in a context where *query* is bound. Since *query* is a unit variable, it is only available to other units, through an import clause. Thus the macro definition must occur within the body of the importing unit:

```
(define hotel-reg-unit
  (unit/sig (import db-sig) (export hotel-reg-sig)
    (define-syntax select ————)
    ————))
```

Putting the macro definitions in the client code, however, defeats the component abstraction. The database component should provide everything necessary to write client code—both dynamic and static constructs—so that every application component can use these facilities.

One attempt at a solution might define **select** in the database component and export it to the client components. With this solution, the client component could not be compiled until it is linked with a particular database component, for two reasons. First, compilation of a unit depends on the definition of every syntactic extension used in its body. Second, the contents of a unit are not available to other units until link-time. Thus if the definition of a macro resides in a unit, then its clients cannot be compiled until link-time. Furthermore, the same client unit might be compiled differently for every linkage it participates in. Thus, this proposal violates the important property of component programming that a component be independently compilable.

4 Signatures and Static Information

Our solution integrates macros and components by including macro definitions in signatures. Since signatures specify the static properties of units, they are a natural place to put syntactic extensions. In this section we present the design, pragmatics, and implementation of the new unit system.

4.1 Extended Signatures

A signature contains the set of names that make up a unit's interface. In the extended system, it also contains macro definitions that can refer to the signature's names.

The extended syntax of signature definitions is:

```
(define-signature signature-id
  (variable-id*
   macro-definition*))
```

With this extension, it is possible to put the **select** macro in the *db-sig* signature, rather than manually copying it into every client unit:

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx) ————)])
```

The syntax for units remains the same. When the *db-sig* signature is used in an import clause, however, it also inserts the **select** macro into the unit:

```
(unit/sig (import db-sig) (export application-sig)
  (select [(room rate) rooms with (available = true)]
    (format "~a, available for just $~a" room rate)))
```

Macro expansion of the unit body produces the desired target code:

```
(unit/sig (import db-sig) (export application-sig)
  (query (make-select-cmd (list 'room 'rate)
                          'rooms
                          (list (cons 'available true)))
         (lambda (fieldset)
           (apply (lambda (room rate)
                    (format "~a, available for just $~a" room rate))
                  fieldset))))
```

The expansion of the macro does not depend on the particular version of the database unit linked at run-time. In fact, the above unit may be linked to many database units during the course of the same program execution.

4.2 Preserving Lexical Scope

As discussed in Sect. 3.1, respect for lexical scope is a critical property of Scheme's macro systems. It ensures that variable references behave in the natural way. In particular, variable references inserted by a macro refer to the variables in the macro definition's context, not those in the context of the macro's use.

The problem becomes more complex when we allow macros to occur inside of signatures. When a signature macro refers to a signature variable, there is no definite binding of the variable for the macro to refer to. Rather, when the signature is *instantiated*, that is, when it is used in a unit's import clause, the instance of the macro will refer to the instance of the imported variable.

This observation suggests a natural extension of the principle of lexical scoping for macros. In a signature, a free occurrence of a name in a #'-form should have the same meaning as it does in the context of the signature definition, unless the name is also a signature element. In the latter case, the name should refer to the variable linkage created when the signature is used in a unit **import** clause.

To illustrate this principle, consider the example from earlier:

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx)
     (syntax-case stx (with =)
       [(select [(field ...) table with (f1 = v2) ...] body)
        #'(query (make-select-cmd (list (quote field) ...) ————)
                 (lambda (fieldset) (apply ————)))]))])
```

In the template for **select**, *query* and *make-select-cmd* must denote the unit import variables. In contrast, *list* and *apply* must refer to the standard Scheme procedures, because those were their meanings where the macro was defined. Any other interpretation of macro definitions would violate the scoping principle of Scheme. It is the task of the implementation to make sure that these properties hold.

4.3 Implementation

In order to compile units, the compiler must be able to completely expand the definitions and expressions in a unit's body and identify all imported and exported variables. The information necessary to do this is statically bound to the signature names, and the compilation of a **unit/sig** form consists of fetching this static information, eliminating the signatures from the code, and compiling the resulting core unit form.

Signature definition. When the compiler encounters a **define-signature** form, it builds a catalog consisting of the variable names and macro definitions that the signature contains. The catalog contains syntax objects, which are closed in the syntactic environment in which the macro signature occurs. This ensures that when the macro definitions are imported into a unit, lexical scoping is preserved.

Finally, it statically binds the signature name to a signature structure containing the information above. Thus,

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx) ————)])
```

binds *db-sig* to the static information

```
(make-signature 'db-sig
                ;; Variables
                (list #'query #'make-select-cmd)
                ;; Macro names
                (list #'select)
                ;; Macro definitions
                (list #'(define-syntax (select stx) ————)))
```

When *db-sig* appears in the import clause of a **unit/sig** form, the compiler looks up the static binding of *db-sig* and uses the catalog to eliminate the use of the signature.

Signature elimination. The compiler translates **unit/sig** forms into an intermediate core unit form, replacing signatures with their contents. In the resulting code, imported and exported variables are explicitly enumerated, and all signature-carried macro definitions are inlined into the core unit body. This elimination process respects scoping, as outlined in Sect. 4.2.

Consider the following example:

```
(define-signature db-sig
  [query make-select-cmd (define-syntax select ————)])

(let ([query^loc ————])
  (unit/sig (import db-sig) (export application-sig) ————))
```

The **select** macro definition that occurs in the signature refers to a variable whose name appears in the **let** binding. The macro definition must be closed in

the environment of the signature definition so that those names are not captured by the **let** binding. By using syntax objects in the signature catalog, we retain the correct syntactic environment for the macro definition.

There are two problems to overcome in implementing **unit/sig** correctly. First, the environment of the macro definitions is not quite complete; it lacks bindings for the signature's variables. Second, the names in the signature, being closed in a different environment, will not bind the names in unit body.

To illustrate, we write $query^{sig}$ for the identifier closed in the signature's environment and $query^{unit}$ for the identifier occurring in the unit body. Given the **unit/sig** expression

(**unit/sig** (**import** *db-sig*) (**export** *application-sig*)
  ($query^{unit}$ ————)
  (**select**$^{\mathbf{unit}}$ [(*room rate*) *rooms* **with** (*available* = *true*)] (———— *rooms rate*)))

if the variable names and macro definitions from *db-sig* were copied into the unit body, their identifiers would still have *sig* superscripts.

(**unit/sig** (**import** ($query^{sig}$ $make\text{-}select\text{-}cmd^{sig}$)) (**export** *application-sig*)
  (**define-syntax** (**select**$^{\mathbf{sig}}$ *stx*) ———— #'$query^{sig}$ ————)
  ($query^{unit}$ ————)
  (**select**$^{\mathbf{unit}}$ [(*room rate*) *rooms* **with** (*available* = *true*)] (———— *rooms rate*)))

Note that $query^{sig}$ is bound in the resulting unit body, so the environment for the definition of **select**$^{\mathbf{sig}}$ is complete. Unfortunately, $query^{unit}$ and **select**$^{\mathbf{unit}}$ are still unbound in the unit body.

To solve this problem, the compiler must identify the unit's local identifier for each signature identifier. For example, given $query^{sig}$, the compiler must determine that the local name is $query^{unit}$. Each such pair of foreign and local identifiers must have the same meaning. The compiler extends the environment of the unit form with bindings that alias each local identifier to the corresponding imported identifier.

In summary, the compiler copies each macro definition from its imported signatures into the unit body, explicitly enumerates the imports and exports (using the foreign names), and creates aliases for the local names. This pass eliminates the signatures and produces an intermediate core unit form in which all imported and exported variables are explicit, and the body consists of macro definitions, definitions, and expressions.

Figure 3 shows the result of this translation, using **unit** as the syntax for core unit forms and a **define-alias** instruction to create aliases. The **define-alias** form can be implemented with the low-level primitives of the PLT Scheme macro system.

Core units. Once the signatures have been eliminated, all import and export variables are explicitly enumerated and the unit body contains all imported macros definitions.

Compiling a core unit involves expanding the body—a simple process once the macro definitions have been brought into the proper context—and compiling imported and exported variables as cells managed by the unit linker rather than primitive Scheme variables.

```
(make-signed-unit
  ;; Metadata (intentionally omitted)
  ————————
  ;; Core unit
  (unit (import query^sig make-select-cmd^sig)
        (export ————————)
    ;; Imported macro definitions
    (define-syntax (select^sig stx) ———— #'query^sig ————————)
    ;; Alias code
    (define-alias (select^unit = select^sig)
                  (query^unit = query^sig)
                  (make-select-cmd^unit = make-select-cmd^sig))
    ;; Code from unit/sig begins here
    (query^unit ————————)
    (select^unit [(room rate) rooms with (available = true)] (———— room rate))))
```

Fig. 3. Translation into core unit

5 Static Programming

Many constructs can be expressed with a macro whose output depends only on the macro's input. The **select** macro (Fig. 2) is an example of such a construct. However, the compilation of some constructs relies on information from other locations in the program. For example, the pattern matching construct for algebraic datatypes relies on the definition of the datatype being matched. PLT Scheme's macro system supports the implementation of these kinds of constructs by letting macros communicate using compile-time variables.

A database has a statically-known structure called its *schema* that determines what queries are valid. We could add a mechanism to *db-sig* that lets clients specify the schema. Then **select** would be able to check at compile time that every query will succeed when run.

The **database-table** form lets a programmer express information about the database's relevant tables, fields, and types, and make the information accessible to the **select** macro.

```
;; database-table's syntax: (database-table table-name (field-name field-type) ...)
(database-table rooms (room number) (rate number) (available boolean))
(database-table reservations (room number) (name text) (when date))
```

Using these table declarations, the **select** macro can ensure that the table in question is declared, and that it contains the given fields. It can also emit run-time checks for the field types.

The **database-table** and **select** macros communicate through a compile-time variable named **schema**, which contains a mutable box. Compile-time variables are introduced using the **define-for-syntax** form, and they exist during

```
(define-for-syntax schema (box null))

(define-syntax (database-table stx)
  (syntax-case stx ()
    [(database-table table (field-name field-type) ...)
     #'(begin-for-syntax
         (let ([table-record '(table (field-name field-type) ...)])
           (set-box! schema (cons table-record (unbox schema)))))]))
```

Fig. 4. Implementation of **database-table**

```
(define-syntax (select stx)
  ;; get-table-info : identifier → table-info
  ;; Given a table name, returns the list of fields and types associated with it,
  ;; or raises an exception if the table is not known.
  (define (get-table-info table-stx)
    (let ([table-entry (assoc (syntax-object→datum table-stx) (unbox schema))])
      (if table-entry
          (cdr table-entry)
          (raise-syntax-error 'select "unknown table" table-stx))))

  ;; check-field : identifier table-info → void
  ;; Checks that the given field name belongs to the table.
  (define (check-field field-stx table-info)
    (let ([field-record (assoc (syntax-object→datum field-stx) table-info)])
      (unless field-record
        (raise-syntax-error 'select "field not declared" field-stx))))

  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     (let ([table-info (get-table-info #'table)])
       (for-each (lambda (field-id) (check-field field-id table-info))
                 (syntax→list #'(field ...)))
       ;; The resulting code is the same as before (see Fig. 2)
       #'————)]))
```

Fig. 5. Implementation of **select** with static checking

macro expansion. See Fig. 4 for the implementation of **schema** and **database-table**. The **begin-for-syntax** form in the expansion of the **database-table** form indicates that the enclosed expression should execute during macro expansion. The **database-table** form is similar to examples from previous work [12].

The new **select** macro checks the table name and fields it receives against the information in the stored schema. The revised implementation shown in Fig. 5

produces the same code as the previous **select** macro, but it additionally uses helper procedures to check that the table and fields are declared. If they are not, the macro signals a compile time error.[3]

Consider the following unit that uses the new *db-sig* facilities:

```
(unit/sig (import db-sig) (export application-sig)
  (database-table rooms (room number) (rate number) (available boolean))
  (database-table reservations (room number) (name text) (when date))
  (select [(room rate) rooms with (available = true)] (cons room rate))
  (select [(room duration) reservations with] (———— duration ————)))
```

The first use of **select** is correct, but the second use triggers a compile-time error, notifying the programmer that *duration* is not a valid field in the *reservations* table.

The *db-sig* signature contains the definitions of the **schema** variable and **database-table** macro alongside the **select** macro. Thus, when a unit's import specifies the *db-sig* signature, **schema**'s and **database-table**'s definitions are copied into the unit's body. Consequently, each unit with a *db-sig* import receives its own **schema** box, keeping table definitions from mixing between different units.

The **select** and **database-table** macros present other opportunities for static extensions that we do not explore here. For example, the **database-table** form could produce code to dynamically verify the accuracy of the static schema description, and the **select** macro could add additional code to check whether constant field specifications have the correct type.

6 Related Work

Bawden [16] has proposed a system of lexically-scoped, "first-class" macros based on a type system. The "first-class" macros are statically resolvable, which preserves compilability, but the values for the bindings used in a macro's expansion can be passed into and returned from functions. A "first-class" macro is defined in a template that includes macro definitions and a listing of variables that the macro is parameterized over, similar to our signatures. His system uses types to statically track macro uses, whereas in our system signatures are statically attached to units, avoiding the need for a type system.

Krishnamurthi's *unit/lang* construct [17] allows programmers to specify the programming language of a component in addition to its external interface. A language contains new macro-like extensions (in fact, languages are more powerful than macros) and run-time primitives. A unit/lang component internally specifies which language it imports, similar to how our units specify their signatures. However, the internally specified language position does not coordinate with the externally linked component parameters, so lexically-scoped macros

[3] A programming environment such as DrScheme [15] can use the information provided to *raise-syntax-error* to highlight the source of the problem.

cannot refer to these parameters. Our system also makes it simpler to mix together orthogonal language extensions, since there is no need to manually define a language that contains the desired combination of extensions.

The Scheme community has formalized the principles of scope-respecting macros. Kohlbecker et al. [13] first introduced the hygiene property for macros, and subsequent papers developed referential transparency [8,14]. Recent papers have addressed macros and internally-linked modules [11], including a notion of phase separation for macros and modules [12]. Other work in macro semantics from MacroML [18,19] has not addressed the full complexity of Scheme macros. In particular, these systems do not model macros that can expand into macro definitions.

A different line of research has developed module systems to support component programming. Components are parameterized using collections of code that obey static interfaces, but the information carried by these interfaces is generally limited. In ML module systems [4,20,21], signatures contain types (and kinds, i.e., types for types) and datatype shapes (used for pattern matching). Similarly, the signatures in the original model of units [6] contain types and kinds. In the Jiazzi unit system [22], signatures contain class shapes, which play a type-like role.

The Scheme48 [5] module system provides some support for modules with parameterized imports. However, the signatures for the imports only contain the information that a binding is a macro, and the not macro itself. Consequently, parameterized modules that import macros cannot be independently compiled. We believe that our techniques could correct this problem.

7 Conclusion

We have designed and implemented an extension of the PLT Scheme unit system that allows programmers to attach language extensions to signatures, thus enriching the interfaces available to client code. Our extension preserves the essential properties of the unit system, such as independent compilation and external linking, as well as the lexical scoping principles of the macro system.

References

1. Wirth, N.: Programming in MODULA-2 (3rd corrected ed.). Springer-Verlag New York, Inc., New York, NY, USA (1985)
2. Szyperski, C.: Component Software. Addison-Wesley (1998)
3. MacQueen, D.: Modules for standard ml. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. (1984) 198–207
4. Leroy, X.: Manifest types, modules, and separate compilation. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1994) 109–122
5. Kelsey, R., Rees, J., Sperber, M.: Scheme48 Reference Manual. 1.1 edn. (2005) `http://s48.org/1.1/manual/s48manual.html`.

6. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (1998) 236–248
7. Kelsey, R., Clinger, W., Rees (Editors), J.: Revised^5 report of the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
8. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation **5** (1993) 295–326
9. Queinnec, C.: Modules in scheme. In: Proceedings of the Third Workshop on Scheme and Functional Programming. (2002) 89–95
10. Serrano, M.: Bigloo: A "practical Scheme compiler". 2.7a edn. (2005) `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.
11. Waddell, O., Dybvig, R.K.: Extending the scope of syntactic abstraction. In: Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, New York, NY (1999) 203–213
12. Flatt, M.: Composable and compilable macros: You want it *when?* In: ACM SIGPLAN International Conference on Functional Programming. (2002)
13. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., Duba, B.F.: Hygienic macro expansion. In: ACM Symposium on Lisp and Functional Programming. (1986) 151–161
14. Clinger, W., Rees, J.: Macros that work. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1991) 155–162
15. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12** (2002) 159–182 A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.
16. Bawden, A.: First-class macros have types. In: Proc. symposium on Principles of programming languages, ACM Press (2000) 133–141
17. Krishnamurthi, S.: Linguistic Reuse. PhD thesis, Rice University (2001)
18. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: International Conference on Functional Programming. (2001) 74–85
19. Taha, W., Johann, P.: Staged notational definitions. In: Proceedings of the second international conference on Generative programming and component engineering. (2003) 97–116
20. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1994) 123–137
21. Harper, R., Pierce, B.C.: Design issues in advanced module systems. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2004) To appear.
22. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned Java. In: Proc. conference on Object oriented programming, systems, languages, and applications, ACM Press (2001) 211–222

Component-Oriented Programming with Sharing: Containment is Not Ownership

Daniel Hirschkoff[1], Tom Hirschowitz[1], Damien Pous[1],
Alan Schmitt[2], and Jean-Bernard Stefani[2]

[1] LIP ENS Lyon, 46, Allée d'Italie 69364 Lyon Cedex 07 - France
[2] INRIA Rhône-Alpes, 655 Avenue de l'Europe, 38334 St Ismier, France

Abstract. Component-oriented programming yields a tension between higher-order features (deployment, reconfiguration, passivation), encapsulation, and component sharing. We propose a discipline for component-oriented programming to address this issue, and we define a process calculus whose operational semantics embodies this programming discipline. We present several examples that illustrate how the calculus supports component sharing, while allowing strong encapsulation and higher-order primitives.

1 Introduction

Wide-area distributed systems and their applications are increasingly built as heterogeneous, dynamic assemblages of software components. This modular structure persists during execution: such systems provide the means to control their run-time modular configuration, which encompasses automatic deployment, unanticipated evolution, passivation, run-time reconfiguration, and introspection. This expressive power conflicts with the strong encapsulation properties generally expected from modular programs.

A key tension point is *component sharing*, which allows two remote components to encapsulate a common component, as depicted in Figure 1, where the component L (e.g. a software library) is shared among C and D. How does one preserve encapsulation in this case? In particular, what happens to L and D if A removes C from the configuration? How can C replace L by L', without necessarily impacting D? Essentially, the difficulty lies in combining (1) encapsulation with fine-grain, objective control over communications, (2) locality passivation, migration, and replication, and (3) access to shared components with simple communication rules.

Previous models of component-oriented programming do not completely address these three requirements. Models that do not address requirement (3) comprise process calculi with hierarchical localities that feature local communications only (i.e., no direct communication between arbitrarily distant localities in the locality forest) [5, 4, 14, 3, 8, 18]. Indeed, sharing is representable in such models, but at the expense of complex routing rules which are difficult to maintain. Models that do not have this routing complexity problem, but are weak on requirement (1), include the Cell calculus [15] and process calculi with localities that do not restrict communications betwen localities [10, 12, 20, 19, 11, 1, 16]. The tKlaim calculus [9] is a recent variant of Klaim that allows the establishment of different communication topologies between localities.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 389–404, 2005.

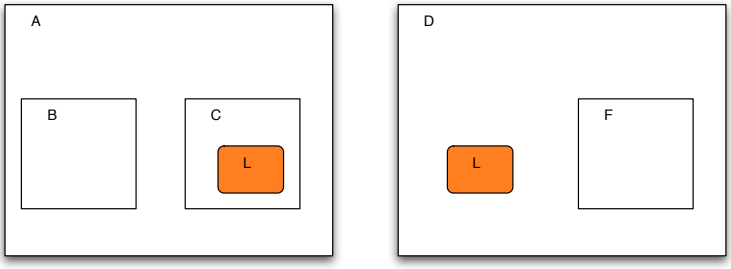


Fig. 1. A configuration with sharing

However, such calculus still falls short of full encapsulation of sub localities, since there is no objective control over process migration and execution.

Our starting point to solve the issue of component sharing is that, from the standpoint of the latter kind of models (weak on (1)), the problem is reminiscent of the *aliasing problem* in object-oriented languages [13]: sharing is easy, but encapsulation is problematic. To solve this problem, Clarke et al. introduce *ownership types* [6, 7], which attribute to each object o an *owner* that controls the references to o. We adapt this idea of ownership to the setting of process calculi. However, instead of designing a type system to preserve encapsulation, we enforce it at the level of the operational semantics, as follows. We split the usual hierarchical forest of localities into two graphs: the *ownership forest* and the *containment graph*. Locality passivation must be local for the ownership forest, and communication must be local for the containment graph. As in type systems for ownership, we require, by scoping constraints in our semantics, that owners be dominators: the owner of a component c dominates (in the ownership forest) all the components holding references to c. Owing to this condition, the aliasing problem does not arise: when updating a component c, its owner has access to all references to c. Moreover, the containment graph may be an arbitrary directed graph, which allows component sharing. The resulting language, an extension of the Kell calculus [18], turns out to be an interesting model of component-oriented programming, as we show by encoding key aspects of the Fractal component model [2].

Our main contributions are as follows: (1) we propose a programming discipline for component-oriented programming to address the issue of component sharing, while preserving encapsulation and higher-order features; (2) we define a process calculus whose operational semantics embodies this programming discipline; (3) we argue that our calculus is suitable to represent most idioms of component-oriented programming, by reviewing key concepts from a concrete component model; (4) additionally, we propose a new, more modular, presentation of the Kell calculus.

The paper is organized as follows. §2 briefly presents the Fractal model, discusses the modeling of Fractal components in the Kell calculus, and introduces informally several examples of component sharing. §3 extends the Kell calculus with primitive component sharing. §4 shows how to program several component sharing examples within the obtained calculus. §5 concludes the paper with a discussion of future research.

2 Components and the Kell Calculus

After giving an informal description of the Kell calculus [18], which is our starting point, we present the main elements of a concrete component model, the Fractal model [2]. We discuss to which extent Fractal component configurations without sharing can be interpreted as processes of the Kell calculus. We then present various examples of component configurations with sharing, and explain informally how we extend the Kell calculus with sharing to deal with these examples.

2.1 The Kell Calculus

The Kell calculus is a higher-order process calculus with hierarchical localities (called *kells*), local communication, and locality passivation. Actions in the Kell calculus are communication actions and passivation actions. Communication is said to be *local* as it may occur only within a kell, between a kell and its sub kells, or between a kell and its immediate parent, as illustrated below.

1. Receipt of local message $a\langle Q\rangle.T$ on port a bearing process Q and continuation T by local trigger (input construct) $a\langle x\rangle \triangleright P$.

$$a\langle Q\rangle.T \mid (a\langle x\rangle \triangleright P) \rightarrow T \mid P\{Q/x\}$$

2. Receipt of message $a\langle Q\rangle.T$ residing in sub kell b by local trigger $a^{\downarrow}\langle x\rangle \triangleright P$.

$$b[a\langle Q\rangle.T].S \mid (a^{\downarrow}\langle x\rangle \triangleright P) \rightarrow b[T].S \mid P\{Q/x\}$$

   In pattern $a^{\downarrow}\langle x\rangle$, the arrow $\downarrow$ denotes a message that should come from a sub kell.
3. Receipt of message $a\langle Q\rangle.T$ residing out of the enclosing kell by local trigger $a^{\uparrow}\langle x\rangle \triangleright P$.

$$a\langle Q\rangle.T \mid b[a^{\uparrow}\langle x\rangle \triangleright P].S \rightarrow T \mid b[P\{Q/x\}].S$$

   In input pattern $a^{\uparrow}\langle x\rangle$, the arrow $\uparrow$ denotes a message that should come from the outside of the immediately enclosing kell.

These constructs may be combined using *join patterns* [10] that are triggered only when the required messages are simultaneously present, as in the following example (note that $\mid$ has higher precedence than $\triangleright$).

$$a\langle Q\rangle.T \mid b[c\langle R\rangle.U \mid (a^{\uparrow}\langle x\rangle \mid c\langle y\rangle \triangleright P)].S \rightarrow T \mid b[U \mid P\{Q/x, R/y\}].S$$

Communication with other localities has to be explicitly programmed in the language. For instance, in order to exchange messages, two sibling kells need the help of their common parent, as depicted in the following example.

$$\begin{array}{rl} & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) \quad \mid b[c\langle P\rangle.Q] \mid e[(c^{\uparrow}\langle x\rangle \triangleright T)]] \\ \rightarrow & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) \mid c\langle P\rangle \mid b[Q] \quad \mid e[(c^{\uparrow}\langle x\rangle \triangleright T)]] \\ \rightarrow & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) \quad \mid b[Q] \quad \mid e[T\{P/x\}] \quad] \end{array}$$

In this example, the parent locality contains a permanent forwarder $c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle$ that pulls messages of the shape $c\langle P\rangle$ out of its sub kells. This allows sub kells to receive these messages using an *up pattern* $c^{\uparrow}\langle x\rangle$. The construction $(\xi \diamond P)$ denotes a replicated trigger, i.e., a trigger which persists after a reaction, and is in fact a shorthand for $\nu t.Y_{t,\xi,P} \mid t\langle Y_{t,\xi,P}\rangle$, where $Y_{t,\xi,P} = (t\langle y\rangle \mid \xi \triangleright P \mid y \mid t\langle y\rangle)$.

Passivation in the Kell calculus is depicted in the following example, where the kell named a is destroyed, and the process Q it contains is used in the guarded process P.

$$a[Q].T \mid (a[x] \triangleright P) \rightarrow T \mid P\{Q/x\}$$

Assume, for instance, that we want to model the dynamic update of component b, where the new version P of the component program is received on channel a. We could do so, in one atomic action, using the following join pattern where the new version $b[P]$ is spawned, replacing the previous b component.

$$a\langle P\rangle \mid (a\langle x\rangle \mid b[y] \triangleright b[x]) \mid b[Q] \;\rightarrow\; b[P]$$

2.2 The Fractal Component Model and Its Interpretation in the Kell Calculus

Fractal is a general component model which is intended to implement, deploy, monitor, and dynamically configure complex software systems, including in particular operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), introspection capabilities (to monitor and control the execution of a running system), and re-configuration capabilities (to deploy and dynamically configure a system).

A Fractal component is a run-time entity which is encapsulated, which has a distinct identity, and which is either primitive or composite (built from other components). Bindings between components are described explicitly, either by local, *primitive* bindings, using explicit component interfaces, or by remote, *composite* bindings, using components whose role is to embody communication paths. Features like encapsulation and interfaces are rather standard. The originality of the Fractal model lies in its reflective features and in its ability to define component configurations with sharing. In order to allow for well scoped dynamic reconfiguration, components in Fractal can be endowed with controllers, which provide a meta-level access to a component internals, allowing for component introspection and the control of component behaviour. A Fractal component consists of two parts: *contents*, that correspond to its internal components, and a *membrane*, which provides so-called *control interfaces* to introspect and reconfigure the internal features of the component. The membrane of a component is typically composed of several controllers.

Representing a Fractal component (without sharing) in the Kell calculus is relatively straightforward. A component named a, takes the form $a[P \mid Q]$, where process P corresponds to the membrane of the component, and process Q, of the form $c_1[Q_1] \mid \ldots \mid c_n[Q_n]$, corresponds to the contents of the component, with n sub components c_1 to c_n. Interfaces of a component can be interpreted as channels on which a component can emit or receive messages. The membrane P is composed of controllers implementing the control interfaces of the component.

The Fractal model specifies several useful forms of controllers, which can be combined and extended to yield components with different reflective features. Let us briefly describe some of them, and sketch their interpretation in the Kell calculus.

An *attribute* of a component is a configurable property that can be manipulated by the means of an *attribute controller*. It can be interpreted as some value held in a memory cell by a component membrane. A membrane providing an attribute controller interface is easy to program, by emitting the current value of the attribute on a private channel and by providing channels to read and update this value.

$$\nu s.(\mathtt{get}^{\uparrow}\langle r\rangle \mid s\langle v\rangle \diamond s\langle v\rangle \mid r\langle v\rangle) \mid (\mathtt{set}^{\uparrow}\langle v'\rangle \mid s\langle v\rangle \diamond s\langle v'\rangle) \mid s\langle 0\rangle$$

A *contents controller* supports an interface to list, add, and remove sub components in the contents of a component. A membrane providing a simplistic contents controller interface could be of the form $\mathtt{Add} \mid \mathtt{Rmv} \mid \ldots$, with the following definitions (in which the contents controller interface is manifested by the `cc` channel carrying the request type (where $\backslash add$ means a name that is exactly add), the name c of the targetted component, and either the program of the added component (including both membrane and contents) or a channel r to return the removed component).

$$\mathtt{Add} = (\mathtt{cc}^{\uparrow}\langle \backslash add, c, x\rangle \diamond c[x]) \qquad \mathtt{Rmv} = (\mathtt{cc}^{\uparrow}\langle \backslash rmv, c, r\rangle \diamond (c[x] \triangleright r\langle c, x\rangle))$$

A less simplistic encoding would take into account additional details, such as exception conditions (e.g, the possible absence of a component to remove). However, the above definitions convey the essence of the contents controller.

A *life-cycle controller* allows an explicit control over the execution of a component. As an illustration, we can define a membrane P providing a simple interface to suspend and resume execution of sub components (where the life-cycle interface is manifested by the `lfc` channel, and a sub component c is supended by turning it into a message on a channel of the same name as the component).

$$P = \mathtt{Suspend} \mid \mathtt{Resume} \mid \ldots \qquad \mathtt{Suspend} = (\mathtt{lfc}^{\uparrow}\langle \backslash suspend, c\rangle \diamond (c[x] \triangleright c\langle x\rangle))$$
$$\mathtt{Resume} = (\mathtt{lfc}^{\uparrow}\langle \backslash resume, c\rangle \diamond (c\langle x\rangle \triangleright c[x]))$$

Again, a more realistic implementation would be more complex, but this section only aims to show that capturing the operational essence of a reflective component model such as Fractal (without component sharing) is relatively direct using the Kell calculus. For another example, Schmitt and Stefani [17] provide an interpretation of a *binding controller*, allowing a component to bind and unbind its client interfaces to server interfaces.

2.3 Component Sharing

Component sharing arises in situations where some resource must be accessed by several client components. A first example of such a situation is that of a log service, which merely provides client components the ability to register status information. Figure 1 depicts an example configuration, where L is the log service component, and C and D are client components. In this case, communications are unidirectional, from the client

$$\prod_{i\in I;j\in J_i} m_{ij}\left[(msg\langle k,l,x\rangle \diamond n_i\langle k,l,x\rangle) \mid (n_i^{\uparrow}\langle \backslash i,\backslash j,x\rangle \diamond x) \mid P_{ij}\right]$$

$$\mid \quad R\left[\prod_{i,j\in I;i\neq j} (n_i^{\uparrow}\langle \backslash j,l,x\rangle \diamond n_j\langle j,l,x\rangle)\right] \quad \mid \quad \prod_{i\in I} (n_i^{\downarrow}\langle k,l,x\rangle \diamond n_i\langle k,l,x\rangle)$$

Fig. 2. A router configuration

components to the shared component, and the log service maintains its own mutable state. Passivation of a client does not affect the execution of the log service or the processing of logging requests previously sent by that client.

Figure 1 can illustrate as well a second example of component sharing, that of a shared programming library or module. In this case, the communication between client components C and D and the library L is bidirectional (typically, a request/response style of communication). The expected behavior in presence of passivation is different from the first example: if a client is passivated, requests to functions in the library should be suspended along with the rest of the client activity.

As a third example, consider a database service used by several components of a system (for instance, a directory service), which can again be depicted as in Figure 1. Here, the communications between clients and the service are bidirectional, but they are no longer independent as in the previous example, for the database service maintains a mutable state that can be viewed and updated by each client component.

The previous examples correspond to pure software architectures and describe configurations on a single machine. One can also consider mixed software/hardware configurations. For instance, consider the case of a router R connecting several networks N_i with $i \in I$. Each network N_i connects machines m_{ij} with $j \in J_i$. There are several ways to model such a configuration in the Kell calculus without sharing. If one wants to model the networks as components and have messages be directly exchanged between machines and the networks, and between the networks and the router, then the locality of communications and the tree structure of kells impose the following shape:

$$R\left[\prod_{i\in I} N_i\left[\prod_{j\in J_i} m_{ij}\,[\ldots]\right]\right]$$

where $\prod_{i\in I} P_i$ means the parallel composition of the processes P_i.

Such an approach is not satisfactory because the passivation of the router or of a network, e.g., to model their failure, implies the passivation of several machines. A solution consists of modelling a network N_i by a channel n_i, as in Fig.2. Machines send outbound messages on the channel msg with the destination machine address k, l, where k is the destination network, and the message to deliver x. Each machine m_{ij} contains a rule that forwards such messages to the local network n_i. Each network N_i is represented by the replicated pulling of messages on n_i out of sub kells. The

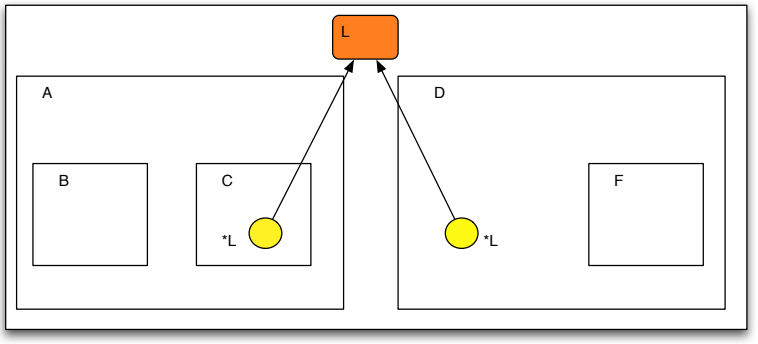


Fig. 3. A Kell calculus configuration with sharing

router pulls messages that are in a network different from their destination—$n_i\langle k, l, P\rangle$ with $i \neq k$—and routes them to the correct network. Finally, every machine m_{ij} picks from the local network the messages that target it using the pattern $n_i^{\uparrow}\langle\backslash i, \backslash j, x\rangle$. This encoding, however, does not model the fact that the networks are disjoint resources shared by the machines and connected by a router.

In this paper, we extend the Kell calculus with explicit sharing, following the ideas sketched in §1. Technically, in our extension, the ownership forest is captured by the locality hierarchy. For instance, in configuration $C = a[b[P] \mid c[R \mid d[Q]]]$, component a is the owner of components b and c, while c is the owner of component d. The containment graph is captured via *references* to shared components: thus the process $*a$ denotes a reference to the component named a. For instance, in the configuration $D = a[H \mid b[P \mid *e] \mid c[R \mid *e] \mid e[Q]]$, component a is the owner of component e, which is shared by components b and c as each of them holds a reference $*e$. The scope of a component e, where it is accessible by references $*e$, is the sub tree rooted at the owner of e, unless there is a deeper component named e whose scope encompasses the reference. Note that the scope does not include e itself. In our extension, a reference $*e$ can be created and communicated, exactly as a name. In the latter case, note that references may escape their original scope: for instance, in the configuration D above, if H passivates component b, and sends it outside of a, then the reference to the shared component e will escape its scope. Allowing a component reference to escape its scope makes it possible to model in a simple way a primitive form of dynamic binding for shared components. The example of Fig.1 may then be represented in the Kell calculus with sharing as in Fig.3.

Passivation in the new calculus takes place just as in the Kell calculus without sharing. However, communications across kell boundaries now require a reference to that kell to receive a message from it or to send a message to it.

$$b[a\langle Q\rangle.T].S \mid (a^{\downarrow}\langle x\rangle \triangleright P) \mid *b \rightarrow b[T].S \mid P\{Q/x\} \mid *b$$
$$a\langle Q\rangle.T \mid *b \mid b[a^{\uparrow}\langle x\rangle \triangleright P].S \rightarrow T \mid *b \mid b[P\{Q/x\}].S$$

Process:	$P ::= 0 \mid x \mid P\|Q \mid \nu a.P \mid a\langle\widetilde{P}\rangle.Q \mid a[P].Q \mid *a \mid \tilde{\xi} \triangleright P$
Pattern:	$\xi ::= a[x] \mid a^{\alpha}\langle\widetilde{\eta}\rangle \mid *a$
Argument pattern:	$\eta ::= x \mid a \mid \backslash a \mid a \mathrel{!}= b$
Place pattern:	$\alpha ::= \bullet \mid \uparrow \mid \downarrow$
Formula:	$F ::= \epsilon \mid r \mid r^{\perp} \mid F\|F$
Resource:	$r ::= \widetilde{M} \mid a^{\downarrow}(\widetilde{M}) \mid a^{\uparrow}(\widetilde{M}) \mid a[P] \mid *a \mid a \mid s$
Spot:	$s ::= \triangleright \mid a[\triangleright] \mid [\triangleright]$
Message:	$M ::= a\langle\widetilde{P}\rangle$

Fig. 4. Processes and formulas

3 The Calculus

The syntax of the calculus is depicted in Figure 4. It is based on a denumerable set of *variables* x and a denumerable set of *names* a. *Processes* include the standard null process 0, variables x, parallel composition $P|Q$, and name creation $\nu a.P$, plus some less standard constructs. Messages have the shape $a\langle\widetilde{P}\rangle.Q$, where $\widetilde{P}$ is a list of processes (we use $\widetilde{\cdot}$ in the following to denote a list of $\cdot$'s). In $a\langle\widetilde{P}\rangle.Q$, Q is called a *continuation*, because it is triggered synchronously upon consumption of the message. *Kells* have the shape $a[P].Q$, where a is the name of the kell, P is its contents, and, as for messages, Q is its continuation. The calculus admits references $*a$ as processes, for referencing remote kells named a, as informally described in §2.3. References are also used to send names in messages, as illustrated in matching rules M-NAME, M-CST, and M-NEG below. Finally, the calculus features first-class reduction rules, called *triggers*, which are written $\tilde{\xi} \triangleright P$. Here, $\tilde{\xi}$ denotes a list of *patterns*, where each variable and name is bound at most once (see the definition of scoping below). A pattern ξ may be a kell pattern $a[x]$ for passivation of active kells, a reference $*a$, for suppression of containment links, or a message pattern $a^{\alpha}\langle\widetilde{\eta}\rangle$, for plain communication. In the message pattern, $\widetilde{\eta}$ denotes a list of argument patterns of the shape $x, a, \backslash a$, or $a \mathrel{!}= b$. The first two kinds of argument patterns respectively represent input of processes and names. The third kind $\backslash a$ tests the equality of the corresponding message argument with a. The last kind $a \mathrel{!}= b$ checks that the argument is different from b, and inputs it as a. The direction α indicates where the received message should come from: $\uparrow$ messages come from a parent kell, $\bullet$ messages come from the current kell, and $\downarrow$ messages come from a sub kell.

Processes are scoped as follows. Name restriction is a binder, as usual. Moreover, given a trigger $\tilde{\xi} \triangleright P$, the *defined identifiers* $\mathsf{DI}(\tilde{\xi})$ of $\tilde{\xi}$ bind in P. We define $\mathsf{DI}(\tilde{\xi})$ as follows. Given an argument pattern η, define $\mathsf{DI}(x) \stackrel{\Delta}{=} \{x\}$, $\mathsf{DI}(a) \stackrel{\Delta}{=} \mathsf{DI}(a \mathrel{!}= b) \stackrel{\Delta}{=} \{a\}$, and $\mathsf{DI}(\backslash a) \stackrel{\Delta}{=} \emptyset$. Then, let $\mathsf{DI}(\tilde{\xi})$ be the disjoint union of all $\mathsf{DI}(\xi)$, for ξ in $\tilde{\xi}$, with $\mathsf{DI}(a[x]) = \{x\}$, $\mathsf{DI}(*a) = \emptyset$, and $\mathsf{DI}(a^{\alpha}\langle\widetilde{\eta}\rangle)$ the disjoint union of all $\mathsf{DI}(\eta)$, for η in $\widetilde{\eta}$. Let structural congruence $\equiv$ be the smallest congruence including, as usual,

associativity and commutativity of parallel composition, neutrality of 0 w.r.t. $|$, extrusion of name restriction above $|$, ν, and $a[\cdot].P$, and renaming of bound variables and names.

Resources. The reduction relation is based on a labelled transition system (LTS), whose labels represent a trade of *resources* r. As discussed below, such a trade is typically written $F_1 \twoheadrightarrow F_2$, where F_1 and F_2 are *formulas*, to express that the process undergoing the transition offers the resources described by F_2, provided the environment provides the resources in F_1. In particular, the reacting trigger $\tilde{\xi} \triangleright P$ trades some basic resources (messages, passivated kells) against a *reaction token* written $\triangleright$: if the environment provides the expected resources, then the trigger reacts. When composing processes, the corresponding transitions are composed, which may involve the annihilation of some resource requests and corresponding offers, in case they meet.

As defined in Figure 4, there are two kinds of resources. *Basic resources* include messages ($\widetilde{M} \mid a^{\downarrow}(\widetilde{M}) \mid a^{\uparrow}(\widetilde{M})$), where $M ::= a\langle\widetilde{P}\rangle$, passivated kells ($a[P]$), consumed references ($*a$), and permissions ($a$). They are generated directly from processes. For example, a message $a\langle P\rangle.Q$ trades a reaction token against $a\langle P\rangle$, yielding the transition $a\langle P\rangle.Q \xrightarrow{\triangleright \twoheadrightarrow a\langle P\rangle} Q$. As explained in §2.1, we want to control the locality of communications, so this transition should happen in the same kell as the transition involving the reacting trigger, and trades involving $\triangleright$ should only take place at the same level as the reaction.

On the other hand, we cannot completely restrict trades to the level of the reaction, e.g., because the consumed resources may come from shared kells, which syntactically may reside far above the reaction site. This leads us to consider several kinds of reaction tokens, each of them determining the position of the considered transition relatively to the reacting trigger. These reaction tokens are called *spots* $s \in \mathsf{Spots}$.

More precisely, consider a process $S|b[a[(\xi \triangleright P)|Q]|R]$, where the reacting trigger is $\xi \triangleright P$. We have just seen that resources matching the reaction token $\triangleright$ provided by $\xi \triangleright P$ may only come from Q. Immediately above a, i.e., in R, trades may use the information that the reaction lies in some sub kell named a. Thus, in R, $\triangleright$ is viewed as the *sub reaction* token $a[\triangleright]$. Further above a, e.g., from S, it becomes the less precise *internal reaction* token $[\triangleright]$, which only indicates that the reaction lies in some sub kell.

Formulas. Formulas are the labels of our LTS. Intuitively, they match the resources offered and requested by the considered process. Formally, *formulas* are defined as in Figure 4, and considered equivalent modulo the following equation schemes:

$$F_1|F_2 = F_2|F_1 \quad (1) \qquad F|\epsilon = \epsilon|F = F \quad (2) \qquad \frac{r \notin \mathsf{Spots}}{r|r^{\perp} = \epsilon} \quad (3) \qquad s|s^{\perp} = s \quad (4)\,.$$

Equation (3) specifies that basic resources (non-spots) are used linearly: they may be consumed only once; (4) specifies that one spot may satisfy several requests, as a join pattern consumes several messages.

Transitions. The LTS is defined in Figure 5. Rule MATCH describes reaction, using the notion of *matching* defined below, which is a three arguments judgement written $\xi : F \rightarrow \Theta$, where Θ is a *substitution*. A substitution is an element of $(\mathsf{Vars} \rightarrow_{\mathsf{fin}} \mathsf{Processes}) \times (\mathsf{Names} \rightarrow_{\mathsf{fin}} \mathsf{Names})$, i.e., a pair of a finite map from variables to processes and a finite map from names to names. Capture-avoiding substitution is defined

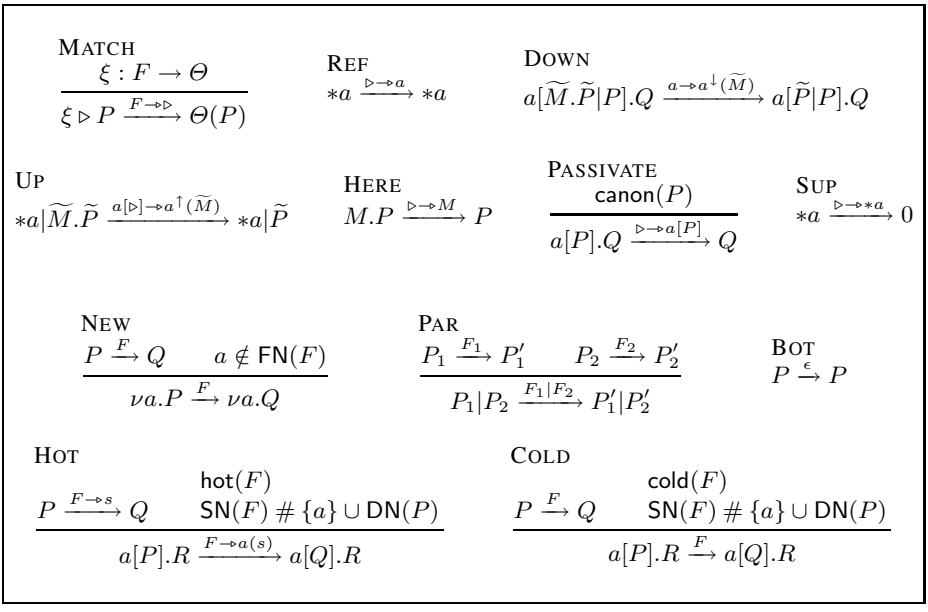


Fig. 5. The LTS

as usual on processes, and written $\Theta(P)$. Define the negation $F^{\perp}$ of a formula F by distributing it over resources, given that $r^{\perp\perp} = r$. Let $F_1 \multimap F_2$ denote $F_1^{\perp}|F_2$. The rule states that if $\xi : F \to \Theta$, then the trigger $\xi \triangleright P$ has a transition to $\Theta(P)$, under the label $F \multimap \triangleright$. Thus, the reaction happens only if the environment provides the resources F (recall that spot $\triangleright$ stands here for the firing of the trigger).

By rule REF, at the level of a reaction, a reference may generate a permission to receive messages from the kell it points to. This permission is then used in rule DOWN to actually consume the corresponding messages. By rule UP, a reference to the reacting kell allows the reaction to consume messages from the kell holding the reference. By rules HERE, PASSIVATE, and SUP, a reaction may consume messages, active kells, and references at its top-level. In rule PASSIVATE, we use the notation $\mathsf{canon}(P)$ to mean that P has no active ν. This means that such ν's must have been extruded before by structural congruence. Formally, a *context* $\mathbb{C}$ is a process with exactly one occurrence of the special variable $\Box$. Textual replacement of $\Box$ with some process P (possibly with capture) is written $\mathbb{C}\{P\}$. A process P is in *canonical form*, written $\mathsf{canon}(P)$, iff for all context $\mathbb{C} \neq \Box$, if $P = \mathbb{C}\{\nu a.Q\}$, then $\mathbb{C}\{\nu a.Q\} \not\equiv \nu a.\mathbb{C}\{Q\}$.

The other rules specify how the transition relation is closed under active contexts. Rule NEW handles the case of ν. Rule PAR combines the resources of several parts of the process. If one argument provides the resources requested by the other, then the trade occurs. Formally, two derivations having an occurrence of the MATCH rule can be put together using this rule: the restriction to only one active trigger per reaction is enforced by the rule for reduction, presented below. Rule BOT closes transitions under parallel composition with spectator processes.

Rule HOT allows to wrap an already existing reaction inside some parent kell: a transition $P \xrightarrow{F \twoheadrightarrow s} Q$ is seen from the enclosing kell as $a[P].R \xrightarrow{F \twoheadrightarrow a(s)} a[Q].R$, where the operation $a(s)$ over spots is defined by $a(\triangleright) \stackrel{\Delta}{=} a[\triangleright]$, and $a(b[\triangleright]) \stackrel{\Delta}{=} a([\triangleright]) \stackrel{\Delta}{=} [\triangleright]$. The rule is subject to two side conditions. First, F must be *hot*, written $\mathsf{hot}(F)$, which means that F matches the syntax $F ::= \epsilon \mid b^{\perp} \mid b^{\downarrow}(\widetilde{M}) \mid b^{\uparrow}(\widetilde{M}) \mid F|F$. Second one must have $\mathsf{SN}(F) \# \mathsf{DN}(P)$ (see below). Intuitively, the presence of s in the label of the conclusion imposes that the reaction occurs in P, so the side condition means that a reacting kell only has three kinds of interactions with its context: 1) it (partially) specifies the place of reaction; 2) it exhibits authorizations to access shared kells; 3) it consumes messages through references to shared kells (in both directions). The second side condition enforces the fact that references point to the closest kell in the hierarchy, as informally stated in §2.3. We call the *defined names* $\mathsf{DN}(P)$ of a process P the set of all a's such that $P \equiv \nu\widetilde{b}.Q|a[R]$. for some $Q, R, \widetilde{b}$, with $a \notin \widetilde{b}$. Moreover, a formula is in *canonical form* iff, for each resource r, it does not contain both r and $r^{\perp}$. We define the *scoped names* $\mathsf{SN}(F)$ of a formula F in canonical form as follows: for resources r of the shape $a^{\downarrow}(\widetilde{M}), a^{\uparrow}(\widetilde{M}), a$, and $a[\triangleright]$, let $\mathsf{SN}(r) \stackrel{\Delta}{=} \{a\}$; for other resources r, let $\mathsf{SN}(r) \stackrel{\Delta}{=} \emptyset$. Additionally, let $\mathsf{SN}(F_1|F_2) \stackrel{\Delta}{=} \mathsf{SN}(F_1) \cup \mathsf{SN}(F_2)$ and $\mathsf{SN}(F^{\perp}) \stackrel{\Delta}{=} \mathsf{SN}(F)$. The rule prevents resources consumed through a reference $*a$ to escape the scope of any kell named a. For instance, a request for a message of the shape $a^{\downarrow}(b\langle P\rangle)$ through a reference $*a$ is supposed to be consumed in (one of) the closest kell(s) named a. Such a request leads to the formula $a^{\perp}|a^{\downarrow}(b\langle P\rangle) \twoheadrightarrow s$: if a down message is found in a, using formula $a \twoheadrightarrow a^{\downarrow}(b\langle P\rangle)$, then the reaction occurs. However, if $P_1 \xrightarrow{a^{\perp}|a^{\downarrow}(b\langle P\rangle) \twoheadrightarrow s} P_2$, then we do not want $c[P_1|a[Q_1]] \xrightarrow{a^{\perp}|a^{\downarrow}(b\langle P\rangle) \twoheadrightarrow s} c[P_2|a[Q_1]]$ to hold, because the message ought to be found in Q_1. Here, $\mathsf{DN}(a[Q_1]) = \{a\}$ which is not disjoint from $\mathsf{SN}(a^{\perp}|a^{\downarrow}(b\langle P\rangle)) = \{a\}$. Note that this check is done only when crossing kell boundaries. Indeed, we allow the presence of more than one kell named a in parallel to the reacting trigger.

Symmetrically to rule HOT, rule COLD allows to transfer resources from kells containing references $*a$ to the reacting kell a, which may be syntactically distant. Let F be cold, written $\mathsf{cold}(F)$, iff F matches the syntax $F ::= b[\triangleright]^{\perp} \mid b^{\uparrow}(\widetilde{M}) \mid F|F$. Rule COLD says that any transition with a cold label is viewed identically from outside the ambient kell, provided the scoping conditions are met. In practice, rule COLD is only used to transfer the consumption of up messages (created by rule UP) through kells.

Matching. Figure 6 defines the matching relation. Rule M-PAR states that matching a pattern $\xi_1|\xi_2$ is like matching ξ_1 and ξ_2 separately, and then combining the result. In the rule, $+$ denotes the union of finite maps with disjoint domains. By rule M-HERE, matching a pattern $a^{\bullet}\langle\widetilde{\eta}\rangle$ against a resource $a\langle\widetilde{P}\rangle$ boils down to match $\widetilde{\eta}$ against $\widetilde{P}$ (as defined below). Rule M-ELSEWHERE handles the cases of down and up messages. Given a pair ζ consisting of a name a and argument patterns $\widetilde{\eta}$, we let ζ^{α} stand for $a^{\alpha}\langle\widetilde{\eta}\rangle$. Similarly, given a list $\widetilde{\zeta} = \zeta_1|\ldots|\zeta_n$, let $\widetilde{\zeta}^{\alpha} = \zeta_1^{\alpha}|\ldots|\zeta_n^{\alpha}$. The rule tunes the directions (up or down) in order to allow rule M-HERE to apply coherently. Rules M-PASSIVATE and M-SUP are straightforward. For message contents, Rule M-CST states that an escaped pattern $\backslash a$ matches itself, yielding no substitution. Rules M-NAME, M-NEG, and

$$\text{M-PAR}\ \frac{\xi_1 : F_1 \to \Theta_1 \qquad \xi_2 : F_2 \to \Theta_2}{\xi_1|\xi_2 : F_1|F_2 \to \Theta_1 + \Theta_2} \qquad \text{M-HERE}\ \frac{\widetilde{\eta} : \widetilde{P} \to \Theta}{a^\bullet\langle\widetilde{\eta}\rangle : a\langle\widetilde{P}\rangle \to \Theta} \qquad \text{M-ELSEWHERE}\ \frac{\widetilde{\zeta}^\bullet : \widetilde{M} \to \Theta}{\widetilde{\zeta}^\alpha : a^\alpha(\widetilde{M}) \to \Theta}$$

$$\text{M-PASSIVATE}\ a[x] : a[P] \to \{x \mapsto P\} \qquad \text{M-SUP}\ *a : *a \to \emptyset \qquad \text{M-PROC}\ x : P \to \{x \mapsto P\} \qquad \text{M-NAME}\ a : *b \to \{a \mapsto b\}$$

$$\text{M-CST}\ \backslash a : *a \to \emptyset \qquad \text{M-NEG}\ \frac{b \neq c}{a\ !{=}\ b : *c \to \{a \mapsto c\}} \qquad \text{M-NIL}\ \epsilon : \epsilon \to \emptyset \qquad \text{M-CONS}\ \frac{\eta : P \to \Theta_1 \qquad \widetilde{\eta} : \widetilde{P} \to \Theta_2}{\eta, \widetilde{\eta} : P, \widetilde{P} \to \Theta_1 + \Theta_2}$$

Fig. 6. Matching

M-PROC handle the input of names and variables. Rules M-NIL and M-CONS dispatch the results.

Reduction. Finally, reduction, written $\to$, is the smallest relation satisfying the rule

$$\frac{P \equiv P' \qquad P' \xrightarrow{s} Q' \qquad Q' \equiv Q}{P \to Q}\ .$$

As exactly one spot is allowed, this rule guarantees that exactly one trigger fires.

4 Examples

Let us first present a simple example.

Example 1. *Consider the following configuration.*

$$A = a[(e_1^\uparrow\langle x\rangle \mid e_2^\uparrow\langle y\rangle \triangleright P) \mid c\langle Q\rangle] \quad | \quad l_1[e_1\langle U\rangle \mid *a] \quad | \quad l_2[e_2\langle V\rangle \mid *a \mid (c^\downarrow\langle z\rangle \triangleright R)]$$

The component a can emit the message $c\langle Q\rangle$, which implies that a reference $*a$ to a can be used to access this message. Hence we have the following reduction where the rule in kell l_2 is triggered.

$$A \to a[(e_1^\uparrow\langle x\rangle \mid e_2^\uparrow\langle y\rangle \triangleright P)] \quad | \quad l_1[e_1\langle U\rangle \mid *a] \quad | \quad l_2[e_2\langle V\rangle \mid *a \mid R\{Q/z\}]$$

The component a can also receive messages from both components l_1 and l_2 since it is a shared sub component of both. Hence we have the following reduction where the rule in kell a is triggered.

$$A \to a[P\{U, V/x, y\} \mid c\langle Q\rangle] \quad | \quad l_1[*a] \quad | \quad l_2[*a \mid (c^\downarrow\langle z\rangle \triangleright R)]$$

Let us now give an example of dynamic binding and reconfiguration in the calculus.

Example 2. *Consider the following configuration, which models a running component receiving instructions to update its sub component c with a new code $P(d)$, which uses a service named d.*

$$A = update\langle c, P(d)\rangle \mid *a \mid a[(update^{\uparrow}\langle b, x\rangle \diamond (b[y] \triangleright b[x])) \mid c[P_c] \mid d[P_d]]$$

It reduces in two steps to $*a \mid a[(update^{\uparrow}\langle b, x\rangle \diamond (b[y] \triangleright b[x])) \mid c[P(d)] \mid d[P_d]]$, *where the references to d in $P(d)$ have been dynamically bound to $d[P_d]$.*

We now review the examples of §2.3 within our calculus. First, assume given two components $Queue[\ldots]$ and $Pair[\ldots]$, working as follows. They expect messages from their parent components, on channels $Queue.push$, $Queue.pop$, $Pair.fst$, and so on. The channels of these messages identify the action to execute. The messages contain a return channel name and the corresponding arguments. On the return channel, $Queue$ and $Pair$ send messages which have to be picked up as down messages by the client parent component. For convenience, we use the syntactic sugar $\mathsf{let}\ x = a(\widetilde{P})\ \mathsf{in}\ Q$ for $\nu b.a\langle b, \widetilde{P}\rangle|(b^{\downarrow}\langle x\rangle \triangleright Q)$, with some fresh b used as return channel. For instance, $\mathsf{let}\ x = Queue.push(P, Q)\ \mathsf{in}\ R$ uses the result x of pushing P on top of Q in R.

Example 3. *The log service example can be represented as follows (reproducing the configuration of Figure 1 with $L = Log$).*

$$\begin{array}{l} Log[*Queue \mid \ldots \text{code to actually log} \ldots \\ \qquad \mid (Log.log^{\uparrow}\langle x\rangle \mid state\langle y\rangle \diamond \mathsf{let}\ z = Queue.push(x, y)\ \mathsf{in}\ state\langle z\rangle)] \\ \mid \quad A[B[\ldots] \mid C[*Log \mid \ldots]] \quad \mid \quad D[*Log \mid F[\ldots]] \end{array}$$

In the rest of the program, the encapsulation links to Log are represented by occurrences of the reference $*Log$. The ownership of Log by, say, o is encoded by the fact that the sub component Log appears at the top-level in o. The implicit scope of Log, restricted to processes encapsulated in o, ensures that o is a dominator of Log.

Example 4. *The shared printer example can be represented as follows, where c stands for "client", and j stands for "job".*

$$\begin{array}{l} Printer[*Queue \mid *Pair \mid \ldots \text{code to actually print} \ldots \\ \qquad \mid (Printer.lpr^{\uparrow}(c, j) \mid state\langle q\rangle \diamond \mathsf{let}\ x = Pair.pair(c, j)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ q' = Queue.push(x, q)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad state\langle q'\rangle) \\ \qquad \mid (Printer.lpq^{\uparrow}(r) \mid state\langle q\rangle \diamond r\langle q\rangle \mid state\langle q\rangle)] \\ \mid \quad A[B[\ldots] \mid C[*Printer \mid \ldots]] \quad \mid \quad D[*Printer \mid F[\ldots]] \end{array}$$

The shared library example can be represented similarly. We can however emphasize the code server aspect of the example with a representation that only requires a unidirectional communication between the clients and the shared library. The shared library is thus modelled as a code server that allows an instance of the library code to be made available on request in the client component that requires it.

Example 5. *The shared library example can be represented as follows, where* $!a\langle P\rangle$ *stands for* $\nu b.(a\langle P\rangle.b\langle\rangle|(b\langle\rangle \diamond a\langle P\rangle.b\langle\rangle))$.

$$\prod_{i\in I;j\in J_i} m_{ij}\left[*N_i \mid (n_i^{\downarrow}\langle \backslash i, \backslash j, x\rangle \diamond x) \mid P_{ij}\right] \mid R\left[\prod_{i,j\in I;i\neq j} (n_i^{\uparrow}\langle \backslash j, l, x\rangle \diamond n_j\langle j, l, x\rangle)\right]$$
$$\mid \prod_{i\in I} N_i\left[*R \mid (n_i^{\downarrow}\langle k, l, x\rangle \diamond n_i\langle k, l, x\rangle) \mid (msg^{\uparrow}\langle k, l, x\rangle \diamond n_i\langle k, l, x\rangle)\right]$$

Fig. 7. A better router configuration

$$Lib[!Lib.get\langle P\rangle] \quad | \quad A[B[\ldots] \mid C[*Lib \mid \ldots]] \quad | \quad D[*Lib \mid F[\ldots]]$$

Finally, we review the router example from Figure 2, which is more direct than Examples 3 and 4 because it does not require any data structure: we just assume that names include integers.

Example 6. *The router example is depicted in Fig.7. It is very similar to Fig.2: the router is identical and shared between the networks, the networks are now kells shared between machines and may directly pull messages out of machines and the router. This encoding allows the failure of the router or a network to only impact inter-machine communication, it also segregates messages in different networks.*

5 Conclusion

Component sharing, as experienced with component models providing it, is a feature that proves extremely useful when describing or programming software architectures or systems with shared resources. We have presented in this paper an extension of the Kell calculus that provides a direct, formal interpretation of component models with sharing. To our knowledge, this is the first calculus offering (1) encapsulation with fine-grain, objective control over communications, (2) locality passivation, migration and replication, and (3) access to shared components with simple communication rules. Our approach draws on a distinction between ownership and containment inspired by recent works on ownership types and the control of aliasing in object-oriented programming languages. In contrast to these works, however, our approach avoids the burden of a type system, by primitively distinguishing ownership from containment, thus enforcing the programming discipline directly in the operational semantics.

The work we have presented here is only preliminary, however. First, the standard issues appearing when one introduces a new process calculus remain to be dealt with, e.g., the development of a bisimulation-based behavioral theory, or of static analyses to ensure semantic properties of processes. Furthermore, it would be interesting to study the exact relation between approaches to object containment and ownership in object-oriented languages and in the Kell calculus with sharing. At a minimum, we need to investigate the different benchmarks used in the object-oriented programming community and study how they are handled in our calculus.

Second, two important, inter-related questions remain, that pertain (1) to the control of communications with shared components, and (2) to the control over dynamic binding. The first issue concerns a potential security hole in our design. It can be succinctly stated as follows: in the extended Kell calculus presented here, the construct $\nu a.a[a[P]]$ is not a perfect firewall, while it is in the plain Kell calculus. This is due to the fact that P may have references to shared kells, which may in turn allow P to emit and receive messages from its environment. We see two posssible solutions to this problem.

First, one could annotate each kell construct $a[\cdot]$ with explicit sieves on communications with shared components. For instance, let us write $a[P]_A$, where $A ::= \emptyset \mid * \mid \widetilde{a} \mid \neg\widetilde{a}$ represents the names of shared components the present component is allowed to communicate with. Then, define the interpretation of annotations by $[\![\emptyset]\!] = \mathsf{Names}$, $[\![*]\!] = \emptyset$, $[\![\widetilde{a}]\!] = \mathsf{Names} \setminus \widetilde{a}$, and $[\![\neg\widetilde{a}]\!] = \widetilde{a}$. The semantics of these consructs is given by a simple modification of the rules HOT and COLD, given by adding textually the side condition $\mathsf{SN}(F) \# [\![A]\!]$ to both of them. With these new constructs and rules, we recover the perfect firewall equation for $\nu a.a[a[P]_\emptyset]_\emptyset$: P cannot communicate with the environment outside of a.

The second, more radical solution is to introduce a second ν operator, say ∇, that would not cross component boundaries. Channel names bound by ∇ would then represent communication channels, while free names and names bound by ν would represent global names. Distant communication would be restricted to channels, thus preventing an incoming piece of code to arbitrarily communicate with distant components. Global names would serve for matching against local messages. We conjecture that the presence of ν and ∇ avoids the need for directional patterns $(\uparrow, \downarrow, \bullet)$. The calculus thus collapses to a simpler version. The second solution might also turn out to solve the second problem (which is not the case of the first solution): the distinction between local channels and global names might give rise to a fine-grain account of dynamic binding, provided the pattern language is enriched adequately.

References

[1] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D.Gorla, M.Loreti, E.Moggi, R.Pugliese, E.Tuosto, B.Venneri. The KLAIM project: Theory and practice. In *GC*, vol. 2874 of *LNCS*. Springer, 2003.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani. An open component model and its support in Java. In *CBSE*, vol. 3054 of *LNCS*. Springer, 2004.

[3] M. Bugliesi, G. Castagna, S. Crafa. Boxed ambients. In *TACS*, vol. 2215 of *LNCS*. Springer, 2001.

[4] L. Cardelli, A. D. Gordon. Mobile ambients. In *FOSSACS*, vol. 1378 of *LNCS*. Springer, 1998.

[5] G. Castagna, F. Zappa Nardelli. The Seal calculus revisited: Contextual equivalence and bisimilarity. In *FSTTCS*, vol. 2556 of *LNCS*. Springer, 2002.

[6] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.

[7] D. Clarke, T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, vol. 2743 of *LNCS*. Springer, 2003.

[8] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *Computing: the Australasian Theory Symposium*, vol. 78 of *ENTCS*. Elsevier, 2003.
[9] R. De Nicola, D. Gorla, R. Pugliese. Global computing in a dynamic network of tuple spaces. In *COORD*, vol. 3454 of *LNCS*. Springer, 2005.
[10] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *POPL*. ACM Press, 1996.
[11] M. Hennessy, J. Rathke, N. Yoshida. SafeDpi: a language for controlling mobile code. In *FOSSACS*, vol. 2987 of *LNCS*. Springer, 2004.
[12] M. Hennessy, J. Riely. Resource access control in systems of mobile agents. In *International Workshop on High-Level Concurrent Languages*, vol. 16(3) of *ENTCS*. Elsevier, 1998.
[13] J. Hogg, D. Lea, A. Wills, D. deChampeaux, R. Holt. The Geneva convention on the treatment of object aliasing, 1991.
[14] F. Levi, D. Sangiorgi. Controlling interference in ambients. In *POPL*. ACM Press, 2000.
[15] Y. D. Liu, S. F. Smith. Modules with interfaces for dynamic linking and communication. In *ECOOP*, vol. 3086 of *LNCS*. Springer, 2004.
[16] A. Ravara, A. Matos, V. Vasconcelos, L. Lopes. Lexically scoped distribution: what you see is what you get. In *FGC*, vol. 85(1) of *ENTCS*. Elsevier, 2003.
[17] A. Schmitt, J.-B. Stefani. The Kell calculus: A family of higher-order distributed process calculi. In *GC*, vol. 3267 of *LNCS*. Springer, 2005.
[18] J.-B. Stefani. A calculus of Kells. In *FGC*, vol. 85(1) of *ENTCS*. Elsevier, 2003.
[19] P. T. Wojciechowski, P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *Concurrency*, 8(2), 2000.
[20] N. Yoshida, M. Hennessy. Assigning types to processes. In *LICS*. IEEE, 2000.

Language Requirements for Large-Scale Generic Libraries

Jeremy Siek and Andrew Lumsdaine

Open Systems Laboratory,
Indiana University
{jsiek, lums}@osl.iu.edu

Abstract. The past decade of experience has demonstrated that the generic programming methodology is highly effective for the design, implementation, and use of large-scale software libraries. The fundamental principle of generic programming is the realization of interfaces for entire sets of components, based on their essential syntactic and semantic requirements, rather than for any particular components. Many programming languages have features for describing interfaces between software components, but none completely support the approach used in generic programming. We have recently developed $\mathcal{G}$, a language designed to provide first-class language support for generic programming and large-scale libraries. In this paper, we present an overview of $\mathcal{G}$ and analyze the interdependence between language features and library design in light of a complete implementation of the Standard Template Library using $\mathcal{G}$. In addition, we discuss important issues related to modularity and encapsulation in large-scale libraries and how language support for validation of components in isolation can prevent many common problems in component integration.

1 Introduction

In the 1980s Musser and Stepanov developed a methodology for creating highly reusable algorithm libraries [1, 2, 3, 4], using the term "generic programming" for their work.[1] Their approach was novel in that their algorithms were not written based on any particular data structure. Rather, the algorithms were written based on requirements that a structure would have to meet for the algorithm to be correct. Such generic algorithms could therefore operate on any data structure provided the structure met the specified requirements. For example, a given generic algorithm could operate on linked lists, arrays, red-black trees (representing ordered sequences), and even structures developed independently of the generic library. Early versions of their generic algorithm libraries were implemented in Scheme, Ada, and C.

In the early 1990s Stepanov and Musser took advantage of the template system in C++ [5] to construct the Standard Template Library (STL) [6, 7]. The STL became part of the C++ Standard, which brought their style of generic programming into the mainstream.

[1] The term "generic programming" is often used to mean any use of "generics", i.e., any use of parametric polymorphism or templates. The term is also used in the functional programming community for function generation based on algebraic datatypes, i.e., "polytypic programming". Here, we use "generic programming" solely in the sense of Musser and Stepanov.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 405–421, 2005.

Since then, the methodology has been successfully applied to the creation of libraries for numerous domains [8, 9, 10, 11, 12].

The ease with which programmers implement and use generic libraries varies greatly depending on the language features available for expressing polymorphism and requirements on type parameters. In [13] we performed a comparative study of modern language support for generic programming, implementing a representative subset of the Boost Graph Library [9] in each of six languages. While some languages performed quite well, none were ideal for generic programming.

C++ was the most flexible of the languages studied, enabling straightforward expression of generic algorithms. C++ is flexible because it does not independently type check or compile template definitions but instead delays checking and compilation until instantiation. As a result, C++ templates are able to perform type-specific operations. For example, the following template applies + to an object of type T.

```
template<class T> T inc(T x) { return 1 + x; }
```

When inc is instantiated with int, the + resolves to integer addition.

```
int main() { inc<int>(2); }
```

When inc is instantiated with double, the + resolves to floating point addition.

```
int main() { inc<double>(3.14159); }
```

However, the flexibility gained by delaying type checking and compilation has a price. Templates are very difficult to validate in isolation for there is no support from the type checker. Even worse, users of template library experience infamously bad error messages when they make a mistake and trigger errors from deep inside the template.

At the other end of the spectrum are languages with parametric polymorphism[2] such as ML. Parametric polymorphism allows for separate type checking and compilation but does not allow type-specific operations to be applied to an object whose type is a type variable. For example, translating the above inc template to ML results in a type error. In the code below, 'a is a type variable corresponding to T and x : 'a declares a parameter with type 'a.

```
fun inc (x : 'a) = x + 1
// error: Can't unify Time.time/word/real/int with 'a
// (Cannot unify with explicit type variable) Found near +( x, 1)
```

This restriction can be side-stepped with higher-order functions, where the type-specific operations are parameters of the polymorphic function.

```
fun inc (x : 'a) (one : 'a) (add : 'a * 'a -> 'a) = add(x, one)
```

However, typical generic algorithms require dozens of type-specific operations and adding dozens of parameters would make them difficult to use.

There is tension between the need for type-specific operations in generic algorithms and the need for separate type checking and compilation. The language presented here, named $\mathcal{G}$, resolves this tension by combining parametric polymorphism with a rich interface specification language that is tailored to generic programming. In [14] we laid the foundation for $\mathcal{G}$, defining a core calculus, named F^G, based on System F [15, 16]. With F^G we captured the essential features for generic programming in a small formal

[2] C++ templates are often incorrectly categorized as a form of parametric polymorphism.

system and proved type safety. The language $\mathcal{G}$ applies the ideas from F^G to a full programming language capable of implementing the entire STL.

1.1 Contributions

The contributions of this paper are the design and evaluation of a language for generic programming:

- We give a high-level and intuitive description of the language $\mathcal{G}$. A formal description of the idealized core of $\mathcal{G}$, named F^G, is presented in [14]. We leave a formal description of the full language $\mathcal{G}$ for future work.
- We evaluate the design of $\mathcal{G}$ with respect to implementing the Standard Template Library. The STL is a large generic library that exercises all aspects of the generic programming methodology. The STL is therefore a fitting first test for validating the design of $\mathcal{G}$.
- We evaluate the design of $\mathcal{G}$ with respect to scalability issues in software development. In particular, we show how $\mathcal{G}$ provides support for the independent validation of components and support for component integration.

Many elements of $\mathcal{G}$ can be found in other programming languages, but $\mathcal{G}$ is unique in providing a carefully selected combination of language features for generic programming. In terms of interface description, the closest relative to $\mathcal{G}$ is Haskell's type classes. However, $\mathcal{G}$ differs in that 1) the concept feature in $\mathcal{G}$ integrates nested types and type sharing (similar to ML), 2) model definitions in $\mathcal{G}$ obey normal scoping rules, and 3) $\mathcal{G}$ explores design issues of type classes for non-type-inferencing languages.

1.2 Road Map

In Section 2 we review the essential ideas and terminology of generic programming and in Section 3 we present an overview of the language $\mathcal{G}$. We review the high-level structure of the Standard Template Library in Section 4 and in Section 5 we report on using $\mathcal{G}$ to implement the STL. In Section 6 we show how component development and integration is facilitated by the $\mathcal{G}$ type system. Related work is discussed in Section 7 and we conclude the paper in Section 8.

2 Generic Programming

The defining characteristics of the generic programming methodology are:

- Algorithms are expressed with minimal assumptions about data abstractions, and vice versa, making them maximally interoperable. This is accomplished by taking a concrete algorithm and lifting the non-essential requirements. For example, an algorithm on linked-lists becomes an algorithm on forward iterators.
- Absolute efficiency is required. Algorithms are never lifted to the point where they lose efficiency. When a single generic algorithm can not achieve the best efficiency for all input types, multiple generic algorithms are implemented and automatic algorithm selection is provided.

```
template<typename Iter, typename T>
// where Iter models Forward Iterator
//       T is the same type as iterator_traits<Iter>::value_type
//       T models Less Than Comparable
Iter lower_bound(Iter first, Iter last, const T& val) {
  typename iterator_traits<Iter>::difference_type
      len = distance(first, last), half;
  Iter middle;
  while (len > 0) {
    half = len >> 1; middle = first;
    advance(middle, half);
    if (*middle < val) { first = middle; ++first; len = len-half-1; }
    else len = half;
  }
  return first;
}
```

Fig. 1. Example of a generic algorithm in C++

The `lower_bound` template in Fig. 1 is a simple example of a generic algorithm. The algorithm is lifted from working on a particular data structure and is instead written in terms of the Forward Iterator concept. A ***concept*** is a set of requirements on a type. A type that meets the requirements is said to ***model*** the concept. The interface of a generic algorithm is specified by requiring that its type parameters model certain concepts. Concepts are not expressible in C++, so type requirements are specified in comments, as in Fig. 1. Built-in pointer types, such as `int*`, and the iterator type for the `std::list` class, are models of Forward Iterator.

A concept may incorporate the requirements of another concept, in which case the first concept is said to ***refine*** the second. The Forward Iterator concept refines the Input Iterator concept, adding the ability to pass through the sequence multiple times. The Input Iterator concept is a refinement of Assignable, Copy Constructible, and Equality Comparable. In addition, for a type `X` to model Input Iterator it must satisfy the following requirements:

- Given any objects `a` and `b` of type `X`, the expressions `++a`, `*a`, `a==b`, and `a!=b` must be valid (these operations must be defined).
- The type `X` must specify two helper types: a value type, which is the return type of the `operator*`, and a difference type, which must model Signed Integral and be suitable for measuring distances between iterators. In general, we refer to such helper types as ***associated types***. The type `X` must provide access to these types through `iterator_traits`.
- Given objects `a` and `b` of type `X`, `a == b` implies `*a` is equivalent to `*b`.
- The increment and dereference operators must be constant time.

Associated types change from model to model. For example, the associated value type for `int*` is `int` and the associated value type for `list<char>::iterator` is `char`.

The grouping of type requirements into concepts enables significant reuse: the Input Iterator concept is directly used as a type requirement in over 28 of the STL algorithms. The Forward Iterator is used in the specification of over 22 STL algorithms.

3 Overview of $\mathcal{G}$

$\mathcal{G}$ is a statically typed imperative language with syntax and memory model similar to C++. We have implemented a compiler that translates $\mathcal{G}$ to C++, but $\mathcal{G}$ could also be interpreted or compiled to byte-code. Compilation units are separately type checked and may be separately compiled, relying only on forward declarations from other compilation units (even compilation units containing generic functions and classes). The languages features of $\mathcal{G}$ that support generic programming are the following:

- Concept and model definitions;
- Constrained polymorphic functions, classes, structs, and type-safe unions;
- Implicit instantiation of polymorphic functions; and
- Concept-based function overloading.

In addition, $\mathcal{G}$ includes the usual types and control constructs of a general purpose programming language.

Concepts are defined using the following syntax:

```
cid<tyid,...> { cmem ... }; cmem ←
funsig | fundef        // Required operations
       | type tyid;            // Associated types
       | type == type;         // Same-type constraints
       | refines cid<type, ...>;
       | require cid<type, ...>;
```

The grammar variable *cid* is for concept names and *tyid* is for type variables. The type variables are place holders for the modeling type (or a list of types for multi-type concepts). *funsig* and *fundef* are function signatures and definitions. In a concept, a function signature says that a model must define a function with the specified signature. A function definition in a concept provides a default implementation. Concepts may be composed with `refines` and `require`. The distinction is that refinement brings in the associated types from the "super" concept. The following is the definition of the `InputIterator` concept in $\mathcal{G}$.

```
concept InputIterator<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>; //this includes Assignable and CopyConstructible
  require SignedIntegral<difference>;
  fun operator*(X b) -> value@;
  fun operator++(X! c) -> X!;
};
```

The modeling relation between a type and a concept is established with a model definition using the following syntax.

```
decl ← model [<tyid,...>] [where { constraint, ... }] cid<type,...> { decl ...};
```

A model may be parameterized with type variables in the <>'s and the `where` clause introduces constraints on the type variables:

constraint ← *cid*<*type*, ...> | *type* == *type*

The following statement establishes that all pointer types are models of `InputIterator`.

```
model <T> InputIterator<T*> {
  type value = T;
  type difference = ptrdiff_t;
};
```

A model definition must satisfy all requirements of the concept. Requirements for associated types are satisfied by type definitions. Requirements for operations may be satisfied by function definitions in the model, by the `where` clause, or by functions in the lexical scope preceding the model definition. Refinements and nested requirements are satisfied by preceding model definitions.

The syntax for generic functions is shown below. The name of the function is the identifier after `fun`, the type parameters are between the <>'s and are constrained by the requirement in the `where` clause. A function's parameters are between the `()`'s and the return type of a function comes after the `->`.

fundef ← `fun` *id* [<*tyid*, ...>] [`where {` *constraint*, ... `}`]
(*type pass* [*id*], ...) `->` *type pass* `{` *stmt* ... `}`
funsig ← `fun` *id* [<*tyid*, ...>] [`where {` *constraint*, ... `}`]
(*type pass* [*id*], ...) `->` *type pass*;
decl ← *fundef* | *funsig*
pass ← *mut ref* `// pass by reference`
| `@` `// pass by value`
mut ← `const` | ϵ `// constant`
| `!` `// mutable`
ref ← `&` | ϵ

The default parameter passing mode in $\mathcal{G}$ is read-only pass-by-reference. Read-write pass-by-reference is indicated by `!` and pass-by-value by `@`.

The body of a polymorphic function is type checked separately from any instantiation of the function. The type parameters are treated as abstract types so no type-specific operations may be applied to them unless otherwise specified by the `where` clause. The `where` clause introduces surrogate model definitions and function signatures (for all the required concept operations) into the scope of the function. The `distance` function is a simple example of a generic function in $\mathcal{G}$.

```
fun distance<Iter> where { InputIterator<Iter> }
(Iter@ first, Iter last) -> InputIterator<Iter>.difference@ {
  let n = zero();
  while (first != last) { ++first; ++n; }
  return n;
}
```

The dot notation used in the return type refers to an associated type, in this case the `difference` type of the iterator.

assoc ← *cid*<*type*, ...>.*id* | *cid*<*type*, ...>.*assoc*
type ← *assoc*

Inside `distance` we use the following three kinds of statements. The `let` statement introduces a variable bound to the value of the expression on the right-hand side. The scope is to the end of the enclosing block and the type of the variable is the type of the right-hand side. $\mathcal{G}$ includes `while`, `return`, and the usual control constructs of C++.

stmt ← `let` *id* `=` *expr*`;` | `while (`*expr*`)` *stmt* | `return` *expr*`;` | ...

Multiple functions with the same name may be defined, and static overload resolution is performed by $\mathcal{G}$ to decide which function to invoke at a particular call site depending on the argument types and also depending on which model definitions are in scope. When more than one overload may be called, the more specific overload is called if one exists ("more specific" is a preorder). The `where` clause and the concept refinement hierarchy are a factor in the ordering.

The syntax for polymorphic classes, structs, and unions is defined below. The grammar variable *clid* is for class, struct, and union names.

decl ← `class` *clid polyhdr* `{` *classmem* ... `};`
decl ← `struct` *clid polyhdr* `{` *mem* ... `};`
decl ← `union` *clid polyhdr* `{` *mem* ... `};`
mem ← *type id*`;`
classmem ← *mem*
| *polyhdr clid*`(`*type pass* [*id*]`,` ...`)` `{` *stmt* ... `}`
| `~`*clid*`()` `{` *stmt* ... `}`
polyhdr ← [`<`*tyid*`,`...`>`] [`where {` *constraint*`,` ... `}`]

Classes consist of data members, constructors, and a destructor. There are no member functions; normal functions are used instead. Data encapsulation (`public`/`private`) is specified at the module level instead of inside the class. Class, struct, and unions are used as types using the syntax below. Such a type is well-formed if the type arguments are well-formed and if the requirements in its where clause are satisfied.

type ← *clid*[`<`*type*`,` ...`>`]

The syntax for calling functions (or polymorphic functions) is the C-style notation:

expr ← *expr*`(`*expr*`,` ...`)`

Arguments for the type parameters of a polymorphic function need not be supplied at the call site: $\mathcal{G}$ will deduce the type arguments by unifying the types of the arguments with the types of the parameters and then ***implicitly instantiate*** the polymorphic function. All of the requirements in the `where` clause must be satisfied by model definitions in the lexical scope preceding the function call. The following is a program that calls the `distance` function, applying it to iterators of type `int*`.

```
fun main() -> int@ {
  let p = new int[8];
  let d = distance(p, p + 4);
  return d - 4;
}
```

A polymorphic function may be explicitly instantiated using this syntax:

expr ← *expr*`<|`*ty*`,` ...`|>`

4 Overview of the STL

The high-level structure of the STL is shown in Fig. 2. The STL contains over fifty generic algorithms. The STL generic algorithms are implemented in terms of a family of iterator abstractions, and the STL containers each provide iterators. As a result, the STL algorithms may be used with any of the STL containers. In fact, the STL algorithms may be used with any data structure that exports iterators with the required capabilities.

Fig. 3 shows the hierarchy of STL's iterator concepts. An arrow indicates that the source concept is a refinement of the target. The iterator concepts arose from the requirements of algorithms: the need to express the minimal requirements for each algorithm. For example, the `merge` algorithm passes through a sequence once, so it only requires the basic requirements of Input Iterator. On the other hand, `sort_heap` requires iterators that can jump arbitrary distances, so it requires Random Access Iterator.

The STL includes a handful of common data structures. When one of these data structures does not fulfill some specialized purpose, the programmer is encouraged to implement the appropriate specialized data structure. All of the STL algorithms can then be made available for the new data structure at the small cost of implementing iterators for the specialized data structure.

Many of the STL algorithms are higher-order: they take functions as parameters, allowing the user to customize the algorithm to their own needs. The STL defines over 25 function objects for creating and composing functions.

The STL also contains a collection of adaptor classes, which are parameterized classes that implement some concept in terms of the type parameter (which is the adapted type). For example, the `back_insert_iterator` adaptor implements Output Iterator in terms of any model of Back Insertion Sequence. The generic `copy` algorithm can then be used with `back_insert_iterator<list<int>>` to append some integers to a list. Adaptors play an important role in the plug-and-play nature of the STL and enable a high degree of reuse.

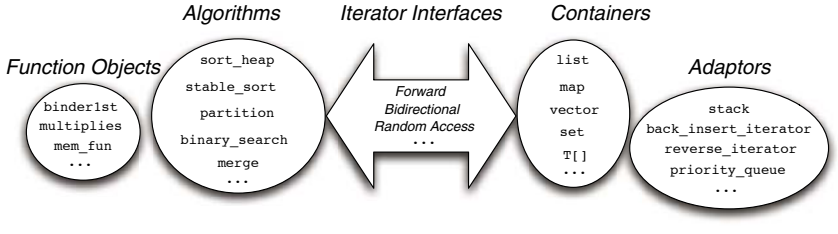


Fig. 2. High-level structure of the STL

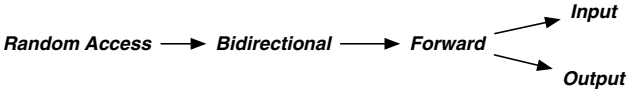


Fig. 3. Iterator concept hierarchy

5 Analysis of $\mathcal{G}$ and the STL

In this section we analyze the interdependence of the language features of $\mathcal{G}$ and generic library design in light of implementing the STL. A primary goal of generic programming is to express algorithms with minimal assumptions about data abstractions, so we first look at how the generic functions of $\mathcal{G}$ can be used to accomplish this. Another goal of generic programming is efficiency, so we investigate the use of function overloading in $\mathcal{G}$ to accomplish automatic algorithm selection. We conclude this section with a brief look at implementing generic containers and adaptors in $\mathcal{G}$.

Algorithms. Fig. 4 depicts a few simple STL algorithms implemented using generic functions in $\mathcal{G}$. The STL provides two versions of most algorithms, such as the overloads for `find` in Fig. 4. The first version is higher-order, taking a predicate function as its third parameter while the second version relies on `operator==`. Functions are first-class in $\mathcal{G}$, so the higher-order version is straightforward to express. As is typical in the STL, there is a high-degree of internal reuse: `remove` uses `remove_copy` and and `find`.

```
fun find<Iter> where { InputIterator<Iter> }
(Iter@ first, Iter last,
 fun(InputIterator<Iter>.value)->bool@ pred) -> Iter@ {
  while (first != last and not pred(*first)) ++first;
  return first;
}
fun find<Iter> where { InputIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  while (first != last and not (*first == value)) ++first;
  return first;
}
fun remove<Iter> where { MutableForwardIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  first = find(first, last, value);
  let i = @Iter(first);
  return first == last ? first : remove_copy(++i, last, first, value);
}
```

Fig. 4. Some STL Algorithms in $\mathcal{G}$

Iterators. Fig. 5 shows the STL iterator hierarchy as represented in $\mathcal{G}$. Required operations are expressed in terms of function signatures, and associated types are expressed with a nested `type` requirement. The refinement hierarchy is established with the `refines` clauses and nested model requirements with `require`. The semantic invariants and complexity guarantees of the iterator concepts are not expressible in $\mathcal{G}$ as they are beyond the scope of its type system.

Automatic Algorithm Selection. To realize the generic programming efficiency goals, $\mathcal{G}$ provides mechanisms for automatic algorithm selection. The following code shows two overloads for `copy`. (We omit the third overload to save space.) The first version is for

```
concept InputIter<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>;
  require SignedIntegral<difference>;
  fun operator*(X) -> value@;
  fun operator++(X!) -> X!;
};
concept OutputIter<X,T> {
  refines Regular<X>;
  fun operator<<(X!, T) -> X!;
};
concept ForwardIter<X> {
  refines DefaultConstructible<X>;
  refines InputIter<X>;
  fun operator*(X) -> value;
};
concept MutableForwardIter<X> {
  refines ForwardIter<X>;
  refines OutputIter<X,value>;
  require Regular<value>;
  fun operator*(X) -> value!;
};
```

```
concept BidirectionalIter<X> {
  refines ForwardIter<X>;
  fun operator--(X!) -> X!;
};
concept MutableBidirectionalIter<X> {
  refines BidirectionalIter<X>;
  refines MutableForwardIter<X>;
};
concept RandomAccessIter<X> {
  refines BidirectionalIter<X>;
  refines LessThanComparable<X>;
  fun operator+(X, difference) -> X@;
  fun operator-(X, difference) -> X@;
  fun operator-(X, X) -> difference@;
};
concept MutableRandomAccessIter<X> {
  refines RandomAccessIter<X>;
  refines MutableBidirectionalIter<X>;
};
```

Fig. 5. The STL Iterator Concepts in $\mathcal{G}$ (`Iterator` has been abbreviated to `Iter`)

input iterators and the second for random access, which uses an integer counter thereby allowing some compilers to better optimize the loop. The two signatures are the same except for the `where` clause. We call this ***concept-based overloading***.

```
InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (; first != last; ++first) result << *first;
  return result;
}
fun copy<Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (n = last - first; n > zero(); --n, ++first) result << *first;
  return result;
}
```

The use of dispatching algorithms such as `copy` inside other generic algorithms is challenging because overload resolution is based on the surrogate models from the `where` clause and not on models defined for the instantiating type arguments. (This rule is needed for separate type checking and compilation). Thus, a call to an overloaded function such as `copy` may resolve to a non-optimal overload. Consider the following implementation of `merge`. The `Iter1` and `Iter2` types are required to model `InputIterator` and the body of `merge` contains two calls to `copy`.

```
fun merge<Iter1,Iter2,Iter3>
where { InputIterator<Iter1>, InputIterator<Iter2>,
        LessThanComparable<InputIterator<Iter1>.value>,
        InputIterator<Iter1>.value == InputIterator<Iter2>.value,
        OutputIterator<Iter3, InputIterator<Iter1>.value> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ { ...
  return copy(first2, last2, copy(first1, last1, result));
}
```

This merge function always calls the slow version of copy even though the actual iterators may be random access. In C++, with tag dispatching, the fast version of copy is called because the overload resolution occurs after template instantiation. However, C++ does not have separate type checking for templates.

To enable dispatching for copy, the type information at the instantiation of merge must be carried into the body of merge (suppose it is instantiated with a random access iterator). This can be done with a combination of concept and model declarations. First, define a concept with a single operation that corresponds to the algorithm.

```
concept CopyRange<I1,I2> {
  fun copy_range(I1,I1,I2) -> I2@;
};
```

Next, add a requirement for this concept to the type requirements of merge and replace the calls to copy with the concept operation copy_range.

```
fun merge<Iter1,Iter2,Iter3>
where { ..., CopyRange<Iter2,Iter3>, CopyRange<Iter1,Iter3> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ { ...
  return copy_range(first2, last2, copy_range(first1, last1, result));
}
```

The final step of the idiom is to create parameterized model declarations for CopyRange. The where clauses of the model definitions match the where clauses of the respective overloads for copy. In the body of each copy_range there is a call to copy which will resolve to the appropriate overload.

```
model <Iter1,Iter2> where { InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
model <Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
```

A call to merge with a random access iterator will use the second model to satisfy the requirement for CopyRange. Thus, when copy_range is invoked inside merge, the fast version of copy is called. A nice property of this idiom is that calls to generic

```
struct list_node<T> where { Regular<T>, DefaultConstructible<T> } {
  list_node<T>* next; list_node<T>* prev; T data;
};
class list<T> where { Regular<T>, DefaultConstructible<T> } {
  list() : n(new list_node<T>()) { n->next = n; n->prev = n; }
  ~list() { ... }
  list_node<T>* n;
};
class list_iterator<T> where { Regular<T>, DefaultConstructible<T> } {
  ... list_node<T>* node;
};
fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T> x) -> T { return x.node->data; }

fun operator++<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T>! x) -> list_iterator<T>!
    { x.node = x.node->next; return x; }

fun begin<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@
    { return @list_iterator<T>(l.n->next); }

fun end<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@ { return @list_iterator<T>(l.n); }
```

Fig. 6. Excerpt from a doubly-linked list container in $\mathcal{G}$

algorithms need not change. A disadvantage of this idiom is that the interface of the generic algorithms becomes more complex.

Containers. The containers of the STL are implemented in $\mathcal{G}$ using polymorphic classes. Fig. 6 shows an excerpt of the doubly-linked `list` container in $\mathcal{G}$. As usual, a dummy sentinel node is used in the implementation. With each STL container comes iterator types that translate between the uniform iterator interface and data-structure specific operations. Fig. 6 shows the `list_iterator` which implements `operator*` in terms of `x.node->data` and `operator++` with `x.node = x.node->next`.

Not shown in Fig. 6 is the implementation of the mutable iterator for `list` (the `list_iterator` provides read-only access). The definitions of the two iterator types are nearly identical, the only difference is that `operator*` returns by read-only reference for the constant iterator whereas it returns by read-write reference for the mutable iterator. The code for these two iterators should be reused but $\mathcal{G}$ does not yet have a language mechanism for this kind of reuse.

In C++ this kind of reuse can be expressed using the Curiously Recurring Template Pattern (CRTP) and by parameterizing the base iterator class on the return type of `operator*`. This approach can not be used in $\mathcal{G}$ because the parameter passing mode may not be parameterized. Further, the semantics of polymorphism in $\mathcal{G}$ does not match the intended use here, we want to *generate* code for the two iterator types at library construction time. A separate *generative* mechanism is needed to complement the generic features of $\mathcal{G}$. As a temporary solution, we used the m4 macro system to factor the com-

mon code from the iterators. The following is an excerpt from the implementation of the iterator operators.

```
define('forward_iter_ops',
'fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
($1<T> x) -> T $2 { return x.node->data; } ...')
forward_iter_ops(list_iterator, &) /* read-only */
forward_iter_ops(mutable_list_iter, !) /* read-write */
```

Adaptors. The `reverse_iterator` class is a representative example of an STL adaptor.

```
class reverse_iterator<Iter>
  where { Regular<Iter>, DefaultConstructible<Iter> }
{
  reverse_iterator(Iter base) : curr(base) { }
  reverse_iterator(reverse_iterator<Iter> other) : curr(other.curr) { }
  Iter curr;
};
```

The `Regular` requirement on the underlying iterator is needed for the copy constructor of `reverse_iterator` and `DefaultConstructible` is needed for the default constructor. This adaptor flips the direction of traversal of the underlying iterator, which is accomplished with the following `operator*` and `operator++`. There is a call to `operator--` on the underlying `Iter` type so `BidirectionalIterator` is required.

```
fun operator*<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter> r) -> BidirectionalIterator<Iter>.value
    { let tmp = @Iter(r.curr); return *--tmp; }

fun operator++<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter>! r) -> reverse_iterator<Iter>!
    { --r.curr; return r; }
```

Polymorphic model definitions are used to establish that `reverse_iterator` is a model of the iterator concepts. The following says that `reverse_iterator` is a model of `InputIterator` whenever the underlying iterator is a model of `BidirectionalIterator`.

```
model <Iter> where { BidirectionalIterator<Iter> }
InputIterator< reverse_iterator<Iter> > {
  type value = BidirectionalIterator<Iter>.value;
  type difference = BidirectionalIterator<Iter>.difference;
};
```

6 Component Development Benefits

Generic programming has enabled programmers from all over the world to construct and share interchangeable components. An example of this is the Boost collection of C++ libraries [17]. While this has benefited programmer productivity, there is room to improve: the cost of reuse is still too high. Programmers routinely run into component integration problems such as namespace pollution, libraries with type errors, documentation inconsistencies, long compile times, and hard to understand error messages. Many languages provide the necessary modularity to solve these problems, but lack the

abstractions to express the STL. On the other hand, C++ can easily express the STL but lacks modularity. The point of this section is to show that not only is $\mathcal{G}$ suitable for expressing the STL but it also provides modularity.

Namespace pollution issues related to cpp macros are an old story, but generic programming brings with it new and subtle issues. For example, function templates in C++ rely on argument dependent lookup (ADL) [18] to access user-defined operations, but ADL breaks namespace modularity. There is tension in between the need to allow for user-supplied operations while at the same time ensuring modularity. In $\mathcal{G}$ this tension is resolved with concepts and `where` clauses that provide a mechanism for specifying rich interfaces while at the same time separating library and user namespaces.

Users of generic libraries in C++ are plagued by long compile times and hard to understand error messages. The reason is C++'s lack of separate compilation and separate type checking. $\mathcal{G}$ addresses both of these problems. In $\mathcal{G}$, generic libraries can be compiled to object code so the user need only link them to the executable. Many of the hard to understand error messages in C++ come from misuses of generic algorithms. For example, the following $\mathcal{G}$ program misuses `stable_sort`: it requires a random access iterator but `list` only provides bidirectional.

```
4  fun main() -> int@{
5    let v = @list<int>();
6    stable_sort(begin(v), end(v));
7    return 0;
8  }
```

In C++ this would evoke pages of error messages with line numbers pointing deep inside the implementation of `stable_sort`. In contrast, the $\mathcal{G}$ compiler prints the following:

```
application stable_sort(begin(v), end(v)), Model
MutableRandomAccessIterator<mutable_list_iter<int>> needed to
satisfy requirement, but it is not defined.
```

Another problem that plagues generic C++ libraries is that type errors often go unnoticed during library development. This is because type checking of templates is delayed until instantiation. A related problem is that the implementation may not be consistent with the documented type requirements for a template, which can result in unexpected compiler errors for the user.

These problems are directly addressed in $\mathcal{G}$: the implementation of a generic function is type-checked with respect to its `where` clause. Thus, when a generic function successfully compiles, it is guaranteed to be free of type errors and the implementation is guaranteed to be consistent with the type requirements in the `where` clause.

Interestingly, while implementing the STL in $\mathcal{G}$, the type checker caught several errors in the STL as defined in C++. One such error was in `replace_copy`. The implementation below was translated directly from the GNU C++ Standard Library, with the `where` clause matching the requirements for `replace_copy` in the C++ Standard [18].

```
196  fun replace_copy<Iter1,Iter2, T> where { InputIterator<Iter1>,
197  Regular<T>, EqualityComparable<T>,
198           OutputIterator<Iter2, InputIterator<Iter1>.value>,
199           OutputIterator<Iter2, T>,
200           EqualityComparable2<InputIterator<Iter1>.value,T> }
201  (Iter1@ first, Iter1 last, Iter2@ result, T old, T neu) -> Iter2@ {
```

```
202    for ( ; first != last; ++first)
203      result << *first == old ? neu : *first;
204    return result;
205  }
```

The $\mathcal{G}$ compiler gives the following error message:

```
two branches of the conditional expression must have the same type
or one must be coercible to the other.
```

This is a subtle bug, which explains why it has gone unnoticed for so long. The type requirements say that both the value type of the iterator and `T` must be writable to the output iterator, but the requirements do not say that the value type and `T` are the same type, or coercible to one another.

7 Related Work

There is a long history of programming language support for polymorphism, dating back to the 1970s [15, 16, 19, 20]. An early precursor to $\mathcal{G}$'s concept feature can be seen in CLU's type set feature [19]. In mathematics, the equivalent notion of algebraic structure has been in use for an even longer time [21].

The concept feature in $\mathcal{G}$ is heavily influenced by the type class feature of Haskell [22], with its nominal conformance and explicit model definitions. However, $\mathcal{G}$'s support for associated types, same type constraints, and concept-based overloading is novel. Also, $\mathcal{G}$'s type system is fundamentally different from Haskell's: it is based on System F [15, 16] instead of Hindley-Milner type inferencing [20]. This difference has some repercussions. In $\mathcal{G}$ there is more control over the scope of concept operations because `where` clauses introduce concept operations into the scope of the body. This difference allows Haskell to infer type requirements but induces the restriction that two type classes in the same module may not have operations with the same name. A difference we discuss in [14] is that in $\mathcal{G}$, overlapping models may coexist in separate scopes but still be used in the same program, whereas in Haskell overlapping models may not be used in the same program. Haskell performed quite well in our comparative study of support for generic programming [13]. However, we pointed out that Haskell was missing support for associated types and work to remedy this has been reported in [23, 24].

Less closely related to $\mathcal{G}$ are languages based on subtype-bounded polymorphism [25] such as Java, C#, and Eiffel. We found subtype-bounded polymorphism less suitable for generic programming and refer the reader to [13] for an in-depth discussion. More recently, the object-oriented language Scala [26] has added abstract type members based of the theory of dependent types. A comparison of this with $\mathcal{G}$'s associated types is planned for future work.

8 Conclusion

This paper presented the design of a new programming language named $\mathcal{G}$ and demonstrated with an implementation of the Standard Template Library that this language is well-suited to generic programming. We were able to implement all of the abstractions

in the STL in a straightforward manner. Further, $\mathcal{G}$ is particularly well-suited for the development of reusable components due to its support of separate type checking and compilation. $\mathcal{G}$'s strong type system provides support for the independent validation of components and $\mathcal{G}$'s system of concepts and constraints allows for rich interactions between components without sacrificing namespace safety. As a result, the language features present in $\mathcal{G}$ hold some promise to increase programmer productivity with respect to the development and use of generic components.

Acknowledgments

We would like to thank Ronald Garcia, Jeremiah Willcock, Doug Gregor, Jaakko Järvi, Dave Abrahams, Dave Musser, and Alexander Stepanov for many discussions and collaborations that informed this work. This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

References

1. Kapur, D., Musser, D.R., Stepanov, A.: Operators and algebraic structures. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, ACM (1981)
2. Musser, D.R., Stepanov, A.A.: Generic programming. In Gianni, P.P., ed.: Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings. Volume 358 of Lecture Notes in Computer Science., Berlin, Springer Verlag (1989) 13–25
3. Musser, D.R., Stepanov, A.A.: A library of generic algorithms in Ada. In: Using Ada (1987 International Ada Conference), New York, NY, ACM SIGAda (1987) 216–225
4. Kershenbaum, A., Musser, D., Stepanov, A.: Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute (1988)
5. Stroustrup, B.: Parameterized types for C++. In: USENIX C++ Conference. (1988)
6. Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (1994)
7. Austern, M.H.: Generic Programming and the STL. Professional computing series. Addison-Wesley (1999)
8. Köthe, U.: Reusable Software in Computer Vision. In: Handbook on Computer Vision and Applications. Volume 3. Acadamic Press (1999)
9. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
10. Boissonnat, J.D., Cazals, F., Da, F., Devillers, O., Pion, S., Rebufat, F., Teillaud, M., Yvinec, M.: Programming with CGAL: the example of triangulations. In: Proceedings of the fifteenth annual symposium on Computational geometry, ACM Press (1999) 421–422
11. Pitt, W.R., Williams, M.A., Steven, M., Sweeney, B., Bleasby, A.J., Moss, D.S.: The bioinformatics template library: generic components for biocomputing. Bioinformatics **17** (2001) 729–737
12. Troyer, M., Todo, S., Trebst, S., and, A.F.: (ALPS: Algorithms and Libraries for Physics Simulations) `http://alps.comp-phys.org/`.
13. Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 115–134

14. Siek, J., Lumsdaine, A.: Essential language support for generic programming. In: PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2005) 73–84
15. Girard, J.Y.: Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur. Thse de doctorat d'état, Université Paris VII, Paris, France (1972)
16. Reynolds, J.C.: Towards a theory of type structure. In Robinet, B., ed.: Programming Symposium. Volume 19 of LNCS., Berlin, Springer-Verlag (1974) 408–425
17. Boost: (Boost C++ Libraries) `http://www.boost.org/`.
18. International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
19. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Communications of the ACM **20** (1977) 564–576
20. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
21. Bourbaki, N.: Elements of Mathematics. Theory of Sets. Springer (1968)
22. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76
23. Chakravarty, M.M.T., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2005) 1–13
24. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '05), New York, NY, USA, ACM Press (2005)
25. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the fourth international conference on functional programming languages and computer architecture. (1989)
26. Odersky, M., al.: An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)

Mapping Features to Models: A Template Approach Based on Superimposed Variants

Krzysztof Czarnecki and Michał Antkiewicz

University of Waterloo, Canada

Abstract. Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, gives them semantics. In this paper, we propose a general template-based approach for mapping feature models to concise representations of variability in different kinds of other models. We show how the approach can be applied to UML 2.0 activity and class models and describe a prototype implementation.

1 Introduction

Feature modeling is an important method and notation to elicit and represent common and variable features of the systems in a product line. It can be used at any level of abstraction, including requirements, architecture and design, components, and platforms; for any kind of artifacts, such as code, models, documentation; and in all stages of product-line engineering. At an early stage, feature modeling enables product-line scoping, i.e., deciding which features should be supported by a product line and which should not. In design, the points and ranges of variation captured in feature models need to be mapped to a common product-line architecture. Furthermore, feature models allow us to scope and derive domain-specific languages, which are used to specify product-line members in generative software development [1, 2, 3]. Finally, feature models are also useful in product development as a basis for estimating development cost and effort, and automated or manual product derivation.

Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, gives them semantics. In this paper, we propose a general approach for mapping feature models to concise representations of variability in different kinds of other models. In contrast to variability approaches in which separate model fragments corresponding to different features are composed, our approach presents the modeler with a model representing a superimposition of all variants whose elements are related to the corresponding features through annotations. We argue that this approach is particularly desirable at the requirements level, as it directly shows the impact of selecting a given feature on the resulting model. The proposed approach is general; it works for any model whose metamodel is

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 422–437, 2005.

expressed in the Meta-Object Facility (MOF) [4] or a comparable modeling formalism, and it can be easily incorporated into an existing model editor. We give the details of our approach for mapping feature models to UML 2.0 activity diagrams and indicate how it can be applied to other kinds of models. We describe a prototype implementation of our approach. The sample models presented in this paper are taken from a large model of an e-commerce platform, which we used to test our approach.

The remainder of the paper is organized as follows. Section 2 reviews background concepts and related work on feature modeling. Next we describe the idea in sections 3 and 4. Section 5 presents the details of template instantiation algorithm. We describe the implementation of the prototype in section 6. We discuss related work in section 7 and conclude the paper in section 8.

2 Background: Feature Modeling

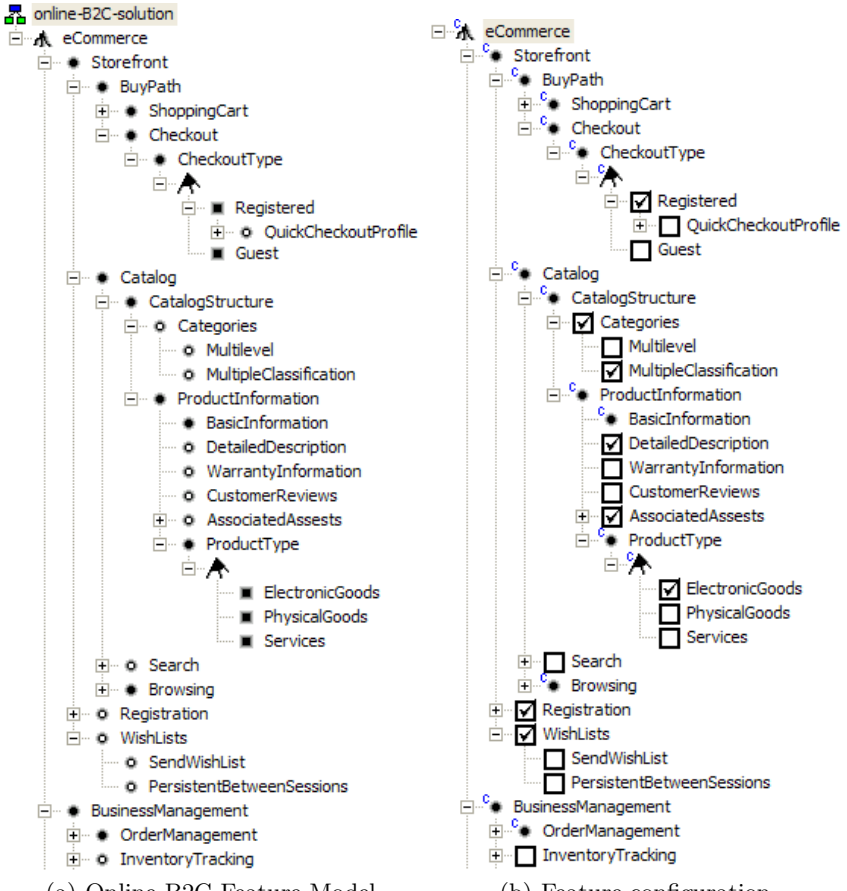
Feature modeling was originally proposed as part of the Feature-Oriented Domain Analysis (FODA) method [5], and since then, it has been applied in a range of business and technical domains (see [6] for list of applications with references). In this work, we use *cardinality-based feature modeling* [7], which extends the original feature modeling from FODA with feature and group cardinalities, feature attributes, feature diagram references, and user-defined annotations.

A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family [1]. Features are organized in *feature diagrams*. A feature diagram is a tree with the root representing a concept (e.g., a software system) and its descendant nodes being features. A *feature model* consists of one or more feature diagrams plus *additional information* such as feature descriptions, global constraints, binding times, priorities, stakeholders, etc.

Figure 1(a) presents a small excerpt from a feature model describing a family of online business-to-consumer (B2C) solutions; the entire model has over 350 features. The model contains one feature diagram, with `eCommerce` as its *root feature*. The root feature has two solitary subfeatures: `Storefront` and `BusinessManagement`. The symbol ● indicates that `Storefront` has a feature cardinality of [1..1]. Feature cardinality is an interval denoting how often a feature with its subfeatures can be cloned as a child of its parent when specifying a concrete system. The cardinality of [1..1] indicates that a feature must exist at least and at most once. On the other hand, the symbol ○ indicates that `WishLists` is an optional feature with cardinality [0..1]. Available checkout types `Registered` and `Guest`, are members of a *feature group*. The group symbol ▲ indicates group cardinality $\langle 1\text{–}k \rangle$, where k is the group size. Thus available checkout types can be any non-empty subset of the two checkout types. *Grouped features* are indicated by the symbol ..■

One can specify additional constraints such as *requires* or *excludes*. For example, the feature `PersistentBetweenSessions` requires the system to implement `Registration` because a wish list is stored in a customer's account. Also, check-

out type `Registered` requires feature `Registration` to be selected. In general, additional constraints in cardinality-based feature models require tree-oriented navigation and query facilities, and may involve logic, arithmetic, string, and set operators on feature attributes and feature sets. Such constraints can be adequately expressed using XPath 2.0 [8].



(a) Online B2C Feature Model (b) Feature configuration

Fig. 1. Sample online B2C feature model and its feature configuration

Semantically, a feature model describes a set of all possible valid *configurations* [7]. Figure 1(b) presents a sample configuration of the online B2C feature model. A configuration specifies a concrete system. In this example, checkout for registered customers is the only available checkout type, the catalog is subdivided into categories, a product can be classified in multiple categories, the catalog contains only electronic goods, etc.

3 Basic Idea: Superimposed Variants

An overview of our approach is shown in Figure 2. A *model family* is represented by a *feature model* and a *model template*. The feature model defines a hierarchy of features together with the constraints on their possible configurations. The model template contains the union of the model elements in all valid *template instances*. The set of the valid template instances corresponds to the extent of the model family. The model template is itself a model expressed in the same *target notation* as the template instances. For example if we want to represent a family of UML activity models, both the model template and the template instances will be expressed using the UML activity modeling notation. The elements of a model template may be annotated using *presence conditions* (PCs) and *meta-expressions* (MEs). These annotations are defined in terms of features and feature attributes from the feature model, and can be evaluated with respect to a feature configuration. A PC attached to a model element indicates whether the element should be present in or removed from a template instance. MEs are used to compute attributes of model elements, such as the name of an element or the return type of an operation.

An instance of a model family can be specified by creating a feature configuration based on the feature model. Based on the feature configuration, the model template is instantiated automatically. The instantiation process is a *model-to-model transformation* with both the input and output expressed in the target notation. It involves evaluating the PCs and MEs with respect to the feature configuration, removing model elements whose PCs evaluate to false and, possibly, additional processing such as simplification (Section 5).

A particularly useful form of PCs are Boolean formulas over the set of variables, where each variable corresponds to a feature from the feature model. Given a feature configuration, the value of a given Boolean variable is true if and only if the corresponding feature is included in the feature configuration.

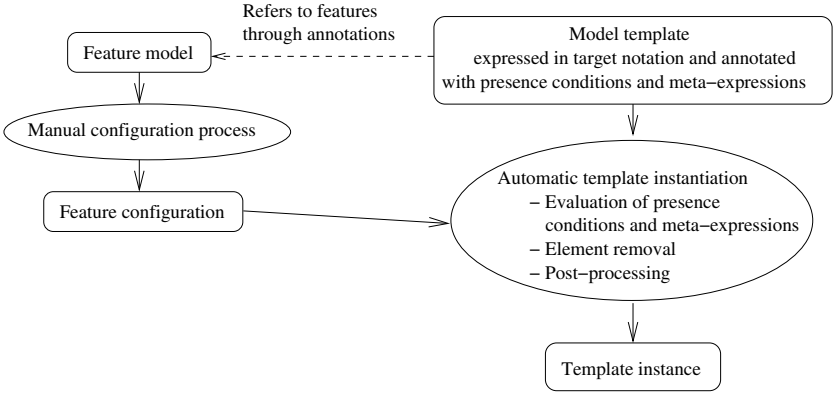
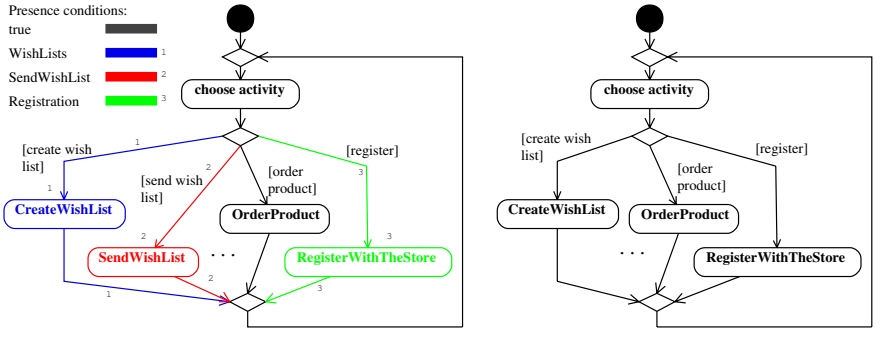


Fig. 2. Overview of the approach



(a) Storefront template (b) Storefront instance

Fig. 3. Sample template activity diagram and its instance

As an example, consider the UML activity diagram in Figure 3(a), which models the top-level activity of a storefront. This diagram is a template since elements relevant to features `WishLists`, `SendWishList` and `Registration` have been annotated with their PCs. The annotations are rendered using a coloring scheme in which each different PC is assigned a different color.[1] In this simple example, each PC consists of a single variable corresponding to a single feature. Figure 3(b) shows a template instance created based on the feature configuration from Figure 1(b). PCs corresponding to feature `SendWishList`, which is not included in the configuration, evaluated to false and the annotated elements were removed from the template.

It is important to note that PCs are interpreted locally with respect to containment hierarchies defined in the metamodel of the target notation. In other words, a PC on an element controls the presence of that element only with respect to its container; if the container is removed, all contained elements are also removed, regardless of their PCs. For that reason, we did not have to annotate the guards on flows in Figure 3(a) because they are contained in the flows according to the UML metamodel.

More complex PCs can be expressed using XPath [9]. Such conditions can access feature attributes, count the number of feature clones in a configuration, and use other XPath operations, as long as the XPath expression evaluates to a Boolean value. If necessary, XPath can be easily extended with user-defined functions.

MEs may be used to compute attributes of basic types, as well as references to model elements. In this paper, we only consider computing references to already existing elements. MEs can be expressed using XPath. As an exam-

[1] The colors are assigned per diagram, and the number of colors needed is limited since diagrams are usually split such that each diagram can fit on the computer screen. Note that colors in this paper are indexed in order for the annotations to be readable in black and white.

ple, consider the activity diagram fragment in Figure 4. The type of the input pin of action `DisplayProducts` is set to `b2cSoln::Category` or `b2cSoln::Catalog` depending on the presence of the `Categories` feature in the feature configuration.

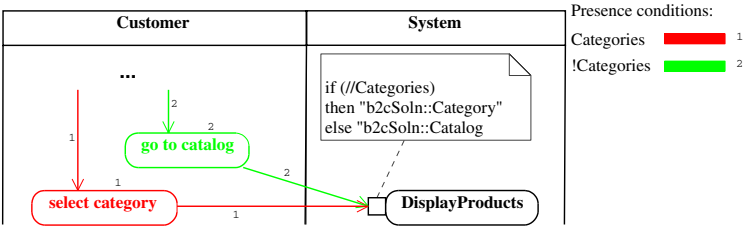


Fig. 4. Example of a type meta-expression

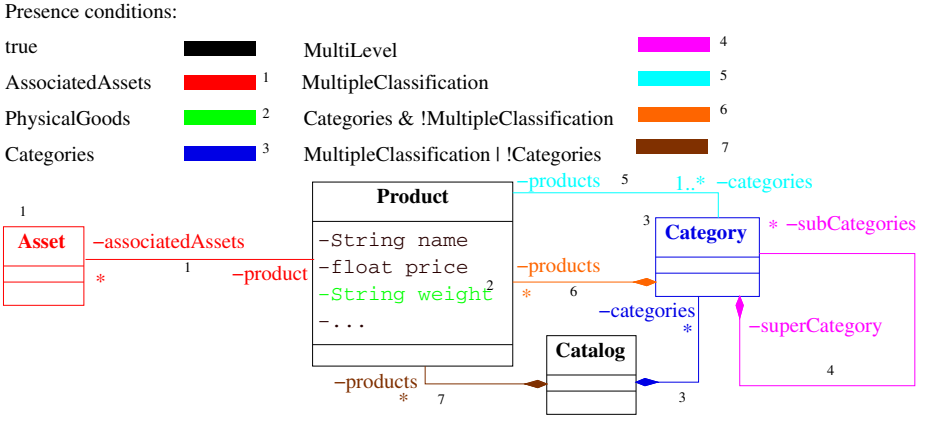


Fig. 5. Example of annotated class diagram

Figure 5 shows an annotated class diagram. Class `Category` is present in a template instance if the feature `Categories` is selected. Feature `MultiLevel` implies a containment hierarchy for `Category`. `MultipleClassification` implies that `Products` can be classified under multiple categories. `AssociatedAssets` implies the class `Asset`, which can be used for storing documents such as technical specifications and manuals and other media. Finally, `PhysicalGoods` implies the attribute `weight` in `Product`.

The realization of our approach for a given target notation involves the following steps:

1. decide on the form of PCs and MEs, for example Boolean formulas and/or XPath expressions;
2. decide on *implicit PCs*. Model elements that are not explicitly annotated by the user will have implicit PCs; implicit PCs will be explained shortly;
3. decide on the annotation mechanism and rendering options for the annotations, e.g., if the target notation is UML, the annotations can be realized as stereotypes; rendering options include labels, icons, and/or coloring;
4. decide on additional processing.

Steps 2–4 depend on the target notation. In the following sections, we will demonstrate details for UML activity diagrams as a target notation. [2]

4 Implicit Presence Conditions

When an element has not been explicitly assigned a PC by the user, an implicit PC (IPC) is assumed. In general, assuming a PC of `true` is a simple choice which is mostly adequate in practice; however, sometimes a more useful IPC for an element of a given type can be provided based on the presence conditions of other elements and the syntax and semantics of the target notation. For example, according to UML syntax, a binary association requires a classifier at each of its ends. Thus, a reasonable choice of IPC for a binary association would be the conjunction of the PCs of both classifiers. This way, removing any of the classifiers will also lead to the removal of the association. IPCs reduce the necessary annotation effort of the user. For example, given the IPC for associations as described, the association between `Product` and `Asset` in Figure 5 does not need to be annotated explicitly.

Table 1 shows our choice of IPCs for UML class and activity model elements. An IPC for a given element is assumed based on its type. In order to determine the IPC for a given model element, we look up the closest matching supertype in Table 1 and take the corresponding IPC. For example, the IPC for instances of `Class` and `Action` is `true` because their closest matching type in Table 1 is `Element`. Since `ActivityFinalNode` and `FlowFinalNode` are subclasses of `FinalNode` according to the UML metamodel, the IPC for `ActivityFinalNode` and `FlowFinalNode` is the same as for `FinalNode` in Table 1.

The choice of IPCs in Table 1 reflects the cardinality and other integrity constraints specified in the UML metamodel. For class models, the IPC for all elements except relationships is `true`. In the case of generalization, which is a binary relationship, the IPC reflects the fact that such a relationship can exist in a template instance only if the classifiers at both ends of the relationship are also present in the template instance. The IPCs for dependencies and associations need to handle the more general case of n-ary relationships. For activity models, the IPC for all elements except control nodes, central buffer node, and call actions is `true`. Control nodes have to have at least one incoming flow and/or at least

[2] For simplicity, we limit ourselves to the intermediate level of activity diagrams as defined in the UML Superstructure document [10].

Table 1. Implicit presence conditions for UML class and activity model elements

Model kind	Element type	Implicit presence condition[a]
General	`Element`	`true`
Class	`Generalization`	Conjunction of the PCs of the `general` and `specific` classifiers
	`Dependency`	`true` iff at least one `client` element's PC evaluates to `true` and at least one `supplier` element's PC evaluates to `true`
	`Association`	`true` iff two or more `memberEnd` properties such that each has a classifier with PCs evaluating to `true` as its type
Activity	`InitialNode`	Disjunction of the PCs of all `outgoing` flows
	`FinalNode`	Disjunction of the PCs of all incoming flows
	`DecisionNode` and `ForkNode`	`true` iff exactly one `incoming` flow's PC evaluates to `true` and one or more `outgoing` flows' PCs evaluate to `true`
	`MergeNode` and `JoinNode`	`true` iff exactly one `outgoing` flow's PC evaluates to `true` and one or more `incoming` flows' PCs evaluate to `true`
	`CentralBufferNode`	Disjunction of the PCs of all `incoming` and `outgoing` flows
	`CallOperationAction`	true iff accumulated PC of the called `operation` evaluates to true
	`CallBehaviorAction`	true iff accumulated PC of the called `behavior` evaluates to true

[a] PC stands for presence condition (both explicit or implicit). Names in typewriter font (except `true`) refer to properties of the corresponding element.

one outgoing flow in an instance as specified in the metamodel. Central buffer has to have at least one incoming flow or outgoing flow. Finally, the target of call actions has to be present.

Control nodes are not intended to be annotated with PCs explicitly since their IPCs will always be adequate. This is not true for relationships in class models because we might want to remove a relationship in a template instance even if the elements the relationship connects are not removed.

IPCs for call actions relfect the fact that removal of the target should also force removal of all actions calling it. Accumulated PC of an element is true iff PCs of all parents of that element evaluate to true.

5 Template Instantiation

A simple and general template instantiation process involves computing MEs and removing elements whose PCs are false; however, the general process can be specialized for a given notation with some additional processing steps, which allow expressing templates in that notation more compactly. We have identified two categories of such additional steps: *patch application* and *simplification*. A patch is a transformation that automatically fixes a problem which may result from removing elements. It is defined for situations in which there exists a unique and intuitive solution to a problem created by element removal.

Simplification involves removing elements that have become redundant after removing other elements. In the case of activity models, we found it useful to provide *automatic flow closure* as a patch and *removal of redundant control nodes* for simplification, which will be explained later.

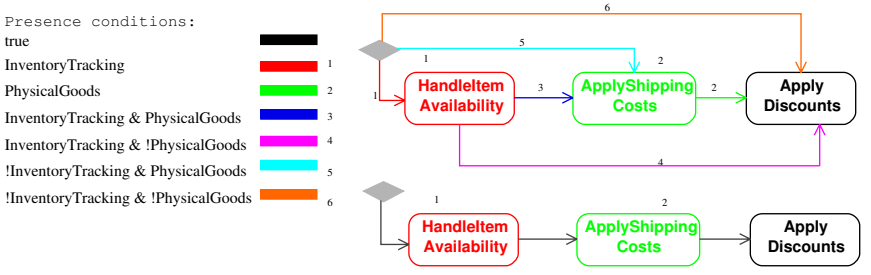


Fig. 6. Model templates with two optional actions without and with automatic flow closure

The motivating example for automatic flow closure is presented in Figure 6. The two actions *HandleItemAvailability* and *ApplyShippingCosts* are optional and implement features `InventoryTracking` and `PhysicalGoods`, respectively. The top part of the figure presents how the two optional actions would have to be modeled without automatic flow closure. The bottom part contains the same fragment expressed in a natural way thanks to the automatic flow closure. The latter ensures that after removing an optional action the still remaining incoming flow and outgoing flow will be closed. If desired, the closure can be prevented by the user by annotating the flows such that they are removed together with the action. It is easy to see that, without flow closure, the number of flows needed in a chain of optional actions grows exponentially with the number of the actions.[3]

The complete template instantiation algorithm can be summarized as follows:

1. *Evaluation of MEs and explicit PCs.* The evaluation is done while traversing the element containment hierarchy in the template in depth-first order. Children of elements whose PCs evaluate to false are not visited because they will be removed.
2. *Removal analysis.* Removal analysis involves computing IPCs and information required for patch application, if any. The IPCs in Table 1 can be computed in a single additional pass after computing the explicit PCs; however, a different choice of IPCs could require multiple iterations. Furthermore, given the IPCs in Table 1, the necessary analysis for automatic flow closure can be performed separately after the IPCs are computed. Again, depending on the choice of IPCs and patches, such separation may not be possible.
3. *Element removal and patch application.* In this step, elements whose PCs are false are removed and patches, if any, are applied. Application of a patch depends on its type and can be performed before or after removal.
4. *Simplification.* Simplification is performed at last.

[3] It is interesting to note that removing an optional action from a sequence of actions in an activity model corresponds to removing an optional statement from a statement list in a textual language, e.g., when the C preprocessor removes a statement within `#ifdef` and `#endif` in a C program. In the latter case, however, flow closure happens naturally without the need for any additional processing.

5.1 Template Instantiation for Activity Diagrams

The removal analysis for activity diagrams identifies situations where flows interrupted by removed elements can be closed. The identification is performed during removal analysis after all IPCs have been computed and proceeds as follows. Let F be the set of elements contained in an activity whose PCs (both explicit and implicit) evaluated to false. We partition F into a set of regions R such that elements in each region are connected, but no two elements from two different regions are connected. Furthermore, let A_r be a set of flows *adjacent* to the region r.

A region r is said to be *closable* iff

1. there is exactly one incoming and exactly one outgoing adjacent flow,[4] i.e., $A_r = \{i, o\} \wedge target(i) \in r \wedge source(o) \in r$
2. there is a flow path connecting $target(i)$ and $source(o)$
3. types of i and o are consistent i.e., both are control or object flows

All closeable regions are closed before elements with PCs being false are removed. Closing a region r with A_r means that o is removed and the target of i is set to the target of o. If a region is not closeable and A_r is not empty, there is an annotation error because flows from A_r would become dangling after the removal of region r. An annotation error can also occur if a flow is not within the region itself, but its ends are.

Simplification for activity nodes involves removing redundant control nodes such as (1) a `DecisionNode` or a `ForkNode` having one outgoing flow, and (2) a `MergeNode` or a `JoinNode` having one incoming flow. More sophisticated control flow simplification could also be applied at this point, such as merging parallel flows without actions between a decision and merge nodes.

As an example of template instantiation for activity models with automatic flow closure, consider the checkout items template in Figure 7 and its instance in Figure 8(b). The instance implements the features selected in the feature configuration from Figure 1(b), where the only specified checkout method is for registered customers (feature `Registered`). Features `Guest`, `QuickCheckoutProfile`, `InventoryTracking` and `PhysicalGoods` are not selected. Note that all control nodes in the template are gray, indicating that they are not annotated and therefore IPCs are assumed.

The result of removal analysis and patch application is shown in Figure 8(a). Based on our configuration, the instantiation algorithm removes four regions: blue (3) for `QuickCheckoutProfile`, green (2) for `Guest` checkout, magenta (4) for `InventoryTracking`, and pink (5) for `PhysicalGoods`. Note that they are all closeable and will be closed before removal. In the case of the blue (3) region, the adjacent flows are those in red (1). Also note that the decision node `type?` has been included in the blue region because its implicit PC evaluated to false.

[4] We only allow one incoming and one outgoing flow. The reason is that in the case of more than one incoming and/or outgoing flow, there is more than one way to close the flows.

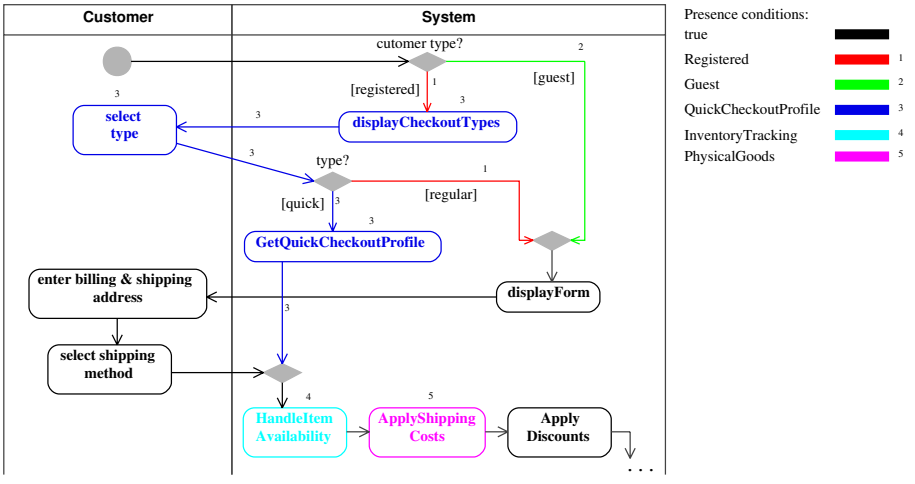
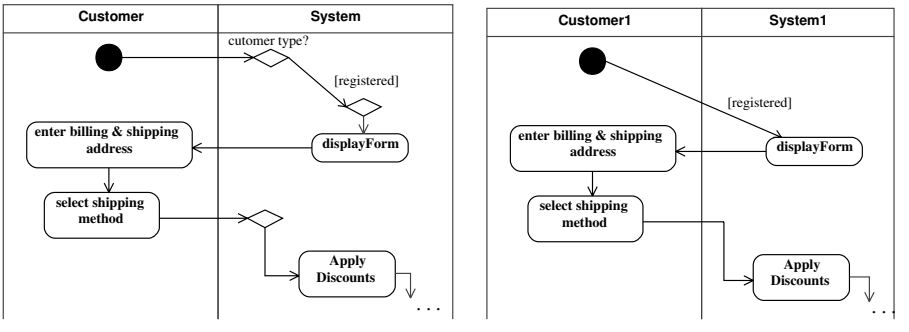


Fig. 7. Checkout Items diagram template



(a) Checkout instance after region removal

(b) Checkout instance after simplification

Fig. 8. Checkout template instance

The final result after simplification, which removed one decision node `customer type?` and two merge nodes, is shown in Figure 8(b).

Examples of useful patches for class models include *generalization chain closure* and *containment chain closure*. They are the counterpart of automatic flow closure for generalization and containment relationships: given the classifiers `A`, `B`, and `C`, if `A` is connected to `B` and `B` is connected to `C`, removing `B` while the PCs of the incoming relationship and the outgoing relationship are true will connect `A` to `C`.

6 Prototype Implementation

We have built a prototype to illustrate how our approach works in practice. The prototype, *fmp2rsm*, is an Eclipse plug-in which integrates our Feature Modeling Plug-In (fmp) [8] with Rational Software Modeler (RSM), a UML modeling tool from IBM.[5] The plug-in implements the template instantiation algorithm from Section 5, and it also performs the automatic coloring of templates based on the PC annotations. The implementation handles all of UML; however, at this point, the convenience of additional processing is available only for activity models as described in Section 5.1.

The plug-in works with four artifacts: (1) UML model template created using RSM, (2) feature model created using fmp (Figure 1 contains screen shots of fmp), and two *variability profiles*, (3) *PC profile* for PC annotations, and (4) *ME profile* for ME annotations.

The PC profile offers two forms of PC annotations: Boolean formulas in Disjunctive Normal Form (DNF) and the more general XPath expressions. Each disjunct (i.e., a conjunction of literals) of a PC in DNF is represented as a stereotype, e.g., `<<f1∧!f2∧f3>>` for the Boolean formula $f1\overline{f2}f3$, and can be created on a selection of multiple features automatically through a menu operation in fmp.[6] Once created, the stereotype becomes available in RSM for annotating template elements. Application of multiple such stereotypes is interpreted as a disjunction. The more general annotations using XPath are created by applying the stereotype `<<PC>>`, which allows the user to enter the desired XPath expression as the value of its `expression:String` property.

The ME profile contains several stereotypes structured similarly to `<<PC>>`, but each applicable to elements of a specific type. For example, `<<NameME>>` can be applied to any element of type `NamedElement` in order to compute its name. Similarly, `<<TypeME>>` can be used to set the `type` property of any `TypedElement`. For `<<TypeME>>`, an expression has to return fully qualified name of either a primitive type (e.g., `UML2::String`), a class (e.g., `b2cSoln::Category` as in Figure 4), an interface, or an enumeration. The ME profile could be automatically generated based on the metamodel of a given notation.

7 Related Work

Variability mechanisms most commonly used in models are those already available in the target notation, such as using a decision node in an activity model to decide between alternative flows or representing class variants as subclasses of an

[5] The fmp2rsm plug-in can be downloaded at `http://gp.uwaterloo.ca/fmp2rsm`.

[6] Each stereotype extends `Element` and has read-only properties encoding the actual Boolean formula. The encoding uses fully qualified feature names which are available as literals of an enumeration type that is automatically generated by fmp2rsm given a feature model. In other words, the stereotype's name is just for documentation and may use abbreviated feature names.

abstract class. Inheritance—a classical variability mechanism in class diagrams—has also been adapted for activities [11] and statecharts [12, 13]. Limitations of these approaches include the lack of static configuration, as in the case of dynamic choice such as a decision node, the potential of combinatorial explosion for static inheritance hierarchies, and complexity increase and limited traceability in the case of design patterns.

Another class of approaches is based on annotations expressing variability. In the case of UML, such annotations are usually provided as a profile with stereotypes, such as `<<optional>>` and `<<variant>>`, e.g., [14]. Although our approach is also annotation-based, we provide a separate representation of variability in the form of a feature model. Without the latter, there is no clear notion of features and the user has to find and select variable elements in the model directly. Furthermore, patching and simplification, as proposed in our approach, results in simpler templates. Finally, we provide full template support by means of MEs.

Wasowski describes automatic generation of variants of behavioral models (in particular statecharts) by restrictions [15]. As in our approach, the modeler creates a single model containing all variants. A variant is automatically created using a form of partial evaluation and slicing based on specified restrictions. The restriction approach differs from our template approach in several ways. First, it involves more sophisticated analysis in which restrictions on inputs and outputs are propagated throughout the model automatically. While this may result in a significantly reduced annotation effort, the effect of automatic partial evaluation and slicing may be hard to predict for the user. In a template approach, the user has full control through explicit annotations. Another difference is that the model to be restricted has to be semantically correct in the sense that it is ready to be executed without any processing, whereas templates only need to be syntactically well-formed. The restriction approach is adequate, particularly if there is a variant that contains all of the initial model without any restrictions; however, if this is not the case, it is likely that a template will be simpler than an unrestricted model. For example, alternative flows can simply be attached to an activity node, while the model restriction approach would require using a decision node. Also, additional processing, such as automatic flow closure, further reduces the complexity of a template, whereas in the model restriction approach, each optional action would need an extra decision node. Finally, the template can be easily adapted to any notation. This is different with the restriction approach, which is semantics-based and needs to be individually developed for each notation.

Another group of work are concern separation approaches, such as AHEAD [16], and the hyperspace approach [17] and its UML realization HyperUML [18]. These approaches allow the composition of crosscutting model fragments. In particular, HyperUML uses feature models to represent the composition space of UML model fragments. Mixin-based composition of statecharts [19, 20] also falls into this category, but with the particular focus on ensuring provably correct composition. Concern separation approaches focus on separation. Templates, on the other hand, work best if the user wants to see the model fragment correspond-

ing to a feature embedded in the context of the entire model. For example, separating the blue (3) region corresponding to the feature `QuickCheckoutProfile` in Figure 8 as a component does not seem to be interesting. This is because the fragment is not reusable and can be best understood in the context where it is applied. Separation approaches are of particular interest when features should be realized as components that can be composed in many ways, which is the case in mature and highly flexible architectures. They are also preferred for representing crosscutting concerns that can be meaningfully stated in separation, such as logging or security. For example, the fact that the checkout activity requires authorization can be expressed as a security annotation (a kind of join point), leaving the insertion of call actions to the approprite authorization and authentication activities to an aspect weaver.

A few modeling tools on the market support model templates in some form, but usually in an ad hoc manner. For example, templates in Rational XDE are modeled as parameterized collaborations, even though the template can contain meta-code creating arbitrary models, i.e., the template instance is not a collaboration. Obviously, variability can also be realized by direct model manipulation, such as using composition directives [21] or XMI manipulation [22].

Finally, feature models have been previously used together with textual templates as the structure definition of template input, e.g., [23]. However, the application to model templates is, to our knowledge, new.

8 Concluding Remarks

The purpose of this work can be seen from different perspectives: (1) giving semantics to features in feature models by mapping them to other models and (2) using feature models to provide a concise representation of variability contained in other models. Expressing PCs and MEs in terms of features provides traceability between features and their realization in models.

Although we think our approach is particularly useful at the requirements level, it can be applied for models at any level, e.g., architecture and implementation models.

From the usability perspective, the approach is intuitive. Model templates are in the target notation, so there is no need to learn new specialized languages (except for simple feature models) and existing tools can also be reused. Implicit conditions, patching, and simplification minimize annotation effort and decrease visual complexity, which makes model templates more concise. Coloring makes it easy to see what will be contributed by selecting a given feature. Model templates can be created incrementally and simultaneously with the feature model.

During our case study we have observed that the majority of PCs are single features. However the ability to write more complex PCs allows us to avoid polluting the feature model with features related to the implementation details of the template. For example, a “glue” element usually requires a PC being a conjunction of features. If a PC could only be simple features, an additional feature corresponding to the “glue” element would need to be introduced.

A possible concern is that annotation is not always simple and may require few iterations; however, further tool support can be offered, e.g., for filtering model template parts relevant to certain features and subset of systems, and automatic verification guaranteeing the well-formedness of all possible template instances. Those additional capabilities, as well as support for element cloning in model templates, will be covered in future work.

References

1. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston, MA (2000)
2. Czarnecki, K.: Overview of Generative Software Development. In: Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, September, 15–17, 2004, Mont Saint-Michel, France. (2004) http://www.swen.uwaterloo.ca/kczarnec/gsdoverview.pdf.
3. Batory, D.: Feature Models, Grammars, and Propositional Formulas. Technical Report TR-05-14, University of Texas at Austin, Texas (2005)
4. Object Management Group: Meta-Object Facility. (2002) http://www.omg.org/technology/documents/formal/mof.htm.
5. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice **10** (2005) 143–169 http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf.
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice **10** (2005) 7–29
8. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Paper available from http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf. Software available from gp.uwaterloo.ca/fmp.
9. World Wide Web Consortium: XML Path Language (XPath) 2.0. (2005) http://www.w3.org/TR/xpath20/.
10. Object Management Group: Unified Modeling Language 2.0. (2004) http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.zip.
11. Schnieders, A., Puhlmann, F.: Activity diagram inheritance. In Abramowicz, W., ed.: BIS 2005 - Business Information Systems. 8th International Conference, Poznan, Poland, 2005, Proceedings. (2005)
12. Lee, J., Xue, N.L., Kuei, T.L.: A note on state modeling through inheritance. SIGSOFT Softw. Eng. Notes **23** (1998) 104–110
13. A J H Simons, M P Stannett, K.E.B., Holcombe, W.M.L.: Plug and play safely: Rules for behavioural compatibility. In: Proc. 6th IASTED Int. Conf. Software Engineering and Applications. (2002) 263–268
14. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a uml profile for software product lines. In: PFE. (2003) 129–139

15. Wasowski, A.: Automatic generation of program families by model restrictions. In Nord, R.L., ed.: Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings. Volume 3154 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer-Verlag (2004) 73–89
16. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering (ICSE), Portland, Oregon, Los Alamitos, CA, IEEE Computer Society (2003) 187–197
17. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 107–119
18. Philippow, I., Riebisch, M., Boellert, K.: The hyper/UML approach for feature based software design. In Akkawi, F., Aldawud, O., Booch, G., Clarke, S., Gray, J., Harrison, B., Kandé, M., Stein, D., Tarr, P., Zakaria, A., eds.: The 4th AOSD Modeling With UML Workshop. (2003)
19. McNeile, A.T., Simons, N.: State machines as mixins. Journal of Object Technology **2** (2003) 85–101
20. Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams. In: FIW. (2003) 43–58
21. Straw, G., Georg, G., Song, E., Ghosh, S., France, R., Bieman, J.M.: Model composition directives. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J., eds.: UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings. Volume 3273 of LNCS., Springer (2004) 84–97
22. Jarzabek, S., Zhang, H.: Xml-based method and tool for handling variant requirements in domain models. In: RE. (2001) 166–173
23. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. Volume 2487 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer-Verlag (2002) 156–172

Developing Dynamic and Adaptable Applications with CAM/DAOP: A Virtual Office Application *

(Tool Demonstration)

Mónica Pinto, Daniel Jiménez, and Lidia Fuentes

Dpto. de Lenguajes y Ciencias de la Computación. University of Málaga, Spain
{pinto, priego, lff}@lcc.uma.es

Abstract. CAM/DAOP is a component and aspect based model and platform implemented using Java/RMI and reflective techniques. Using CAM/DAOP we have developed several collaborative applications, where the most relevant one is a Virtual Office application, which allows dispersed users to collaborate as if they were co-located. Attendees of the demonstration will see how to develop dynamic and adaptable applications with CAM/DAOP, from the design through to the implementation phases. We will place emphasis on showing how to adapt the behavior of CAM/DAOP applications at runtime, simply by modifying the architectural information provided during the application development.

1 Introduction

CAM/DAOP [1] is a component and aspect based model and platform. CAM is a new model for designing component and aspect based applications. The CAM model defines the main entities of a CAM application and the relationships among them. The DAOP platform is a distributed component and aspect platform that implements the CAM model and provides a dynamic weaving mechanism that plugs software aspects into components at runtime.

Another relevant feature of our approach is the use of DAOP-ADL, an XML-based architecture description language that is used to describe the structure of CAM applications in terms of a set of components, aspects and composition constraints. The most relevant issue is that this description can be automatically generated from the UML-based CAM diagrams generated during design. Moreover, the information generated with DAOP-ADL is loaded onto the internal structures of the DAOP platform when the application is initiated and it is used at runtime by the DAOP platform to perform the dynamic composition of components and aspects.

The development of CAM/DAOP applications is supported by an integrated development process and environment that help software developers to understand how to use aspect-oriented technologies, specifically CAM/DAOP. This

* This work is supported in part by Spanish MCYT Project TIC2002-04309-C02-02 and in part by European Commission grant IST-2-004349 (European NoE on AOSD)

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 438–441, 2005.

process establishes the relationship among CAM, DAOP-ADL and DAOP and guides software developers through the construction of CAM/DAOP applications from the design to the execution phases. The IDE will automatically produce the information generated in each phase of the process; in the case were the automation were possible.

Due both to the use of the DAOP-ADL language and to the dynamic composition mechanism offered by the DAOP platform, our approach is especially suitable for developing applications with high requirements of dynamic adaptability. Up to now we have developed several collaborative applications using CAM/DAOP; from simple applications such as chat applications or the Pictionary game, to more complex ones such as a virtual office application[1]. These applications are dynamic and distributed applications that are characterized by high runtime interaction among the users connected in the same session, and have high requirements of dynamic adaptability, so they are good candidates for the testing of our approach.

The primary aim of this demonstration is to show how to develop dynamic and adaptable component and aspect based applications with CAM/DAOP. The demo will go through the different phases involved in the development of the virtual office application: (1) the application design using the CAM model; (2) the description of the application architecture using the DAOP-ADL language; (3) the reuse of pre-implemented components and aspects; (4) the execution of the application by the DAOP platform, and (5) the possibility of dynamically adapting the behavior of the application at runtime by modifying the application architecture described in the DAOP-ADL language.

2 The CAM/DAOP Approach

The main features provided by CAM/DAOP are the following: (1) it is possible to separate any extra-functional property from components, modelling them as aspects; (2) the behaviour of aspects and the pointcuts are described in different and independent entities. Therefore, aspects become context-independent and reusable parts; (3) the description of the pointcuts for all the aspects in an application is centralized as part of the composition rules in DAOP-ADL. This makes it easy to trace which aspects are applied to each component and when they are applied - specially at design and architectural levels, and (4) it closes the 'gap' between design and implementation using the DAOP-ADL language. The execution of the application is driven by this information, which was provided with the CAM model during the design phase.

The current implementation of CAM/DAOP uses Java to take advantage of characteristics such as reflective programming, code mobility and RMI. A CAM/DAOP application is a web application that is deployed by making both the implementation of components and aspects and the description of the software architecture accessible through a Web server. This information will be then downloaded at runtime using an applet or the Java WebStart technology.

[1] http://caosd.lcc.uma.es/CAOSDGroup/VirtualOffice.htm

CAM/DAOP is related to other research efforts such as JAC [2], JAsCo [3] or JBoss AOP [4] among others, all of them offering dynamic weaving mechanisms at load time or at runtime. In addition to the weaving mechanism, another similarity with JAsCo and AOP/JBoss is the application of the separation of aspects to components, instead of objects as happens in most approaches. Furthermore, AOP/JBoss share with CAM/DAOP the description of the aspect point cuts in XML, externally and completely independently from the implementation of aspects. Similar to AspectJ [5] and JAC, CAM/DAOP offers a set of tools to help software developers to construct aspect-oriented applications.

3 The CAM/DAOP Development Process

The most unique feature of our approach is the use of the DAOP-ADL language to explicitly describe the architecture of a CAM/DAOP application in XML, and the fact that the use of the DAOP-ADL language establishes a clear "relationship" among the different phases of the software development (from design to execution). In fact, the DAOP-ADL language is the essence of the complete development process, as described below (see figure 1):

1. First, software developers will use a UML editor to draw the UML diagrams of the CAM/DAOP application (phase 1). These diagrams will be used by our IDE to automatically generate part of the software architecture of the application in DAOP-ADL (phase 2).
2. Once the software architect completes the description of the application architecture in DAOP-ADL, the resulting XML document is stored in an architecture repository (phase 2).
3. Then, during the implementation phase software developers will register their components and aspects in a component and aspect repository tool (phase 3). The relevant issue is that this tool automatically generates the description of these entities in DAOP-ADL. These XML descriptions are then automatically incorporated as part of the architecture description when components and aspects are selected. Components, aspects and the architectural XML document are deployed in a web server (phase 4).
4. Finally, as mentioned before, the information in DAOP-ADL is loaded along with components and aspects and it is consulted by the DAOP platform at runtime to perform the late-binding between components and aspects. Furthermore, this information can be modified at runtime to adapt the behaviour of applications developed on top of CAM/DAOP (phase 5).

4 The Virtual Office Application

A virtual office application is a virtual space where users join to collaborate. The space is organized in rooms, which contain all the resources needed to collaborate, such as documents and collaborative tools. Users navigate through the rooms collaborating with other users they meet in the same room. In order to do that,

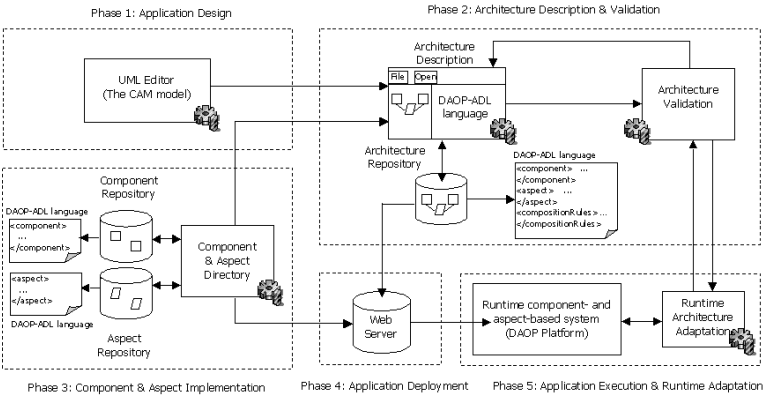


Fig. 1. CAM/DAOP Development Process and IDE

it is important that users have awareness information about the location and the state of all the resources in the environment (other users, tools, documents, etc.). There are many aspects that are worthy of being separated in a virtual office application. Some examples are: the authentication aspect, which checks whether the user is registered in the system; the persistence aspect, which stores and restores the distributed state of the virtual office; the awareness aspect that notifies changes in the state of components, and the access control aspect that checks which users have rights to access the environment resources.

5 Conclusions

After attending this demo those people interested in the construction of adaptable and extensible applications will have seen how the use of an aspect-oriented approach that offers a dynamic weaving mechanism can help to develop this kind of applications. Specifically, we will have shown our experience using CAM/DAOP, our own component and aspect based dynamic platform.

References

1. Pinto, M., Fuentes, L., Troya, J.M.: A Dynamic Component and Aspect Oriented Platform. The Computer Journal. Next Publication.
2. Pawlack, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible and efficient framework for AOP in java. In: Proc. of Reflection'01, Springer-Verlag
3. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: An aspect- oriented approach tailored for component based software development. In: Proc. of AOSD'03, Boston
4. The Aspect Oriented Programming and JBoss Tutorial. (2003) http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html.
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proc. of ECOOP'01, Budapest, Hungary, 327–355

Metamodeling Made Easy – MetaEdit+
(Tool Demonstration)

Risto Pohjonen

MetaCase, Ylistönmäentie 31, 40500 Jyväskylä, Finland
rise@metacase.com

Abstract. Many current metamodeling environments still require manual programming to build full tool support for the modeling language, especially for language constraints, representational elements and graphical editing tools. Because of this, a considerable part of development resources has to be reserved for secondary assets of the final environment instead of its main vehicle, the modeling language itself. In this demonstration, we present the MetaEdit+ metaCASE tool, and show how metamodeling and tool support for domain-specific modeling languages can be completed without programming. We will describe the metamodeling tool set of MetaEdit+ and explain how conceptual and representational metamodeling is carried out with it. Finally, we will look at the executable modeling environment derived from the metamodel.

1 Introduction

Many contemporary metamodeling environments still require a considerable amount of manual programming to build the modeling language definition and its tool support [1]. This can be explained as a way to offer complete control over the resulting modeling and code generation environment, but it also becomes a problem for their developers. Typically, such features as language constraints, representational elements and graphical editing tools are left to be coded by hand. This requires that a considerable amount of time and other resources be allocated to complete something that is secondary to the main objective: the development and integration of the modeling language and its code generation.

MetaEdit+ is a metaCASE tool that provides an alternative solution for building complete tool support for domain-specific modeling (DSM) languages [4]. The tool implements the generic CASE tool functionality that can be configured into a complete and working CASE environment with a metamodel. Metamodeling is carried out with a tool set that allows definition of the metamodel and its representational elements, rules and constraints. Based on these definitions, MetaEdit+ provides the standard set of CASE tool functionality, including graphical editors, design data management, and integration with other tools via its API. The advantage of this approach over manual programming is the easier and faster metamodeling and deployment process – it is possible to get working DSM environment within hours instead of days and weeks, as is the case with tools that require additional manual programming.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 442 – 446, 2005.

2 Metamodeling with MetaEdit+

Metamodeling in MetaEdit+ is based on the GOPPRR metamodeling language [2, 4]. GOPPRR is an acronym formed from language's base types which are Graph, Object, Port, Property, Relationship and Role. Graph is the top-level structure of the metamodel. It defines one language or diagram technique such as Class Diagram or State Transition Diagram. The actual semantics of the graph are defined as the bindings of objects, relationships, roles and ports within the graph. Properties are characterizing attributes that can be attached to each of these other types.

For each of these base types, MetaEdit+ provides an editing tool for creating and modifying new types based on the base type (examples of such tools are shown in Figure 1). A new type is created by sub-typing it from the relevant base type, defining its name and defining a set of properties that are attached to this type. It is also worthwhile to note that this new type can be sub-typed further and it is therefore possible to build inheritance hierarchies of types in an object-oriented fashion.

The created types are the fragments of the metamodel from which the complete language is built. To specify the rules of how they can be used together, tools like the Binding Tool – as shown in Figure 2 – are used. This describes how the objects within a graph can be connected together via relationships, roles and ports, and how often each kind of connection may occur.

Fig. 1. The Object and Property type definition tools of MetaEdit+

Fig. 2. The binding and constraint definitions in MetaEdit+

Bindings are semantically powerful metamodeling constructs, sufficient for defining many structural rules of the modeling language. There are, however, other kinds of rules that refine and constrain the behavior and the use of the language. For example, for a State Transition Diagram one could set a constraint that states that "each start state may have only one From role that leaves it". For defining this kind of rules, MetaEdit+ provides a Constraints Definer tool (also illustrated in Figure 2). As with the Binding Tool, the rules and constraints defined with this tool are enforced at run-time, ensuring the correctness of the models.

In addition to such conceptual parts as types, bindings and rules, a complete metamodel also requires the definition of representational counterparts of these conceptual elements. MetaEdit+ provides a Symbol Editor tool for drawing such graphical symbols for objects, relationships and roles. Properties appear as part of these symbols and ports are tackled as part of object symbol definitions. An example of a symbol definition in Symbol Editor is shown in Figure 3.

The conceptual and representational parts of the metamodel are used by MetaEdit+ to configure the modeling part of the tool, making typical CASE tool functionality immediately available for the user. An example of one such CASE tool feature, a Diagram Editor, is shown in Figure 4.

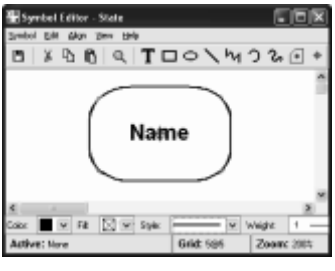


Fig. 3. Symbol definition for object "State" in the Symbol Editor of MetaEdit+

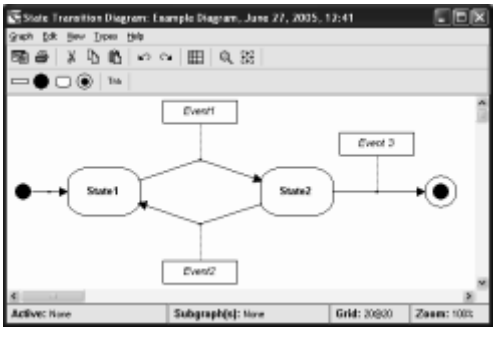


Fig. 4. Diagram Editor of MetaEdit+

As a technical note, MetaEdit+ is based on an implementation of the GOPPRR metamodeling language. Written in Smalltalk, it provides useful flexibility for the metamodeller. This not only enables easy sub-typing of the base types and their properties, but also unlimited reuse and linking of types. The metamodeling state is always "live", i.e. the tool will automatically propagate changes in the metamodels into the models.

3 Example of Extending the Environment: Code Generation

In the previous section we discussed the issue of how to obtain tool support for a DSM language without manual programming. One of the key benefits cited for the DSM approach is the promise of 100% code generation when using domain-specific code generators. Domain-specific code generation, on the other hand, almost always require a specifically written code generator that is tailored for the given domain and target platform. While it is not possible to avoid manual programming in this case, MetaEdit+ provides a means to minimize the effort with its reporting and scripting language (shown in Figure 5). The language is specifically designed for the tasks of accessing, navigating and extracting information from the design data, and turning that information into text: program code, documentation or checking reports. The advantage here lies with the close integration of the language into MetaEdit+, and the reporting tool that saves a lot of routine effort for the developer of the code generator.

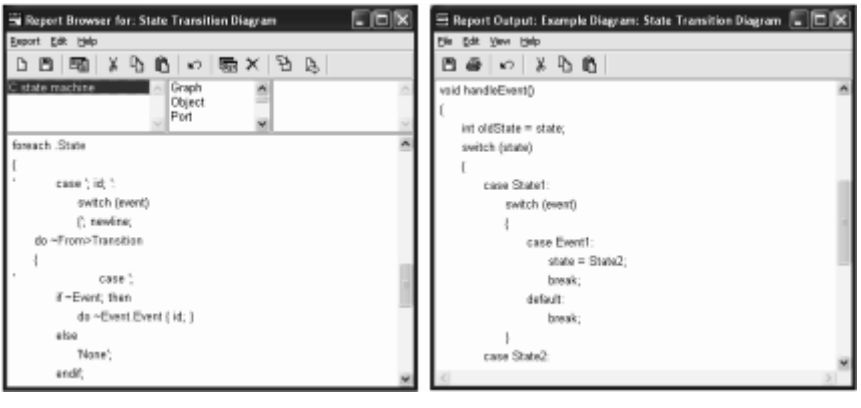


Fig. 5. Code generator definition and generation output

4 Conclusion

In this demonstration we have presented an approach for building modeling and code generation tool support for DSM languages by configuring MetaEdit+ metaCASE environment with metamodels. The main benefit of this approach – that has been validated by several industry cases of developing and adopting the DSM with MetaEdit+ [3, 5] – is the fast and flexible metamodeling process. It not only sets the

metamodeler free to concentrate on the modeling language and code generator instead of secondary implementation issues, but also enables him or her to quickly prototype and test the language definition in a live environment. The same flexibility also applies to the maintenance of the deployed DSM environment: changes in metamodel and code generator propagate into the production environment and models automatically.

An easy metamodeling experience has always been one of the main objectives of the development of MetaEdit+ and this will be the trend for the future development efforts as well. As for the forthcoming versions of MetaEdit+, we are about to provide means for graphical metamodeling and integration with other metamodeling languages and environments. There will be also such developments on the conceptual and representational metamodel elements as more functional symbol elements, dynamic ports and extensions in reporting and scripting language. It is also expected that the growing interest and awareness of metamodeling and DSM will raise new kinds of requirements for the tool vendors to consider.

References

1. Steven Kelly. Tools for Domain-Specific Modeling – Generating Code from Models. Dr Dobb's journal, September 2004.
2. Steven Kelly, Kalle Lyytinen and Matti Rossi. MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Y. Vassiliou and J. Mylopoulos (editors) *Proceedings of the 8th International Conference on Advanced Information Systems Engineering*, Springer Verlag, 1996.
3. Janne Luoma, Steven Kelly and Juha-Pekka Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In Juha-Pekka Tolvanen, Jonathan Sprinkle and Matti Rossi (editors) *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04)*, Computer Science and Information System Reports, Technical Reports, TR-33, University of Jyväskylä, Finland 2004.
4. MetaEdit+ User's Guide. MetaCase, Jyväskylä, Finland 2004
5. MetaCase web at *http://www.metacase.com*

Author Index