

Walid Mohamed Taha (Ed.)

LNCS 5658

# Domain-Specific Languages

IFIP TC 2 Working Conference, DSL 2009  
Oxford, UK, July 2009  
Proceedings



ifip



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Walid Mohamed Taha (Ed.)

# Domain-Specific Languages

IFIP TC 2 Working Conference, DSL 2009  
Oxford, UK, July 15-17, 2009  
Proceedings



Springer

Volume Editor

Walid Mohamed Taha  
Rice University  
Department of Computer Science  
6100 South Main Street  
Houston, TX 77005, USA  
E-mail: [taha@rice.edu](mailto:taha@rice.edu)

Library of Congress Control Number: 2009929831

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743

ISBN-10 3-642-03033-5 Springer Berlin Heidelberg New York

ISBN-13 978-3-642-03033-8 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

[springer.com](http://springer.com)

© IFIP International Federation for Information Processing 2009

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper SPIN: 12716142 06/3180 5 4 3 2 1 0

# Preface

Dijkstra once wrote that computer science is no more about computers than astronomy is about telescopes. Despite the many incredible advances in computer science from times that predate practical mechanical computing, there is still a myriad of fundamental questions in understanding the interface between computers and the rest of the world. Why is it still hard to mechanize many tasks that seem to be fundamentally routine, even as we see ever-increasing capacity for raw mechanical computing? The disciplined study of domain-specific languages (DSLs) is an emerging area in computer science, and is one which has the potential to revolutionize the field, and bring us closer to answering this question. DSLs are formalisms that have four general characteristics.

- They relate to a well-defined domain of discourse, be it controlling traffic lights or space ships.
- They have well-defined notation, such as the ones that exist for prescribing music, dance routines, or strategy in a football game.
- The informal or intuitive meaning of the notation is clear. This can easily be overlooked, especially since intuitive meaning can be expressed by many different notations that may be received very differently by users.
- The formal meaning is clear and mechanizable, as is, hopefully, the case for the instructions we give to our bank or to a merchant online.

While much attention has been given to finding the one “best” general programming language, DSLs such as Microsoft Excel and MathWorks MATLAB grew to have a much larger user base than many traditional programming languages. Much work needs to be done to train and equip future generations of scientists and engineers to recognize the need for such DSLs and to realize their potential.

This volume presents the proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages (DSL 2009). The proceedings comprise 18 peer-reviewed publications. These papers were selected by the Program Committee from a total of 48 submissions. Most papers received at least four reviews, and two received three. First, reviewers read the full-length submissions and entered their reviews into the online system. Then, all reviewers read all reviews, and points of disagreement were discussed until a consensus was reached. The topics covered by the papers presented at the conference represent the diversity of applications, challenges, and methods of DSLs. We are confident that the advances embodied in these papers will provide further impetus to accelerating development in this field.

*Dedicated to the memory of Peter Landin, 1930-2009*

# Organization

IFIP WC DSL is a joint activity of the Technical Committee on Software: Theory and Practice (TC 2), and in particular three IFIP Working Groups (WGs):

- WG 2.1 Algorithmic Languages and Calculi
- WG 2.8 Functional Programming
- WG 2.11 Program Generation

## Executive Committee

- |                   |   |
|-------------------|---|
| General Chair     | Jeremy Gibbons (University of Oxford, UK) |
| Program Committee |   |
| Chair             | Walid Mohamed Taha (Rice University, USA) |
| Publicity Chair   | Emir Pasalic (LogicBlox, USA)             |

## Program Committee

- |                    |  |
|--------------------|--|
| Jon Bentley        | Avaya Labs, USA                          |
| Martin Erwig       | Oregon State University, USA             |
| Jeff Gray          | University of Alabama at Birmingham, USA |
| Robert Grimm       | New York University, USA                 |
| Jim Grundy         | Intel Strategic CAD Labs, USA            |
| Thomas Henzinger   | EPFL, Switzerland                        |
| Samuel Kamin       | UIUC, USA                                |
| Richard Kieburtz   | Portland State University, USA           |
| Ralf Lämmel        | University of Koblenz, Germany           |
| Julia Lawall       | University of Copenhagen, Denmark        |
| Benjamin Pierce    | University of Pennsylvania, USA          |
| Vivek Sarkar       | Rice University, USA                     |
| Jeremy Siek        | University of Colorado at Boulder, USA   |
| José Nuno Oliveira | University of Minho, Portugal            |
| Doaitse Swierstra  | Utrecht University, The Netherlands      |
| Walid Mohamed Taha | Rice University, USA (Chair)             |
| Eelco Visser       | Delft University, The Netherlands        |
| William Waite      | University of Colorado at Boulder, USA   |
| Stephanie Weirich  | University of Pennsylvania, USA          |

## External Reviewers

Paulo Sérgio Almeida	Danny Groenewegen	Paulo Silva
Jose Almeida	Zef Hemel	Paul Strauss
Roland Backhouse	Johan Jeuring	Yu Sun
Tim Bauer	Maartje de Jonge	Wouter Swierstra
Dan Bentley	Lennart Kats	Robert Tairas
Alcino Cunha	Nicholas Kraft	Tarmo Uustalu
Zekai Demirezen	Andres Loeh	Sander Vermolen
Trung Thangh	Marjan Mernik	Joost Visser
Dinh-Trong	John O'Leary	Eric Walkingshaw
Jake Donham	Nicolas Palix	David Weiss
Jeroen Fokker	Jorge Sousa Pinto	
Aniruddha Gokhale	Joao Saraiva	

# Table of Contents

## Semantics

J Is for JavaScript: A Direct-Style Correspondence between Algol-Like Languages and JavaScript Using First-Class Continuations . . . . .	1
<i>Olivier Danvy, Chung-chieh Shan, and Ian Zerny</i>	

Model-Driven Engineering from Modular Monadic Semantics: Implementation Techniques Targeting Hardware and Software . . . . .	20
<i>William L. Harrison, Adam M. Procter, Jason Agron, Garrin Kimmell, and Gerard Allwein</i>	

## Methods and Tools

A MuDDy Experience–ML Bindings to a BDD Library . . . . .	45
<i>Ken Friis Larsen</i>	

Gel: A Generic Extensible Language . . . . .	58
<i>Jose Falcon and William R. Cook</i>	

A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications . . . . .	78
<i>Zoé Drey, Julien Mercadal, and Charles Consel</i>	

LEESA: Embedding Strategic and XPath-Like Object Structure Traversals in C++ . . . . .	100
<i>Sumant Tambe and Aniruddha Gokhale</i>	

Unit Testing for Domain-Specific Languages . . . . .	125
<i>Hui Wu, Jeff Gray, and Marjan Mernik</i>	

Combining DSLs and Ontologies Using Metamodel Integration . . . . .	148
<i>Tobias Walter and Jürgen Ebert</i>	

## Case Studies

A Domain Specific Language for Composable Memory Transactions in Java . . . . .	170
<i>André Rauber Du Bois and Marcos Echevarria</i>	

CLOPS: A DSL for Command Line Options . . . . .	187
<i>Mikoláš Janota, Fintan Fairmichael, Viliam Holub, Radu Grigore, Julien Charles, Dermot Cochran, and Joseph R. Kiniry</i>	

Nettle: A Language for Configuring Routing Networks . . . . .	211
<i>Andreas Voellmy and Paul Hudak</i>	
Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell: A Domain-Specific Library for Computational Vulnerability Assessment . . . . .	236
<i>Daniel Lincke, Patrik Jansson, Marcin Zalewski, and Cezar Ionescu</i>	
Domain-Specific Language for HW/SW Co-design for FPGAs . . . . .	262
<i>Jason Agron</i>	
A Haskell Hosted DSL for Writing Transformation Systems . . . . .	285
<i>Andy Gill</i>	
Varying Domain Representations in Hagl: Extending the Expressiveness of a DSL for Experimental Game Theory . . . . .	310
<i>Eric Walkingshaw and Martin Erwig</i>	
A DSL for Explaining Probabilistic Reasoning . . . . .	335
<i>Martin Erwig and Eric Walkingshaw</i>	
Embedded Probabilistic Programming . . . . .	360
<i>Oleg Kiselyov and Chung-chieh Shan</i>	
Operator Language: A Program Generation Framework for Fast Kernels . . . . .	385
<i>Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel</i>	
<b>Author Index . . . . .</b>	<b>411</b>

# J Is for JavaScript: A Direct-Style Correspondence between Algol-Like Languages and JavaScript Using First-Class Continuations

Olivier Danvy<sup>1</sup>, Chung-chieh Shan<sup>2</sup>, and Ian Zerny<sup>1</sup>

<sup>1</sup> Department of Computer Science, Aarhus University  
Aabogade 34, DK-8200 Aarhus N, Denmark  
`{danvy,zerny}@cs.au.dk`

<sup>2</sup> Department of Computer Science, Rutgers University  
110 Frelinghuysen Road, Piscataway, NJ 08854, USA  
`ccshan@rutgers.edu`

**Abstract.** It is a time-honored fashion to implement a domain-specific language (DSL) by translation to a general-purpose language. Such an implementation is more portable, but an unidiomatic translation jeopardizes performance because, in practice, language implementations favor the common cases. This tension arises especially when the domain calls for complex control structures. We illustrate this tension by revisiting Landin’s original correspondence between Algol and Church’s lambda-notation.

We translate domain-specific programs with lexically scoped jumps to JavaScript. Our translation produces the same block structure and binding structure as in the source program, à la Abdali. The target code uses a control operator in direct style, à la Landin. In fact, the control operator used is almost Landin’s J—hence our title. Our translation thus complements a continuation-passing translation à la Steele. These two extreme translations require JavaScript implementations to cater either for first-class continuations, as Rhino does, or for proper tail recursion. Less extreme translations should emit more idiomatic control-flow instructions such as `for`, `break`, and `throw`.

The present experiment leads us to conclude that translations should preserve not just the data structures and the block structure of a source program, but also its control structure. We thus identify a new class of use cases for control structures in JavaScript, namely the idiomatic translation of control structures from DSLs.

## 1 Introduction

It has long been routine to define a programming language by writing an interpreter in or a compiler towards a pre-existing language [27, 28, 32, 39, 40, 41]. This tradition began with John McCarthy [34], who argued that it is not a circular argument but a valid pedagogical device to use a pre-existing language as a

notation for expressing computation. McCarthy drew an analogy between translations from one programming language to another and Tarski’s efforts to define one mathematical logic in terms of another, “which have proved so successful and fruitful” [34, page 7] even though they still draw criticism today [19].

Translations between programming languages have also been used to reason about expressiveness. For example, Böhm and Jacopini used a compiling argument to show that flowcharts and expression languages with recursion have the same expressive power [7], and Fischer invented the CPS transformation to show the equivalence between the deletion strategy and the retention strategy to implement activation records [16]. Finally, this style of formal specification is also useful for building interpreters and compilers in practice. Indeed, it has given rise to industrial-strength software development [6].

The translation approach is alive and well today, whether the defined (source) language is considered to be domain-specific or general-purpose, and whether the defining (target) language is a variant of the lambda calculus, C, or some other language. This stream of successes is especially remarkable given that differences between the defined and defining languages often force the translation to be quite ingenious, albeit not entirely perspicuous.

**Pervasive ingenuity:** Some translations impose global changes on programs.

For example, when the defined language features a side effect that the defining language does not support, a definitional interpreter must encode the effect throughout the translation using, e.g., state-passing or continuation-passing style [41]. These styles have since been crisply factored out into *computational monads* [38, 49].

**Homomorphic ingenuity:** Some translations manage to relate structural elements between source and target programs. For example, McCarthy translated flowcharts to mutually recursive equations by mapping each program point to a recursive equation and each variable to a parameter in a recursive equation [33]. These equations were tail-recursive, a property that Mazurkiewicz then used for proving properties about flowcharts [31]. Landin translated Algol programs to applicative expressions by mapping block structure to subexpression structure, assignments to state effects, and jumps to control effects [27]. Working homomorphically with a smaller defining language, namely the lambda calculus, Abdali mapped local functions into global, lambda-lifted recursive equations in a storeless fashion [1, 3], but invented continuation-passing style en passant to implement jumps [2].

**Seemingly no ingenuity:** Even when the translation turns out to be just an identity function, careful design may be required. For example, it is desirable for a partial evaluator to be *Jones optimal*, which means that specializing a self-interpreter to a program yields the same program modulo renaming [21]. Achieving Jones optimality requires much of the partial evaluator (e.g., it must resolve scoping references statically), but also of the interpreter: for one thing, the interpreter must be in direct style; indeed, specializing a continuation-passing self-interpreter with a Jones-optimal partial evaluator yields programs in continuation-passing style. To take another example, even

though Pascal can be regarded as a subset of Scheme and hence easily translated to Scheme by (essentially) the identity function, such a translation calls for a Scheme compiler, such as Orbit [25], that is designed so that Pascal-like programs (i.e., programs with only downwards funargs<sup>1</sup> and downwards contargs) would be compiled as efficiently as by a Pascal compiler [26].

**Shoehorning:** Not all properties of the defined language are naturally supported by the defining language and its run-time system. Perhaps the best-known example is proper tail-recursion and its implementation by trampolining [4, 18, 37, 43, 47]. A close second is first-class continuations [30, 36, 44].

For humans and machines to work with a translation more easily, it is our belief that homomorphic ingenuity and seemingly no ingenuity are preferable over pervasive ingenuity and shoehorning. In short, we believe that a translation should take advantage of the expressive power of the defining language in an *idiomatic* way, if only for implementations of the defining language to execute target programs more efficiently. As shown by the examples above, this goal of idiomaticity often calls for the defined and defining languages to be carefully designed and reconciled. In other words, the principle of idiomaticity guides not just language implementation but also language design, especially today as the proverbial 700 DSLs blossom on new platforms such as browsers running JavaScript.

This article establishes an idiomatic translation between a specific and illustrative pair of languages.

- Our defined language is an Algol-like block-structured language with lexically scoped jumps. It is illustrative not just because most DSLs have block structure and lexical scope, but also because many DSLs feature complex control constructs motivated by their domains such as pattern matching and logical inference.
- Our defining language is JavaScript with first-class continuation objects, as implemented in Rhino [10]. It is illustrative not just because many DSLs are implemented by translation to JavaScript, but also because it lets us exhibit an extremely uniform and frugal translation among more complex alternatives that use a greater variety of control-flow instructions.

Taking advantage of the close correspondence between Rhino’s continuation objects and Landin’s J operator [14], we thus revive Landin’s original correspondence between Algol programs and applicative expressions [27, 28].

---

<sup>1</sup> ‘Funarg’ is an abbreviation for ‘functional argument’ and by analogy, ‘contarg’ stands for ‘continuation argument’. They refer to the ability of passing functions as arguments (downwards funargs) and returning functions as results (upwards funargs). The ‘downward’ and ‘upward’ orientation stems from implementing function calls with a current control stack: downwards is toward what has been pushed (and is still there) and upwards is toward what has been popped (and thus may be gone [16]).

## 2 Continuation Objects in JavaScript

This section introduces continuation objects in the JavaScript implementation Rhino, compares them to Landin’s J operator, and explains how we use them to express jumps in a block-structured language.

In Rhino, a continuation object can be created by evaluating the expression `new Continuation()`. The resulting object represents the continuation of *the call to* the function that evaluated this expression. That is, invoking a continuation object (as if it were a function) returns from the function that created it. For example, the following program only prints 2.

```
function foo () {
    var JI = new Continuation();
    JI(2);
    print(1);
}
print(foo());
```

The continuation is undelimited and remains valid throughout the rest of the program’s execution. For example, the following program prints 1 followed by an unbounded series of 2’s.

```
var JI;
function foo () {
    JI = new Continuation();
    return 1;
}
print(foo());
JI(2);
```

### 2.1 Landin’s Translation of Jumps Using J

We name the captured continuations above `JI` because creating a continuation object in Rhino is equivalent to invoking Landin’s J operator on the identity function [14]. The J operator is the first control operator to have graced expression-oriented programming languages. Landin invented it specifically to translate Algol jumps to applicative expressions in direct style [27, 29]. If JavaScript featured the J operator, then Landin would have translated the loop

```
i := 1000000;
loop: i := i - 1;
      if i > 0 then goto loop;
```

to the following JavaScript program.

```
var i;
var loop = J(function () {
    i = i - 1;
    if (i > 0) loop();
});
i = 1000000;
loop();
```

The application of `J` above creates a function `loop` that, when invoked, evaluates the function body `i = i - 1; if (i > 0) loop();` with the continuation of the call to the program. In other words, `loop` denotes a “state appender” [8] and the invocation `loop()` jumps into the function body in such a fashion that the function body directly returns to the caller of the translated program. We thus express a jump to `loop` as `loop()`. This program contains two jumps to `loop`, an implicit fall-through and an explicit `goto`, so the JavaScript code above contains two occurrences of `loop()`.

The expression `new Continuation()` in Rhino is equivalent to the expression `J(function (x) { return x; })` in JavaScript with the `J` operator. Conversely, Landin and Thielecke [48] noted that `J` can be expressed in terms of `JI` as

$$J = (\lambda c. \lambda f. \lambda x. c(fx)) JI.$$

Following Landin’s translation strategy, then, one might define

```
function compose (c,f) { return function (x) { c(f(x)); } }
```

in JavaScript and then translate the loop above as follows.

```
var i;
var loop = compose(new Continuation(), function () {
    i = i - 1;
    if (i > 0) loop();
});
i = 1000000;
loop();
```

(Because we translate each label to a function that takes no argument, the two occurrences of `x` in the definition of `compose` can be omitted.) This translation, like Landin’s, is attractive in that it idiomatically preserves the block and binding structure of the source program: the main program block translates to the main program block, a sequence of commands translates to a sequence of commands, and the variable `i` and the label `loop` translate to the variables `i` and `loop`.

Unfortunately, although this last translation runs in Rhino, it overflows the stack. The reason is that 1000000 calls to `c` pile up on the stack and are not discarded until the function body returns for the first time. Although we can express `J` in terms of `JI`, it is unclear how to do so without this space leak.

## 2.2 Our Translation of Jumps Using `JI`

In order to translate jumps while preserving the stack-space usage behavior of programs, we modify Landin’s translation slightly: every captured continuation shall accept a thunk and invoke it immediately [15]. In other words, every function call shall expect a thunk to be returned and invoke it immediately.

Because we cannot change the continuation with which Rhino invokes the main program, this modification requires that we wrap up the main program in a function `main`, which returns a thunk that should be invoked immediately. Below is our final translation of the loop, which completes without overflowing the stack in Rhino.

```

var main = function () {
  var JI = new Continuation();
  var i;
  var loop = function () { JI(function () {
    i = i - 1;
    if (i > 0) loop();
  });
  return function () {
    i = 1000000;
    loop();
  };
};
main();

```

This use of thunks is reminiscent of trampolining, but our technique using `JI` does not require the caller of a function to invoke thunks repeatedly until a final result is reached. Rather, the continuation of `main()` accepts a thunk and invokes it exactly once. If another thunk needs to be invoked, such as `function () { i = i - 1; if (i > 0) loop(); }` in this example, the same continuation needs to be invoked again. In other words, the function `main` in our target program returns exactly once more than the number of times a jump occurs in the source program.

### 2.3 Other Uses of J

Outside of its inventor and his student [8], J was first used by Rod Burstall to traverse a search tree in direct style and breadth first, using a queue of first-class continuations [9]. We have re-expressed Burstall’s breadth-first tree traversal in Rhino. With its queue of first-class continuations, this program falls out of the range of our translator from pidgin Algol to JavaScript.

## 3 Source and Target Languages

The key property of our direct-style translation is that it is homomorphic and thus preserves idioms: declarations translate to declarations, blocks to blocks, commands to commands, function calls to function calls, and so forth. Since the translation is homomorphic, we dispense with it altogether in this presentation and simply consider the restricted language obtained as the image of the translation. We thus present the grammar of JavaScript programs in the image of the translation. Figure 1 displays the productions of special interest to our translation, accounting for the essential features of Algol we wish translated to JavaScript. For completeness we include the full grammar in Appendix A. This full grammar accounts for all of the example programs (see Section 4).

- A program is a sequence of declarations and commands. Such a program is translated to a top-level function in the same way procedures are, as shown by the `<program>` production. This translation is required to support labels and jumps at the top level of a program, as illustrated in Section 2.2.

---

```

<program> ::= var main = function () {
    var JI = new Continuation();
    <dcl>*
    return function () { <cmd>* };
};

main();
<dcl-var> ::= var <ident>;
<dcl-proc> ::= var <ident> = function ( <formals>? ) {
    var JI = new Continuation();
    <dcl>*
    return function () { <cmd>* };
};

<dcl-label> ::= var <ident> = function () { JI(function () { <cmd>* }); };
<cmd-goto> ::= <ident>();
<cmd-yield> ::= (function () {<ident>=new Continuation(); <cmd-goto>})();
<exp-call> ::= <ident>( <args>? )()

```

---

**Fig. 1.** Essential grammar of JavaScript in the image of the translation

- The **<dcl-var>** production shows that each variable declaration, is translated directly to a variable declaration in JavaScript. All variables are implicitly initialized to  $\perp$ , i.e., *undefined* in JavaScript.
- The **<dcl-proc>** production shows that a procedure declaration is translated to a JavaScript function that accepts the same formals and returns a thunk containing the procedure commands. In a scope visible to the thunk of commands, nested declarations are defined and the return continuation of the function closure is captured and bound to **JI**. All declarations are mutually recursive so that declarations can refer to each other as well as **JI**.
- A label consists of a name and a sequence of commands. For simplicity we assume that explicit jumps have been inserted in the program wherever execution may flow from one label to another. The **<dcl-label>** production shows that a label definition is translated to a JavaScript function of no arguments, whose body applies the return continuation **JI** to the thunk of label commands. Thus, invoking this function returns the thunk from the currently running procedure.
- The **<cmd-goto>** production shows that goto commands are translated to ordinary function calls with no arguments in JavaScript.
- The **<cmd-yield>** production shows how a *yield* command consists of rebinding the caller’s label with the current continuation, followed by a goto command to transfer control. The surrounding function declaration is required to capture the current continuation. The translation must supply both the *from* and *to* labels. The labels are required to be declared in the exact same block. The *yield* command, however, may appear elsewhere.
- The **<exp-call>** production shows that a procedure call is translated to a JavaScript function call with the same arguments, and the result of the

application, a thunk of procedure commands, is forced to obtain the actual result of the procedure call.

All remaining productions, found in Appendix A, show the straightforward and idiomatic translation to JavaScript.

For simplicity we consider only characters in the ASCII character set with some restrictions on their use. All identifiers are required to be alpha-numeric; the set of label identifiers must not overlap with that of variable and procedure identifiers; and `J1` is a reserved keyword and must not be used as an identifier. To declare named functions, we use variable binding and function expressions in the form of `"var <ident> = function ..."`. This is necessary as function declarations can, in our translation, appear in any block statement, whereas in JavaScript function declarations are only valid at the top-level or directly inside a function body.

## 4 A Representative Collection of Program Samples

The goal of this section is to illustrate with classical examples the direct-style correspondence between pidgin Algol with lexically scoped jumps and JavaScript with continuation objects. We consider in turn backward jumps within the same block (Section 4.1), backward and forward jumps within the same block (Section 4.2), outwards jumps (Section 4.3), and coroutines (Section 4.4). For what it is worth, our embedding passes Knuth’s man-or-boy test (Section 4.5).

Each of the following kinds of jumps, except for coroutines, can be translated as a special case using more specialized control structures offered by JavaScript. We discuss such specialized translation schemes further in Section 5.

### 4.1 Backward Jumps

To simulate backward jumps within the same block, our translation simply declares a sequence of lexical variables, each denoting the continuation of a label, and uses one of these variables for each jump.

Backward jumps can be used to repeatedly try a computation until a condition is met, as in a loop. It can also be used to express more involved iteration patterns that recur in domains such as pattern matching and logical inference. These patterns are exemplified by the search phase of Knuth, Morris and Pratt’s string matcher [24].

The KMP string matcher searches for the first occurrence of a string in a text. It first preprocesses the string into a failure table, then traverses the text incrementally, searching whether the string is a prefix of the current suffix of the text. In case of character mismatch, the string is shifted farther by a distance determined by the failure table. In their original article, Knuth, Morris and Pratt display a version of the search phase that is ‘compiled’—i.e., specialized—with respect to a given string [24, Section 3] [11, Appendix]. Appendix B.1 displays this specialized version in JavaScript.

A similar illustration of backward jumps can be found in Flanagan and Matsumoto’s book about the Ruby programming language, where they use first-class continuations to simulate a subset of BASIC [17, Section 5.8.3].

## 4.2 Backward and Forward Jumps

To simulate backward and forward jumps within the same block, our translation declares a group of mutually recursive lexical variables, each denoting the continuation of a label, and uses one of these variables for each jump. Appendix B.2 shows this simulation at work with Knuth’s modification of Hoare’s Quicksort program [23, p. 285].

## 4.3 Outward Jumps

In modern parlance, outward jumps are exceptions. Again, our translation simulates them by declaring a lexical variable denoting the continuation of each label and using one for each jump.

Appendix B.3 displays a program that recursively descends into a binary tree, testing whether it represents a Calder mobile [13]. If one of the subtrees is unbalanced, the computation jumps out.

## 4.4 Coroutines

Whereas calling a subroutine transfers control to its beginning, calling a coroutine resumes execution at the point where the coroutine last yielded control [12]. Coroutines are useful for modeling domains with concurrent interacting processes. To account for coroutines, we use the `<cmd-yield>` production in Figure 1.

We have implemented a standard example of stream transducers (mapping pairs to triples [46]), Knuth’s alternative version of Quicksort using coroutines [23, p. 287], and samefringe, the prototypical example of asynchronous recursive traversals [5, 20, 35]. The samefringe program is displayed in Appendix B.4.

## 4.5 Knuth’s Man-or-Boy Test

Finally, for what it is worth, our translator passes Knuth’s man-or-boy test [22], using explicit thunks to implement call by name. Of course, this test exercises not jumps but recursion and non-local references. In that sense, it is more Rhino that passes Knuth’s test than our homomorphic translator.

## 5 Conclusion and Perspectives

We have realized, in the modern setting of JavaScript, Landin’s visionary correspondence between block-structured programs with jumps and applicative expressions [27]. Our translation is idiomatic in that it maps the blocks and declarations as well as labels and jumps of the source program homomorphically to corresponding structures in the target program. The correspondence is so direct, in fact, that we can regard the image of the translation as a source language, and thus the translation as an identity function. The idiomaticity of the translation is the key to its success in producing programs that run with the same asymptotic time and space behavior as the Algol-like programs in our source language, assuming that Rhino implements downwards contargs in a reasonable way.

The simplicity and efficacy of our translation exemplifies how the principle of idiomaticity is useful for language implementation and language design: the

correspondence between Landin’s J operator and Rhino’s continuation objects guides the implementation and design of our Algol-like language. In particular, we believe that a translation should preserve the control structures of source programs and map them to idiomatic control structures in the target language.

In this regard, our translation is extremely uniform and frugal in that it maps all control structures using continuation objects. Another extreme strategy, which some might consider less idiomatic and more pervasive, is to follow Steele [45] and translate all control structures using continuation-passing style or even trampolining. These translations call for JavaScript implementations that support either first-class continuations or proper tail recursion. Between these two simple extremes, there is an admittedly more complex middle path that relies on a control-flow analysis to detect particular patterns of control and translate them opportunistically to particular control-flow instructions in JavaScript that are more widely implemented well. For example:

- Programs that do not use labels and jumps can be translated without the thunks described in Section 2.2.
- Backward jumps within the same block can be translated to labeled `break` and `continue` statements.
- Backward and forward jumps within the same block can be translated using a loop that dispatches on a current state. (In the case of Quicksort [23], such a translation would reverse the ‘boolean variable elimination’ optimization that motivated Knuth’s use of forward jumps in the first place.)
- Outward jumps can be translated using local exceptions.

In general, each control pattern should be translated idiomatically to, and thus constitutes a new use case for, a corresponding control structure in JavaScript.

*Acknowledgments.* This article was written while the second author was visiting Aarhus University in the fall of 2008. The authors are grateful to Dave Herman for an e-mail exchange about implementing finite-state machines in JavaScript and to Florian Loitsch and the anonymous reviewers for their comments. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

## References

1. Kamal Abdali, S.: A lambda-calculus model of programming languages, part I: Simple constructs. *Computer Languages* 1(4), 287–301 (1976)
2. Kamal Abdali, S.: A lambda-calculus model of programming languages, part II: Jumps and procedures. *Computer Languages* 1(4), 303–320 (1976)
3. Kamal Abdali, S., Wise, D.S.: Standard, storeless semantics for ALGOL-style block structure and call-by-name. In: Melton, A.C. (ed.) MFPS 1985. LNCS, vol. 239, pp. 1–19. Springer, Heidelberg (1986)
4. Bawden, A.: Reification without evaluation. In: (Corky) Cartwright, R. (ed.) Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah, July 1988, pp. 342–351. ACM Press, New York (1988)

5. Biernacki, D., Danvy, O., Shan, C.-c.: On the static and dynamic extents of delimited continuations. *Science of Computer Programming* 60(3), 274–297 (2006)
6. Bjørner, D., Jones, C.B.: *Formal Specification & Software Development*. Prentice-Hall International, London (1982)
7. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM* 9(5), 366–371 (1966)
8. Burge, W.H.: *Recursive Programming Techniques*. Addison-Wesley, Reading (1975)
9. Burstall, R.M.: Writing search algorithms in functional form. In: Michie, D. (ed.) *Machine Intelligence*, vol. 5, pp. 373–385. Edinburgh University Press (1969)
10. Clements, J., Sundaram, A., Herman, D.: Implementing continuation marks in JavaScript. In: Clinger, W. (ed.) *Proceedings of the 2008 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Victoria, British Columbia, September 2008, pp. 1–9 (2008)
11. Consel, C., Danvy, O.: Partial evaluation of pattern matching in strings. *Information Processing Letters* 30(2), 79–86 (1989)
12. Conway, M.E.: Design of a separable transition-diagram compiler. *Communications of the ACM* 6(7), 396–408 (1963)
13. Danvy, O.: Sur un exemple de Patrick Greussay. Research Report BRICS RS-04-41, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark (December 2004)
14. Danvy, O., Millikin, K.: A rational deconstruction of Landin’s SECD machine with the J operator. *Logical Methods in Computer Science* 4(4:12), 1–67 (2008)
15. Kent Dybvig, R., Hieb, R.: Engines from continuations. *Computer Languages* 14(2), 109–123 (1989)
16. Fischer, M.J.: Lambda-calculus schemata. *LISP and Symbolic Computation* 6(3/4), 259–288 (1993), <http://www.brics.dk/~hosc/vol06/03-fischer.html>; A preliminary version was presented at the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7(1) (January 1972)
17. Flanagan, D., Matsumoto, Y.: *The Ruby Programming Language*. O’Reilly Media, Inc., Sebastopol (2008)
18. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Lee, P. (ed.) *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Paris, France, vol. 34(9), pp. 18–27. ACM Press, New York (1999)
19. Girard, J.-Y.: Locus solum. *Mathematical Structures in Computer Science* 11(3), 301–506 (2001)
20. Hewitt, C., Bishop, P., Steiger, R., Greif, I., Smith, B., Matson, T., Hale, R.: Behavioral semantics of nonrecursive control structures. In: Robinet, B. (ed.) *Programming Symposium*. LNCS, vol. 19, pp. 385–407. Springer, Heidelberg (1974)
21. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London (1993), <http://www.dina.kvl.dk/~sestoft/pebook/>
22. Knuth, D.E.: Man or boy? *ALGOL Bulletin* 17, 7 (1964)
23. Knuth, D.E.: Structured programming with go to statements. *Computing Surveys* 6(4), 261–301 (1974)
24. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. *SIAM Journal on Computing* 6(2), 323–350 (1977)

25. Kranz, D., Kesley, R., Rees, J., Hudak, P., Philbin, J., Adams, N.: Orbit: An optimizing compiler for Scheme. In: Proceedings of the ACM SIGPLAN 1986 Symposium on Compiler Construction, Palo Alto, California, pp. 219–233. ACM Press, New York (1986)
26. Kranz, D.A.: ORBIT: An Optimizing Compiler for Scheme. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut (February 1988), Research Report 632
27. Landin, P.J.: A correspondence between Algol 60 and Church's lambda notation, Parts 1 and 2. *Communications of the ACM* 8, 89–101, 158–165 (1965)
28. Landin, P.J.: The next 700 programming languages. *Communications of the ACM* 9(3), 157–166 (1966)
29. Landin, P.J.: Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In: Danvy, O. (ed.) *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, CW 1997 (1997); Technical report BRICS NS-96-13, Aarhus University, pp. 1:1–9, Paris, France (January 1997)
30. Loitsch, F.: Scheme to JavaScript Compilation. PhD thesis, Université de Nice, Nice, France (March 2009)
31. Mazurkiewicz, A.W.: Proving algorithms by tail functions. *Information and Control* 18, 220–226 (1971)
32. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3(4), 184–195 (1960)
33. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) *Information Processing 1962*, Proceedings of IFIP Congress 62, pp. 21–28. North-Holland, Amsterdam (1962)
34. McCarthy, J.: A formal description of a subset of ALGOL. In: Steel, T.B. (ed.) *Formal Language Description Languages for Computer Programming*, pp. 1–12. North-Holland, Amsterdam (1966)
35. McCarthy, J.: Another samefringe. *SIGART Newsletter* 61 (February 1977)
36. McDermott, D.: An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In: Davis, R.E., Allen, J.R. (eds.) *Conference Record of the 1980 LISP Conference*, Stanford, California, August 1980, pp. 154–162 (1980)
37. Minamide, Y.: Selective tail call elimination. In: Cousot, R. (ed.) *SAS 2003. LNCS*, vol. 2694, pp. 153–170. Springer, Heidelberg (2003)
38. Moggi, E.: Notions of computation and monads. *Information and Computation* 93, 55–92 (1991)
39. Lockwood Morris, F.: Correctness of Translations of Programming Languages – an Algebraic Approach. PhD thesis, Computer Science Department, Stanford University (August 1972), Technical report STAN-CS-72-303
40. Lockwood Morris, F.: The next 700 formal language descriptions. *Lisp and Symbolic Computation* 6(3/4), 249–258 (1993); Reprinted from a manuscript dated (1970)
41. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: *Proceedings of 25th ACM National Conference*, Boston, Massachusetts, pp. 717–740 (1972); Reprinted in *Higher-Order and Symbolic Computation* 11(4), 363–397 (1998), with a foreword [42]
42. Reynolds, J.C.: Definitional interpreters revisited. *Higher-Order and Symbolic Computation* 11(4), 355–361 (1998)
43. Schinz, M., Odersky, M.: Tail call elimination on the Java Virtual Machine. In: Benton, N., Kennedy, A. (eds.) *BABEL 2001: First International Workshop on Multi-Language Infrastructure and Interoperability*, Firenze, Italy, September 2001. *Electronic Notes in Theoretical Computer Science*, vol. (59), pp. 155–168. Elsevier Science, Amsterdam (2001)

44. Stallman, R.M.: Phantom stacks: If you look too hard, they aren't there. In: AI Memo 556, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (July 1980)
45. Steele Jr., G.L.: Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory. Massachusetts Institute of Technology, Cambridge, Massachusetts, Technical report AI-TR-474 (May 1978)
46. Talcott, C.L.: The Essence of  $\mathcal{R}$ um: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation. PhD thesis, Department of Computer Science, Stanford University, Stanford, California (August 1985)
47. Tarditi, D., Lee, P., Acharya, A.: No assembly required: Compiling Standard ML to C. ACM Letters on Programming Languages and Systems 1(2), 161–177 (1992)
48. Thielecke, H.: Comparing control constructs by double-barrelled CPS. Higher-Order and Symbolic Computation 15(2/3), 141–160 (2002)
49. Wadler, P.: The essence of functional programming (invited talk). In: Appel, A.W. (ed.) Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, pp. 1–14. ACM Press, New York (1992)

## A Grammar of JavaScript in the Image of the Translation

```

<ascii>      ::= ....
<ident>      ::= [a-zA-Z0-9]+ -- not including JI
<program>    ::= var main = function () {
                  var JI = new Continuation();
                  <dcl>*
                  return function () { <cmd>* };
                };
                main();
<dcl>        ::= <dcl-var>
                  | <dcl-proc>
                  | <dcl-label>
<cmd>        ::= <cmd-block>
                  | <cmd-assign>
                  | <cmd-goto>
                  | <cmd-yield>
                  | <cmd-return>
                  | <cmd-if>
                  | <cmd-print>
                  | <cmd-exp>
<exp>        ::= <exp-var>
                  | <exp-int>
                  | <exp-bool>
                  | <exp-chr>
                  | <exp-str>
                  | <exp-bottom>
                  | <exp-call>
                  | <exp-array>
                  | <exp-index>
                  | <exp-length>

```

```

        | <exp-binary>
        | <exp-nest>
<dcl-var>   ::= var <ident>;
<formals>    ::= <ident> | <ident> , <formals>
<dcl-proc>   ::= var <ident> = function ( <formals>? ) {
                  var JI = new Continuation();
                  <dcl>*
                  return function () { <cmd>* };
                };
<dcl-label>  ::= var <ident> = function () { JI(function () { <cmd>* }); };
<cmd-block>   ::= { <dcl>* <cmd>* }
<cmd-assign>  ::= <lvalue> = <exp>;
<cmd-goto>   ::= <ident>();
<cmd-yield>  ::= (function () {<ident> = new Continuation(); <cmd-goto>})();
<cmd-return> ::= return <exp>;
<cmd-if>     ::= if ( <exp> ) <cmd>
<cmd-print>  ::= print( <exp> );
<cmd-exp>    ::= <exp>;
<lvalue>      ::= <exp-var> | <exp-index>
<exp-var>    ::= <ident>
<exp-int>    ::= [0-9]*
<exp-bool>   ::= true | false
<exp-chr>    ::= '<ascii>'
<exp-str>    ::= "<ascii>*"
<exp-bottom> ::= undefined
<args>        ::= <exp> | <exp> , <args>
<exp-call>   ::= <ident>(<args>? )()
<array-elm>  ::= <exp> | <exp> , <array-elm>
<exp-array>  ::= [ <array-elm> ]
<exp-index>  ::= <exp>[ <exp> ]
<exp-length> ::= <exp>.length
<bin-op>     ::= == | != | < | > | + | - | *
<exp-binary> ::= <exp> <bin-op> <exp>
<exp-nest>   ::= ( <exp> )

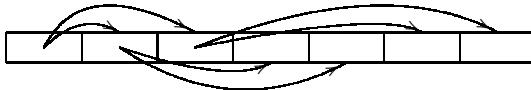
```

## B Example JavaScript Programs

In the following examples we represent trees in terms of arrays. Depending on the use, a tree can have either weighted internal nodes, as used in Calder mobiles (Appendix B.3), or weighted leaf nodes, as used in Samefringe (Appendix B.4).



In each case we denote the weightless nodes (•) with  $-1$ . A tree is then encoded as a linear block of values with the successor functions  $\text{left}(n) = 2n + 1$  and  $\text{right}(n) = 2n + 2$  as illustrated below.



## B.1 Backward Jumps: KMP String Search

```

        k = k + 1;
        L7();
    })};}
var L7 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    L8();
})};}
var L8 = function () { JI(function () {
    if (text[k] != 'c') L5();
    k = k + 1;
    L9();
})};}
var L9 = function () { JI(function () {
    if (text[k] != 'a') L0();
    k = k + 1;
    L10();
})};}
var L10 = function () { JI(function () {
    if (text[k] != 'b') L1();
    k = k + 1;
    return k - 10;
})};}
return function () {
    k = 0;
    n = text.length;
    text = text + "@a";
    L0();
};
};

return function () {
    print(kmp("ababcbcabcacabcaab")());
};

};

main();

```

## B.2 Backward and Forward Jumps: Quicksort

```

var main = function () {
    var JI = new Continuation();
    var A;
    var qs = function (m, n) {
        var JI = new Continuation();
        var i;
        var j;
        var v;
        var loop1 = function () { JI(function () {
            if (A[i] > v) {
                A[j] = A[i];
                upf();
            }
        })};
    };
};

```

```

        }
        upt();
    })};}
var upt = function () { JI(function () {
    i = i + 1;
    if (i < j) loop1();
    common();
})};}
var loop2 = function () { JI(function () {
    if (v > A[j]) {
        A[i] = A[j];
        upt();
    }
    upf();
})};}
var upf = function () { JI(function () {
    j = j - 1;
    if (i < j) loop2();
    common();
})};}
var common = function () { JI(function () {
    A[j] = v;
    if (n - m > 1) {
        qs(m, j - 1)();
        qs(j + 1, n)();
    }
})};}
return function () {
    i = m;
    j = n;
    v = A[j];
    loop1();
};
};

return function () {
    A = [5, 2, 1, 4, 6, 3];
    print("Random: " + A);
    qs(0, A.length - 1)();
    print("Sorted: " + A);
};

};

main();

```

### B.3 Outward Jumps: Calder Mobiles

```

var main = function () {
    var JI = new Continuation();
    var calder = function (T) {
        var JI = new Continuation();
        var fail = function () { JI(function () {

```

```

        return false;
    });
    var visit = function (i) {
        var JI = new Continuation();
        var n;
        var n1;
        var n2;
        return function () {
            n = T[i];
            if (n == -1) return 0;
            i = i * 2;
            n1 = visit(i + 1)();
            n2 = visit(i + 2)();
            if (n1 == n2) return n + n1 + n2;
            fail();
        };
    };
    return function () {
        visit(0)();
        return true;
    };
};
return function () {
    print(calder([2, 5, 1, 2, 2, 4, 4,
                  -1, -1, -1, -1, -1, -1, -1, -1])());
};
};

main();

```

#### B.4 Coroutines: Samefringe

```

var main = function () {
    var JI = new Continuation();
    var t1;
    var t2;
    var traverse = function (t, f) {
        var JI = new Continuation();
        var visit = function (i) {
            var JI = new Continuation();
            var n;
            return function () {
                n = t[i];
                if (n != -1) return f(n)();
                i = 2 * i;
                visit(i + 1)();
                visit(i + 2)();
            };
        };
        return function () {
            visit(0)();
        };
    };
}

```

```
        };
    };
    var samefringe = function (t1, t2) {
        var JI = new Continuation();
        var v1;
        var v2;
        var next1 = function (e) {
            var JI = new Continuation();
            return function () {
                v1 = e;
                (function () { l1 = new Continuation(); l2(); })();
            };
        };
        var next2 = function (e) {
            var JI = new Continuation();
            return function () {
                v2 = e;
                (function () { l2 = new Continuation(); compare(); })();
            };
        };
        var l1 = function () { JI(function () {
            traverse(t1, next1)();
            next1(undefined)();
       });};
        var l2 = function () { JI(function () {
            traverse(t2, next2)();
            next2(undefined)();
       });};
        var compare = function () { JI(function () {
            if (v1 != v2) return false;
            if (v1 == undefined) return true;
            l1();
       });};
        return function () {
            l1();
        };
    };
    return function () {
        t1 = [-1, 1,
               -1, undefined, undefined, 2, 3];
        t2 = [-1,
               -1, 3, 1, 2, undefined, undefined];
        print(samefringe(t1, t2)());
    };
};
main()();
```

# Model-Driven Engineering from Modular Monadic Semantics: Implementation Techniques Targeting Hardware and Software\*

William L. Harrison<sup>1</sup>, Adam M. Procter<sup>1</sup>, Jason Agron<sup>2</sup>, Garrin Kimmell<sup>3</sup>,  
and Gerard Allwein<sup>4</sup>

<sup>1</sup> Department of CS, University of Missouri, Columbia, Missouri, USA

<sup>2</sup> Department of CS & CE, University of Arkansas, Fayetteville, Arkansas, USA

<sup>3</sup> Department of EECS, University of Kansas, Lawrence, Kansas, USA

<sup>4</sup> US Naval Research Laboratory, Code 5543, Washington, DC, USA

**Abstract.** Recent research has shown how the formal modeling of concurrent systems can benefit from monadic structuring. With this approach, a formal system model is really a program in a domain specific language defined by a monad for shared-state concurrency. Can these models be compiled into efficient implementations? This paper addresses this question and presents an overview of techniques for compiling monadic concurrency models directly into reasonably efficient software and hardware implementations. The implementation techniques described in this article form the basis of a semantics-directed approach to model-driven engineering.

## 1 Introduction

System software is notoriously difficult to reason about—formally or informally—and this, in turn, greatly complicates the construction of high assurance systems. This difficulty stems from the conceptual “distance” between the abstract models of systems and concrete system implementations. Formal system models are expressed in terms of high-level abstractions while system implementations reflect the low-level details of hardware, machine languages and C. One recent trend in systems engineering—model-driven engineering (MDE) [39]—attempts to overcome this distance by synthesizing implementations directly from system specifications. The MDE methodology is attractive for high assurance applications because the process proceeds from a domain specific modeling language that, when specified with a suitable formal semantics, will support verification.

The starting point for this work is recent research applying modular monadic semantics to the design and verification of trustworthy concurrent systems [15, 14, 12, 13]. There are a number of natural “next” questions concerning the set of design and verification principles developed in the aforementioned publications.

---

\* This research was supported by NSF CAREER Award 00017806; US Naval Res. Lab. Contract 1302-08-015S; DARPA/AFRL Contract FA8650-07-C-7733; the Gilliom Cyber Security Gift Fund; Cadstone, LLC; and the ITTC Tech. Transfer Fund.

Can system implementations be generated from monad-based specifications and, if so, how is this best accomplished? Are critical system properties preserved by implementation techniques? Can acceptable performance across many dimensions (including speed, size, power consumption, etc.) be achieved? This paper addresses the first question and leaves the others to future work.

This paper considers an instance of MDE for which the models are based in the modular monadic semantics (MMS) of concurrent languages [12,34]. Monads have a dual nature, being both algebraic structures with properties and a kind of domain-specific language (DSL) supporting programming. The view of monads as DSLs is just the standard view within the functional programming community expressed in a non-standard way: monads encapsulate a specific flavor of computation and provide language constructs (i.e., monadically-typed operators) in which to build computations.

**The contributions of this paper are:** (1) An exploration of the design requirements for a monadic DSL for describing small concurrent systems with shared state (e.g., embedded controllers). This language is called *Cheap Threads* after an article of the same name [18]. (2) Implementation techniques for Cheap Threads (CT) programs targeting both a fixed instruction set and hardware circuits. The first technique is a traditional compiler for a non-traditional language. The second technique translates a CT program into VHDL code from which circuitry may be synthesized and loaded into an FPGA. A DSL for specifying state machines called *FSMLang* [3] serves as an intermediate language. (3) A significant case study demonstrating the application of these techniques to the automatic generation of software-defined radios [23] from CT-style specifications.

It is sometimes said of domain specific languages that what is left out of them is more important than what is left in. By restricting its expressiveness, a DSL design can better capture the idioms of its targeted domain. This is just as true in our case where the limitations imposed are intended to aid in the ultimate goal of verifying high assurance properties. Both with respect to its semantics and implementation strategies, what is left out of CT is crucial to the success of this approach. Because the semantics of CT is fundamentally simpler than that of Haskell, its implementation can be made far simpler as well. Although it dispenses with the constructs that complicate Haskell 98’s semantics, CT retains the necessary expressiveness—but no more expressiveness than is necessary.

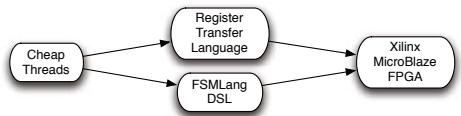
The decision to create a standalone CT language (as opposed to an embedding within a language such as Haskell) was made for the sake of semantic simplicity. Haskell 98 [36], for example, has a surprisingly complicated semantics [17, 21, 11] due to certain of its features (e.g., the `seq` operator, expressive pattern-matching, polymorphic recursion and type classes), while its extension, GHC Haskell, has no standard semantics at all. Furthermore, excluding features such as general recursion results in much more predictable time and space behavior. CT possesses a built-in semantics as each CT language construct corresponds to an algebraic operator of a fixed monad.

Syntactically, CT is simply a sublanguage of Haskell, extended with a simple declaration form for specifying monads. Any CT program is also a Haskell pro-

gram that can be executed with standard implementations like GHC or Hugs. CT contains only the small core of Haskell 98 necessary for defining computations within these monads (function and data declarations, etc.). In particular, CT dispenses with first-class functions, curried functions and partial application, recursive data structures, type classes, polymorphism, the *IO* monad, and much of the complexity of Haskell 98’s pattern matching. General recursion is eschewed in favor of an explicit fixed point operator which operates only on tail-recursive functions.

Another reason to define CT as a standalone language is that, as it will be independent of Haskell implementations, the ultimate results of this research can be re-used far more readily in a variety of settings. There is no high assurance run-time system for Haskell, so relying on the GHC run-time, for example, just “kicks the high assurance can down the road.” For embedded systems and many varieties of system software (e.g., garbage collectors, network controllers, device drivers, web servers, etc.), the size of the object code produced would have to be small to be practical, thus eliminating all current Haskell implementations. The presence of garbage collection in Haskell run-time systems is completely unacceptable for some software systems because of timing issues (e.g., flight control software). It is fortunate, therefore, that CT does not require the full power of the Haskell RTS with its attendant unpredictable time and space behavior.

When considered individually, software compilation and hardware synthesis represent signal achievements in Computer Science. While there has been some success with mixed target compilation—i.e., the generation of either hardware or software implementations from a single source—the record is mixed. The challenge derives in trying to compile the same specification into targets—i.e., hardware and software—that encapsulate vastly different notions of computation.



The use of monads in the present work allows us to explicitly tailor the source notion of computation so that it may be implemented efficiently in either hardware or software. The implementation techniques considered here are portrayed in the inset figure. The Cheap Threads language is defined in Section 3. The top route (described in Section 4) shows what is essentially a traditional compiler for a non-traditional language producing RISC MicroBlaze machine language. The lower path (described in Section 5) first generates an intermediate form in *FSMLang*, which is a DSL specifying for abstract state machines. *FSMLang* provides a clean target for synthesis at a lower level of abstraction than CT and can be readily translated into a netlist for the Xilinx FPGA. The rest of this section presents related work and Section 2 presents an overview of monadic semantics and defunctionalization. Section 6 presents a case study in the application of Cheap Threads to the specification and implementation of software defined radios. Section 7 presents conclusions and future work.

**Related Work.** Recent research concerns the construction and verification of formal models of separation kernels using concepts from language semantics

[15, 14, 12]. These models may be viewed as a domain-specific language (DSL) for separation kernels and can easily be embedded in the higher-order functional programming language Haskell 98 to support rapid prototyping and testing.

The “by layer” approach to implementing monadic programs—i.e., compiling by monad transformer—was first explored by Filinski [9] who demonstrated how the most commonly used layers (i.e., monad transformers) may be translated into a  $\lambda$ -calculus with first-class continuations; the resulting program could be then further compiled in the manner of Appel [5]. Filinski’s approach would work here as it handles all of the monad transformers used in CT. Li and Zdancewic [26] show how a monadic language of threads may be implemented via the GHC compiler; their implementation is efficient with respect to execution speed. Their work is not intended to provide efficiency with respect to code size as is ours. Liang and Hudak [27] embed non-proper morphisms of a source monadic expression in Standard ML, thereby providing a means of implementing monadic programs. We opted for a more standard approach to compilation than these as doing so would give more control over the target code produced and, furthermore, seemed more susceptible to verification.

Recent research applies DSLs to bridge the distance between the abstract models of systems and concrete system implementations. DSLs targeting the design, construction, and/or verification of application-specific schedulers are CATAPULTS [38], BOSSA [24], and Hume [10]. DSLs have also been successfully applied to the construction of sensor networks and network processors [25, 7]. The research reported here has similar goals to this work, although we also seek to address high assurance as well; this motivated our use of monadic semantics as a basis for system modeling and implementation.

Semantics-directed compilation (SDC) [35] is a form of programming language compilation which processes a semantic representation of the source program to synthesize the target program. Because one starts from a formal specification of the input program, semantics-directed compilers are more easily proved correct than traditionally structured compilers, and this is the principal motivation for structuring compilers in this manner. This research uses a classic program transformation called *defunctionalization* as a basis for SDC. Defunctionalization is a powerful program transformation discovered originally in the early 1970s by Reynolds [37] that has found renewed interest in the work of Danvy et al. [8, 1, 2]. Hutton and Wright [20] defunctionalize an interpreter for a small language with exceptions into an abstract machine implementation. Earlier, they described a verified compiler for the same language [19].

Monads have been used as a modular structuring technique for DSLs [41] and language compilers [16]. The research reported here differs from these in that monads are taken as a source language to be implemented rather than as a tool for organizing the implementations of language processors.

Lava [6] is a domain-specific language for describing hardware circuitry, embedded in Haskell. Similarly, HAWK [29] is a Haskell-based DSL for the modeling and verification of microprocessor architectures. Both of these DSLs utilize the embedding within Haskell to allow the modeling of hardware signals as Haskell

lazy lists, providing a simple simulation capability. Moreover, Lava and Hawk allow a programmer to define the *structural* implementation of circuits, using the Haskell host language to capture common structural patterns as recursive combinators. This stands in contrast to the work presented in this paper, where we use a monadic language to describe and compile *behavioral* models of systems, in keeping with our goal of enabling verification of high-level system properties.

SAFL [32, 40] is a functional language designed for efficient compilation to hardware circuits. SAFL provides a first-order language, restricted to allow static allocation of resources, in which programs are described behaviorally and then compiled to Verilog netlists for simulation and synthesis. An extension, called SAFL+, adds mutable references and first-class channels for concurrency. CT provides similar capabilities [23], especially in regards to hardware compilation. However, language extensions facilitating stateful and concurrent effects are constructed using the composition of monad transformers to add these orthogonal notions of computation.

## 2 Background

We assume of necessity that the reader possesses some familiarity with monads and their uses in functional programming and the denotational semantics of languages with effects. This section includes additional background information about monadic semantics intended to serve as a quick review. Readers requiring more should consult the references for further background [31, 28].

**Monads.** A monad is a triple  $\langle M, \eta, \star \rangle$  consisting of a type constructor  $M$  and two operations  $\eta$  (unit) and  $\star$  (bind) with the following types:  $\eta : a \rightarrow M a$  and  $(\star) : M a \rightarrow (a \rightarrow M b) \rightarrow M b$ . These operations must obey the well-known *monad laws* [31]. The  $\eta$  operator is the monadic analogue of the identity function, injecting a value into the monad. The  $\star$  operator is a form of sequential application. The “null bind” operator,  $\gg : M a \rightarrow M b \rightarrow M b$ , is defined as:  $x \gg k = x \star \lambda \_. k$ . The binding (i.e., “ $\lambda \_$ ”) acts as a dummy variable, ignoring the value produced by  $x$ .

Recent research [12, 18] has demonstrated how concurrent behaviors (including process synchronization, asynchronous message passing, preemption, forking, etc.) may be described formally and succinctly using monadic semantics. These kernels are constructed using *resumption* monads [34]. *Resumptions* are a denotational model for concurrency discovered by Plotkin that were later formulated as monads by Moggi [31]. Intuitively, a resumption model views a program as a (potentially infinite) sequence of *atoms*,  $[a_0, a_1, \dots]$ , where each  $a_i$  may be thought of as an atomic machine instruction. Concurrent execution of multiple programs may then be modeled as an interleaving of each of these program threads (or as the set of all such interleavings).

Monad transformers allow us to easily combine and extend monads. There are various formulations of monad transformers; we follow that given in Liang et al. [28]. Below we give several equivalent definitions for both a “layered” state monad,  $K$ , and a resumption monad,  $R$ . The first definition of monad  $K$

is in terms of state monad transformers,  $StateT\ Reg_i$ , and the identity monad,  $I\ a = a$ . The state types,  $Reg_i$ , can be taken, for the sake of this article, to represent integer registers. The resumption monad,  $R$ , is defined first with the resumption monad transformer  $ResT$ , and then in Haskell. The definitions of the state monad and resumption transformers can be found elsewhere [28, 12].

$$\begin{aligned} K &= StateT\ Reg_1 (\cdots (StateT\ Reg_n\ I) \cdots) \\ KA &\cong Reg_1 \rightarrow \cdots \rightarrow Reg_n \rightarrow (A \times Reg_1 \times \cdots \times Reg_n) \\ R &= ResT\ K \\ \mathbf{data}\ R\ a &= Done\ a \mid Pause\ (K\ (R\ a)) \end{aligned}$$

These two monads define the following language:

$$get_i : K\ Reg_i \quad put_i : Reg_i \rightarrow K\ () \quad step : K\ a \rightarrow R\ a$$

The operation,  $get_i$ , reads the current contents of the  $i$ th register and ( $put_i\ v$ ) stores  $v$  in the  $i$ th register. The operation,  $step\ \varphi$ , makes an atomic action out of the  $K$ -computation  $\varphi$ . Monadic operations are sometimes referred to as *non-proper* morphisms.

Resumption based concurrency is best explained by an example. We define a *thread* to be a (possibly infinite) sequence of “atomic operations.” Think of an atomic operation as a single machine instruction and a thread as a stream of such instructions characterizing program execution. Consider first that we have two simple threads  $a = [a_0; a_1]$  and  $b = [b_0]$ . According to the “concurrency as interleaving” model, concurrent execution of threads  $a$  and  $b$  means the set of all their possible interleavings:  $\{[a_0; a_1; b_0], [a_0; b_0; a_1], [b_0; a_0; a_1]\}$ .

The  $ResT$  monad transformer introduces lazy constructors  $Pause$  and  $Done$  that play the rôle of the lazy cons operator in the stream example above. If the atomic operations of  $a$  and  $b$  are computations of type  $K\ ()$ , then the computations of type  $R\ ()$  are the set of possible interleavings:

$$\begin{aligned} &Pause\ (a_0 \gg \eta(Pause\ (a_1 \gg \eta(Pause\ (b_0 \gg \eta(Done\ ()))))))) \\ &Pause\ (a_0 \gg \eta(Pause\ (b_0 \gg \eta(Pause\ (a_1 \gg \eta(Done\ ()))))))) \\ &Pause\ (b_0 \gg \eta(Pause\ (a_0 \gg \eta(Pause\ (a_1 \gg \eta(Done\ ()))))))) \end{aligned}$$

In CT, these threads would be constructed without reference to  $Pause$  and  $Done$  using  $step$ :  $(step\ a_0 \gg_R step\ a_1 \gg_R step\ b_0)$ ; this thread is equal to the first one above.

Just as streams are built with a lazy “cons” operation  $(h : t)$ , the resumption-monadic version uses an analogous formulation:  $Pause\ (h \gg \eta t)$ . The laziness of  $Pause$  allows infinite *computations* to be constructed in  $R$  just as the laziness of cons in  $(h : t)$  allows infinite *streams* to be constructed.

A refinement to the concurrency model provided by  $ResT$ , called *reactive* resumptions [12], supports a request-and-response form of concurrent interaction, while retaining the same notion of interleaving concurrency. Reactive concurrency monads also define a *signal* construct for sending signals between threads. We do not consider the *signal* construct in this article, although it is used in the case study in Section 6. Its implementation is similar to a procedure call.

<pre>(* main0 : int -&gt; int *) fun main0 n   = fac0 n (* fac0 : int -&gt; int *) and fac0 0   = 1   fac0 n   = n * (fac0 (n - 1))</pre>	$\begin{array}{ll} n & \xrightarrow{\text{init}} \langle n, C_0 \rangle \\ \langle 0, k \rangle & \xrightarrow{\text{fac}} \langle k, 1 \rangle \\ \langle n, k \rangle & \xrightarrow{\text{fac}} \langle n - 1, C_1(n, k) \rangle \\ \langle C_1(n, k), v \rangle & \xrightarrow{\text{cont}} \langle k, n \times v \rangle \\ \langle C_0, v \rangle & \xrightarrow{\text{final}} v \end{array}$
---	---

**Fig. 1.** Defunctionalizing Produces an Abstract State Machine The factorial function, `fac0` (left), is transformed via defunctionalization into the state machine (right). Example originally appears in Danvy [2].

**Defunctionalization.** Defunctionalization is a program transformation (due originally to Reynolds [37] and rejuvenated of late by Danvy et al. [1, 2]) that transforms an arbitrary higher order functional program into an equivalent abstract state machine. Defunctionalization may also be performed on monadic programs as well [2]. This section reviews defunctionalization.

<pre>(* main1 : int -&gt; int *) fun main1 n   = fac1 (n, fn a =&gt; a) (* fac1 : int*(int-&gt;int)-&gt;int *) and fac1 (0, k)   = k 1   fac1 (n, k)   = fac1 (n - 1, fn v =&gt; k (n * v))</pre>	<pre>(* main2 : int -&gt; int *)  datatype cont fun main2 n   = C0   C1 of int*cont (* fac2:int*cont-&gt;int *)  (* appcont : cont*int-&gt;int *) and fac2 (0, k)   = appcont (C0, v)   fac2 (n, k)   = appcont (C1 (n, k), v) = fac2 (n-1, C1(n,k))    = appcont (k, n * v)</pre>
---	--

**Fig. 2.** Defunctionalization process for the factorial function. The function, `fac0` (see Fig. 1, left), is transformed into an equivalent state machine by CPS transformation (left), closure conversion (right), and defunctionalization (see Fig. 1, right).

Defunctionalizing the factorial function (Fig. 1, left) produces an equivalent state machine (Fig. 1, right). In this machine, there are three types of configurations on which the rules act; integers are initial/final configurations, pairs of type `int*cont` and pairs of type `cont*int`. The translation first performs the continuation-passing style (CPS) transformation (Fig. 2, left) to expose control flow and then performs closure conversion (Fig. 2, right) to represent the machine states as concrete, first-order data. The function `fac2` resulting from closure conversion is then reformatted into the familiar arrow style for rewrite rules (Fig. 1, right).

Defunctionalization for monadic programs proceeds along precisely the same lines as before once the definitions for the monadic operators (i.e.,  $\eta$ ,  $\star$ , and non-proper morphisms) are unfolded [2]. CT programs are simpler to defunctionalize than general higher-order functions. Because the control flow within a

```

monad  $K = \text{StateT}(\text{Int})\ G$ 
monad  $R = \text{ResT}\ K$ 

actionA ::  $K()$ 
actionA =  $\text{get}_G \star_K \lambda g. \text{put}_G(g + 1)$ 

actionB ::  $K()$ 
actionB =  $\text{get}_G \star_K \lambda g. \text{put}_G(g - 1)$ 

chan ::  $\text{Int} \rightarrow \text{Int} \rightarrow R()$ 
chan =  $\text{fix}(\lambda \kappa. \lambda a. \lambda b.$ 
         $\quad \text{step}(\text{put}_G a \gg_K \text{actionA} \gg_K \text{get}_G) \star_R \lambda newa.$ 
         $\quad \text{step}(\text{put}_G b \gg_K \text{actionB} \gg_K \text{get}_G) \star_R \lambda newb.$ 
         $\quad \kappa newa newb)$ 

main ::  $R()$ 
main =  $\text{chan}\ 0\ 0$ 

```

**Fig. 3.** Example Cheap Threads program

CT program is already made manifest by its monadic structure, the CPS transformation is unnecessary. We show below in Section 5 how CT programs may be defunctionalized.

### 3 Defining the Cheap Threads Language

The CT language is a proper subset of Haskell 98, extended with a special declaration form for specifying monads. It shares a concrete syntax with Haskell—in fact, any CT program may be tested in a standard Haskell environment such as GHC or Hugs, provided that Haskell definitions are supplied for the monads declared in that program. The implementation of tuples and algebraic data types, while straightforward, is not discussed in this paper in order to simplify the presentation. These features are not needed for the case study.

Figure 4 gives a grammar for CT. A program consists of one or more declarations, which may be a type signature, a function declaration, a data declaration, or a monad declaration. All function declarations must be preceded by an explicit type signature. The distinguished symbol *main* serves as the main entry point to the program and must have type  $R()$ . An example program is given in Figure 3. The example defines two atomic state operations *actionA* and *actionB*, which respectively increment and decrement the global register  $G$ . The function *chan* interleaves an infinite sequence of *actionA* operations with an infinite sequence of *actionB*. Between atomic operations, *chan* performs a context switch by saving the value of  $G$  in process *A*, and restoring the value of  $G$  in process *B* (or vice versa). As a result, the processes *A* and *B* do not affect each other's execution, even though they both make use of the same global register.

$  \begin{array}{l}  \text{program} ::= \text{decl}^* \\  \text{decl} ::= \text{tysig} \\  \quad   \quad \text{fundecl} \\  \quad   \quad \text{datadecl} \\  \quad   \quad \text{monaddecl} \\  \text{fundecl} ::= \text{ident} \text{ ident}^* = \text{expr} \\  \text{datadecl} ::= \\  \quad   \quad \text{data} \text{ } \text{dtype} = \text{condecl} \{ \mid \text{condecl} \}^* \\  \text{condecl} ::= \text{constr} \text{ } \text{basetype}^* \\  \text{monaddecl} ::= \text{monad} \text{ } K = \text{layer} \{ + \text{layer} \}^* \\  \quad   \quad \text{monad} \text{ } R = \text{ResT } K \\  \text{layer} ::= \text{StateT} \text{ } (\text{basetype}) \text{ ident} \\  \text{tysig} ::= \text{ident} :: \text{type} \\  \text{basetype} ::= \text{Int} \\  \quad   \quad \text{Bool} \\  \quad   \quad \text{O} \\  \quad   \quad (\text{basetype} \{ , \text{basetype} \}^+)^* \\  \quad   \quad \text{dtype} \\  \text{type} ::= (\text{type}) \\  \quad   \quad \text{basetype} \\  \quad   \quad \text{type} \rightarrow \text{type} \\  \quad   \quad m \text{ } \text{basetype} \\  \text{pat} ::= - \\  \quad   \quad \text{ident} \\  \quad   \quad \text{constr} \text{ } \text{ident}^* \\  \quad   \quad (\text{ident} \{ , \text{ident} \}^+)^*  \end{array}  $	$  \begin{array}{l}  \text{expr} ::= (\text{expr}) \\  \quad   \quad \text{expr} \text{ } \text{expr} \\  \quad   \quad \text{expr} \text{ } \text{binop} \text{ } \text{expr} \\  \quad   \quad \text{integer\_literal} \\  \quad   \quad \text{boolean\_literal} \\  \quad   \quad -\text{expr} \\  \quad   \quad \text{if} \text{ } \text{expr} \text{ then } \text{expr} \text{ else } \text{expr} \\  \quad   \quad \text{case} \text{ } \text{expr} \text{ of} \\  \quad \quad   \quad \{ \text{pat} \rightarrow \text{expr} \}^* \\  \quad   \quad (\text{expr} \{ , \text{expr} \}^+)^* \\  \quad   \quad () \\  \quad   \quad \text{expr} \star_m \text{expr} \\  \quad   \quad \text{expr} \star_m \text{lambda} \\  \quad   \quad \text{expr} \gg_m \text{expr} \\  \quad   \quad \eta_m \text{expr} \\  \quad   \quad \text{fix} \text{expr} \\  \quad   \quad \text{fix} \text{lambda} \\  \quad   \quad \text{get}_\text{ident} \\  \quad   \quad \text{put}_\text{ident} \text{expr} \\  \quad   \quad \text{step} \text{expr} \\  \text{lambda} ::= (\text{lambda}) \\  \quad   \quad \lambda \text{ident}. \text{expr} \\  \quad   \quad \lambda \text{ident}. \text{lambda} \\  m ::= K \mid R  \end{array}  $
--	--

**Fig. 4.** Grammar for the Cheap Threads language

While CT’s concrete syntax is borrowed from Haskell, it is fundamentally a much simpler language. We note a few important distinctions at the outset.

**Declaration Form for Monads.** A special *monad* declaration form is used to construct state and resumption monads. The types and morphisms associated with these declarations are built in to the language—they are not represented in terms of the source language. We require that a program define exactly one state monad named  $K$ , and one resumption monad named  $R$  defined in terms of  $K$ . **Recursion Tightly Controlled, No Mutual Recursion.** Recursive functions may be defined only by explicit use of the fixed point operator *fix*, which only accepts tail-recursive functions. Any function not defined in terms of *fix* is total. Recursion without *fix* is not possible, because the body of a function may not refer to that function’s name, nor to any named function that does not precede it in the source text. Algebraic data types also are not allowed to be recursive. **Simplified Type System.** The type system dispenses entirely with polymorphism and type classes. **No Higher-Order Functions.** The *only* place where the type system allows a higher-order function to be used is as the operand to *fix*. Lambda expressions are only allowed to occur on the right-hand side of a monadic “bind” operator, or as the operand of *fix*. **Simplified Pattern Matching.** Pattern matching is limited to deconstructing algebraic data types and tuples, and may occur only in the context of a *case* expression. Patterns may not be nested.

**Monad Declarations.** CT provides built-in support for constructing monads from the standard state monad transformer *StateT* [28], as well as the resumption monad transformer *ResT* [34, 12] for concurrency. Monads are specified by a

```
monad K = StateT(Int) Reg1 + StateT(Int) Reg2 + StateT(Bool) Flag
monad R = ResT K
```

$get_{Reg1} : K \text{ Int}$	$get_{Reg2} : K \text{ Int}$	$get_{Flag} : K \text{ Bool}$
$put_{Reg1} : \text{Int} \rightarrow K ()$	$put_{Reg2} : \text{Int} \rightarrow K ()$	$put_{Flag} : \text{Bool} \rightarrow K ()$
$step : K a \rightarrow R a$		

**Fig. 5.** Example monad declarations in Cheap Threads (top). Non-proper morphisms produced by the declarations (bottom). N.b., *step* can be typed at any base type *a*; CT is not polymorphic.

special declaration form, similar to (but less general than) that provided by MonadLab [22]. The example in Figure 5 (top) defines two monads *K* and *R*. Monad *K* is built from three applications of the state monad transformer, reflecting a notion of state comprised of two *Int*-typed registers and one *Bool*-typed flag. Note that *StateT* components must have unique names. Monad *R* applies the resumption transformer to *K*, enabling us to express interleavings of *K*-computations.

These declarations produce “bind” and “unit” operations for the *K* and *R* monads, along with the non-proper morphisms for state and resumptions given in Figure 5 (bottom). It is important to note that these operations are primitive within the CT language; unlike their Haskell counterparts, they are *not* implemented in terms of newtypes, higher-order functions, etc., but instead are handled directly by the compiler.

**Restricting Recursion.** An important design consideration of CT is that it should be implemented in a straightforward manner with loop code. To that end, one important distinction between CT and Haskell is that recursion in CT is strictly limited to tail recursive functions whose codomain is in the resumption monad. Recursive declarations must include explicit applications of the fixed point operator *fix*. This operator takes a function whose type is of the form:

$$(\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow R t) \rightarrow (\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow R t)$$

where  $\tau_1, \tau_2, \dots, \tau_n, t$  are base types (i.e. non-functional and non-monadic), and iterates it. A static check, separate from the type system, enforces the requirement that the iterated function be tail recursive. Algebraic data types also are not allowed to be recursive.

**Type System.** CT is a simply-typed language with primitive types *Int*, *Bool*, and *()*; tuples and non-recursive algebraic data types; function types; and monadic types. Support for higher-order functions is limited. A simply-typed system was chosen over a polymorphic one because it makes it easier to restrict type expressiveness. Because the typing rules are mostly standard, we will discuss only the unusual cases.

Due to the fact that functions are not true first-class values, partial application is not allowed, and the use of higher-order functions is limited to the built-in *fix* operator. These restrictions are expressed by the rule for application:

$$\frac{\Gamma \vdash e_1, e_2, \dots, e_n : \tau_1, \tau_2, \dots, \tau_n \quad \Gamma \vdash f : \tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n \rightarrow t}{\Gamma \vdash f e_1 e_2 \dots e_n : t} \quad (\tau_1, \dots, \tau_n, t \text{ do not contain } \rightarrow)$$

Note that while this rule does not stop the programmer from *defining* higher-order functions, it does preclude the *application* of higher-order functions. We make this distinction rather than excise higher-order functions altogether because *fix* actually does operate on higher-order functions of a certain form.

The monadic “bind” and “unit” operators for each supported monad are built in to the language. Rather than supply a single set of operators overloaded over all monads, the operators are subscripted to make explicit the monad in which they operate. In each of the following rules,  $m$  may stand for a monad ( $K$  or  $R$  in Figure 5 (top)), and  $\tau, \tau'$  must be base types.

$$\frac{\Gamma \vdash \varphi : m \tau \quad \Gamma \vdash f : \tau \rightarrow m \tau'}{\Gamma \vdash \varphi \star_m f : m \tau'} \quad \frac{\Gamma \vdash \varphi : m \tau \quad \Gamma \vdash \varphi' : m \tau'}{\Gamma \vdash \varphi >>_m \varphi' : m \tau'} \\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \eta_m e : m \tau}$$

State and resumption monad operations are also built in, and have analogous types to their Haskell counterparts [28,34]. If the state monad  $K$  has a component  $StateT(\tau) \text{Elem}$ , it is assumed that the tags (e.g., *Elem*) are unique. The state operations  $get_{\text{Elem}}$  and  $put_{\text{Elem}}$  are typed as follows:

$$\frac{}{\Gamma \vdash get_{\text{Elem}} : K \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash put_{\text{Elem}} e : K ()}$$

The *step* and *fix* morphisms of the  $R$  monad are typed as follows:

$$\frac{\Gamma \vdash \varphi : K \tau}{\Gamma \vdash step \varphi : R \tau} \quad \frac{\Gamma \vdash f : (\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow R t) \rightarrow (\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow R t)}{\Gamma \vdash fix f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow R t}$$

where  $\tau_1, \tau_2, \dots, \tau_n, t$  are base types.

## 4 Compiling Cheap Threads

In this section we discuss the compilation of CT to intermediate code. The compiler targets the intermediate language described in Table 1, which we will refer to as the RTL. The RTL is a simple register transfer language. Two types of registers exist: general-purpose virtual registers ( $rn$ ), and “named” registers ( $rs$  where  $s$  is a string beginning with a non-numeric character) which are used to hold global state (i.e. state components of  $K$ ).

**Table 1.** The intermediate language targeted by the compiler

Instruction	Meaning
<i>l</i> :	Label.
<i>r</i> := <i>n</i>	Stores the constant value <i>n</i> in register <i>r</i> .
<i>r</i> 1 := <i>r</i> 2	Copies the value in register <i>r</i> 2 into register <i>r</i> 1.
<i>r</i> 1 := <i>r</i> 2 + <i>r</i> 3	Adds the value stored in <i>r</i> 2 to the value stored in <i>r</i> 3 and stores the result in <i>r</i> 1.
<i>r</i> 1 := <i>r</i> 2 - <i>r</i> 3	Subtracts the value stored in <i>r</i> 3 from the value stored in <i>r</i> 2 and stores the result in <i>r</i> 1.
BZero <i>r</i> <i>l</i>	Jumps to label <i>l</i> if the value stored in <i>r</i> is zero.
Jump <i>l</i>	Jumps to label <i>l</i> .

It is a simple matter to generate instructions for a RISC-like architecture from our intermediate language. We have implemented an instruction selection phase for MicroBlaze, a 32-bit RISC soft processor designed to be run on FPGA fabric. Instruction selection is entirely straightforward, so we omit the details.

#### 4.1 Translation from Cheap Threads

This section describes the compilation of CT to the RTL. The compiler is implemented in Haskell, using the parser from the `haskell-src` library. Compilation proceeds in three passes: type checking (see Section 3), inlining, and code generation.

**Inlining.** In the inlining pass, all calls to non-recursive functions are inlined, starting with *main* at the root of the call tree. The output from the inliner is essentially one “giant term” in which the only function applications that occur are those of recursive functions, and the only variable references that occur are to  $\lambda$ -bound variables, which may appear only in a function passed to *fix* or on the right-hand side of  $\star_m$ .

**Code Generation.** After inlining, we generate code for the resulting term in a syntax-directed fashion. The code generation function *codegen* returns a sequence of RTL commands implementing the source expression, and the register in which those commands store the result, if any. Let *CtExpr* be the data type corresponding to *expr* in Figure 4, and *RtlCom* be the data type representing RTL commands. Then the type of *codegen* is  $CtExpr \rightarrow CM([RtlCom], Register)$ , where *CM* is a monad providing mechanisms for generating fresh registers and labels, and binding variables to registers.

**Notational Convention.** For the sake of brevity, we will describe the translation rules according to the convention that  $\lceil e \rceil$  is the code generated by translating the expression *e*, and anywhere after an occurrence of  $\lceil e \rceil$ , we may refer to the register in which *e*’s result is stored as *r<sub>e</sub>*. That is,  $\lceil e \rceil$  is the list of *RtlComs* returned by *codegen e*, and *r<sub>e</sub>* is the *Register* returned by *codegen e*. We use Haskell-style comments (beginning with a long dash — and extending to the end of a source line), and a comment at the end of the translation rule indicates which register is the result register.

**Compiling Pure Expressions.** We first consider the compilation of pure, that is non-monadic, expressions. If the expression to be compiled is of type  $Int$ ,  $Bool$ , or  $()$ , the result is simply stored in a freshly-generated register. For example, we compile addition as follows:

$$\begin{aligned} \lceil e_1 + e_2 \rceil &= \lceil e_1 \rceil ; \lceil e_2 \rceil ; r := r_{e_1} + r_{e_2} \\ &\quad \text{— Where } r \text{ is fresh. Result register is } r. \end{aligned}$$

Values of type  $()$  have no result register.

**Compiling Monadic Expressions.** The  $\star_m$  and  $\gg_m$  operators are compiled much like function composition. Assume without loss of generality that terms on the right-hand side of  $\star_m$  are always  $\lambda$ -expressions. Then:

$$\begin{aligned} \lceil \varphi \star_m \lambda x. \chi \rceil &= \lceil \varphi \rceil ; \lceil \chi[x \mapsto r_\varphi] \rceil \quad \text{— Result register is } r_\chi. \\ \lceil \varphi \gg_m \varphi' \rceil &= \lceil \varphi \rceil ; \lceil \varphi' \rceil \quad \text{— Result register is } r_{\varphi'}. \end{aligned}$$

where  $\lceil \chi[x \mapsto r_\varphi] \rceil$  denotes that  $\chi$  is compiled in the current environment, extended by binding variable  $x$  to register  $r_\varphi$ .

The “unit” operator  $\eta_m$  serves to inject pure computations into a monad. The resumption monad operator  $step$  serves a similar function, lifting  $K$  computations into  $R$ . For  $step$  we generate a label to delineate “ $step$ -ed” state operations; for the purposes of this paper, these labels are purely informational.

$$\begin{aligned} \lceil \eta_m e \rceil &= \lceil e \rceil \quad \text{— Result register is } r_e. \\ \lceil step \varphi \rceil &= l : \lceil \varphi \rceil \quad \text{— Where label } l \text{ is fresh. Result register is } r_\varphi. \end{aligned}$$

Finally, the state monad morphisms  $get_{Elem}$  and  $put_{Elem}$  simply read or modify the associated register  $\mathbf{r}Elem$ .

$$\begin{aligned} \lceil get_{Elem} \rceil &= r := \mathbf{r}Elem \quad \text{— Where } r \text{ is fresh. Result register is } r. \\ \lceil put_{Elem} e \rceil &= \lceil e \rceil ; \mathbf{r}Elem := r_e \quad \text{— No result register.} \end{aligned}$$

**Compiling  $fix$ .** As we mentioned previously, functions produced by the  $fix$  operator must be tail recursive. This means that we can compile applications of  $fix$  to simple loop code. Let  $f$  be any function satisfying this condition, with parameters named  $\kappa, v_1, v_2, \dots, v_n$ . Call the body of this function  $b$ .

$$\begin{aligned} \lceil (fix f) e_1 e_2 \dots e_n \rceil &= \lceil e_1 \rceil ; \lceil e_2 \rceil ; \dots ; \lceil e_n \rceil \\ l : & \lceil b \rceil \quad \text{— in a special environment—see below} \\ & \text{— Where label } l \text{ is fresh. Result register is } r_b. \end{aligned}$$

Expression  $b$  is compiled in an environment where  $v_1, v_2, \dots, v_n$  are bound to registers  $r_{e_1}, r_{e_2}, \dots, r_{e_n}$ , and in which application of  $\kappa$  is compiled by the following special rule:

$$\lceil \kappa e'_1 \dots e'_n \rceil = \lceil e'_1 \rceil ; \lceil e'_2 \rceil ; \dots ; \lceil e'_n \rceil$$

```

 $r_{e_1} := r_{e'_1}$  ;  $r_{e_2} := r_{e'_2}$  ;  $\dots$  ;  $r_{e_n} := r_{e'_n}$ 
Jump  $l$ 
— No result register.

```

Note that the translation schema given above is slightly simplified in that it produces a separate loop for each (outermost) application of a recursive function, resulting in code duplication. This duplication can easily be avoided by memoizing the label at which  $\lceil b \rceil$  is stored.

**Example.** Figure 6 gives the code generated for the example program in Figure 3. Each monadic operation compiles to only a handful of RTL instructions. The resulting code contains only 17 instructions, which one may reasonably expect, at an extremely conservative estimate, to translate to at most a few hundred machine language instructions on any given architecture.

By comparison, ghc-6.10.1 compiling the same code produces a 650-kilobyte binary when targeting x86 on Windows with optimization enabled (disabling optimization produces a slightly larger binary). We believe that most of this code (432 kilobytes) is accounted for by the Haskell runtime system, and much of the remainder is code for various prelude and library functions. Of course, we do not claim that this comparison provides evidence that the CT compiler is “better” than GHC—obviously, GHC compiles a far more expressive source language, and therefore an increase in code size is inevitable. But the comparison does highlight a major advantage of a directly-compiled DSL over one embedded in Haskell: the code produced by Cheap Threads is several orders of magnitude smaller, making it more suitable for use in embedded systems.

```

-- Init G-save for A and B      12:
r1 := 0
r2 := 0
mainloop:
11:
  -- Restore G for process A
  rG := r1
  -- Execute actionA
  r3 := rG
  r4 := 1
  r5 := r3 + r4
  rG := r5
  r6 := rG
12:
  -- Restore G for process B
  rG := r2
  -- Execute actionB
  r7 := rG
  r8 := 1
  r9 := r7 - r8
  rG := r9
  r10 := rG
  -- Save G vals for next iteration
  r1 := r6
  r2 := r10
  -- Loop
  Jump mainloop

```

Fig. 6. RTL code for the program in Figure 3

## 5 Synthesizing Circuits from Cheap Threads Programs

Producing a circuit from a Cheap Threads program proceeds in two steps. First, the source program is defunctionalized, producing an abstract state machine that is readily formatted in the syntax of the FSMLang DSL. The FSMLang compiler is used to produce VHDL code from which a hardware implementation on an FPGA may be synthesized. Section 5.1 defines the defunctionalization transformation for CT. Section 5.2 describes the design and syntax of FSMLang.

### 5.1 Defunctionalizing Cheap Threads

This section formulates the defunctionalization transformation for CT. The resulting state machine,  $\langle States, Rules \rangle$ , consists of a set of states,  $States$ , and a set of transformation rules,  $Rules$ , of type  $States \rightarrow States$ . Defunctionalization takes place “by layer” in that terms typed in  $K$  are defunctionalized separately from those typed in  $R$ .

**Defunctionalizing Layered State Monads.** The states and rules of the target machine arise directly from the definitions of  $K$  and the source term being transformed, respectively. Let us assume that  $K$  is a state monad of the form,  $K = StateT Reg_1 (\dots (StateT Reg_n I) \dots)$ . The elements of  $States$  are tuples determined by (1) the  $\lambda$ -bound variables within the program, (2) the states  $Reg_i$  within monad  $K$ , and an additional component for the value returned by the computation. Variables bound by  $\lambda$  are assumed without loss of generality to be unique and any  $fix$  bound variables (e.g., the “ $\kappa$ ” in “ $fix(\lambda\kappa.\dots)$ ”) are not considered  $\lambda$ -bound. For the language defined in Section 3, the type of return values will always be one of  $Int$ ,  $Bool$ , or  $()$ ; let type  $Val = Int + Bool + ()$ .

Taking this into consideration, the elements of  $States$  of the target machine have type  $Reg_1 \times \dots \times Reg_n \times Var_1 \times \dots \times Var_m \times Val$ . For each  $\lambda$ -bound variable or state in  $K$ , there is a corresponding component within the  $States$  type. Define  $c$  as the total number of components as:  $c = m + n$ . We define the update and read transitions,  $upd_x$  and  $read_{x_i}$ , as:

$$\begin{aligned} (x_1, \dots, x_c, v) &\mapsto_{upd_x} (x_1, \dots, v, \dots, x_c, v) \\ (x_1, \dots, x_i, \dots, x_c, v) &\mapsto_{read_{x_i}} (x_1, \dots, x_i, \dots, x_c, x_i) \end{aligned}$$

The  $upd_x$  transition sets the  $x$  “slot” to the value component while the  $read_{x_i}$  transition sets the value component to the current contents of  $x_i$ .

Each  $K$  term gives rise to one rule only and is derived in a syntax-directed manner. The get, put and unit operations are straightforward. Below, assume that  $1 \leq i \leq n$  (i.e.,  $put_i$  and  $get_i$  are defined only on components corresponding to  $K$  states) and let  $s = (x_1, \dots, x_c, v)$  be the input state in:

$$K[put_i e] = s \mapsto (x_1, \dots, eval\ e\ s, \dots, x_c, ())$$

$$K[get_i] = read_{x_i}$$

$$K[\eta_K e] = s \mapsto (x_1, \dots, x_c, eval\ e\ s)$$

The unit computation,  $\eta_K e$ , is defunctionalized to a transition that only updates the return value state component to the value of  $e$  in the input state  $s$ ,  $eval\ e\ s$ . The definition of  $eval$  is not given as it is an unremarkable expression interpreter. Note that, by construction, expression  $e$  will be of base type and will only refer to the components corresponding to  $\lambda$ -bound variables.

The bind operations for  $K$ ,  $\star_K$  and  $\gg_K$ , are defined in terms of function composition. The transitions are total endofunctions on system configurations and, therefore, possess the standard notion of function composition.

$$K[\varphi \gg_K \gamma] = K[\gamma] \circ K[\varphi]$$

$$K[\varphi \star_K \lambda x. \gamma] = K[\gamma] \circ upd_x \circ K[\varphi]$$

**Defunctionalizing  $R$ .** This section first describes the defunctionalization of the resumption layer at a high level. Then equations formulating  $R[-]$  are given next. Finally, the results of defunctionalizing the running example are then presented. Defunctionalizing an  $R$  computation produces a state machine whose state type includes one more component than does  $K$ 's:

$$PC \times Reg_1 \times \cdots \times Reg_n \times Var_1 \times \cdots \times Var_m \times Val \quad \text{where } PC = Int$$

This additional component may be thought of as a “program counter” represented as an  $Int$ . The resulting state machine also includes multiple transitions.

*High-level overview.* Whereas layered state adds functionality to CT languages in the form of *put* and *get* operations, the resumption layer adds control mechanisms in the form of *step* and *fix*. Roughly speaking, an  $R$ -computation is a sequence of  $K$ -computations chained together by  $\star_R$  or  $\gg_R$ ; using only  $\gg_R$  for presentation purposes, an  $R$ -computation looks like:

$$(step \varphi_1) \gg_R \cdots \gg_R (step \varphi_j)$$

Defunctionalizing each  $(step \varphi_i)$  is performed by applying  $K[-]$  to each  $\varphi_i$  and thereby producing a single corresponding rule,  $l_i \mapsto r_i$ . Defunctionalizing  $\gg_R$  in the above makes the control flow explicit by attaching a “program counter” (starting without loss of generality from 1) to states; this produces the set of rules:

$$\{(1, l_1) \mapsto (2, r_1), \dots, (j-1, l_{j-1}) \mapsto (j, r_{j-1})\}$$

Consider now a fixed computation,  $fix(\lambda\kappa.\lambda\bar{x}.\gamma)$ . As it is tail recursive, occurrences of a recursive call,  $\kappa \bar{e} : R Val$ , will be accomplished by a rule that (1) updates the state variables corresponding to  $\lambda$ -bound variables  $\bar{x}$  to the arguments  $\bar{e}$  and (2) changing the program counter component to point to the “beginning” of  $\gamma$ .

*Detailed formulation.* We present the equations defining  $R[-]$  in a Haskell-like notation, suppressing certain representation details for the sake of clarity. These equations are written in terms of a layered monad,  $M$ , containing an integer state for generating labels and an environment mapping recursively bound variables to their formal parameters. The specification of monad  $M$  is given in the MonadLab DSL [22], which allows monads to be specified simply in terms of monad transformers and hides certain technical details (e.g., order of application and lifting of operations through transformers):

$$\begin{aligned} \mathbf{monad} \ M &= EnvT(Bindings) \ Env + StateT(Int) \\ \mathbf{type} \ Bindings &= Var \rightarrow [Var] \\ \mathbf{type} \ Var &= String \end{aligned}$$

MonadLab generates Haskell code defining the first four functions below; the last operator is defined as:  $counter = get \gg= \lambda i. put(i+1) \gg return i$ .

<i>rdEnv</i>	$:: M \text{ Bindings}$	— read current bindings
<i>inEnv</i>	$:: \text{ Bindings} \rightarrow M \text{ } a \rightarrow M \text{ } a$	— resets current bindings
<i>get</i>	$:: M \text{ Int}$	
<i>put</i>	$:: \text{ Int} \rightarrow M \text{ } ()$	
<i>counter</i>	$:: M \text{ Int}$	— gensym-like label generator

The defunctionalization,  $R[e]$ , is a computation of the transitions corresponding to  $e$ . Defunctionalizing a stepped  $K$ -computation first defunctionalizes its argument ( $\varphi$ ), producing a transition,  $l \mapsto r$ . This  $K$ -transition is converted into an  $R$ -transition by adjoining program counter components to both sides;  $(i, (x_1, \dots, x_c, v))$  is identified with  $(i, x_1, \dots, x_c, v)$ . The unit computation,  $(\eta_R e)$ , is translated analogously to *step*:

$$\begin{aligned}
 R[-] &:: \text{CTExpr} \rightarrow M \text{ [Rule]} \\
 R[\text{step } \varphi] &= \text{counter} \gg= \lambda i. \text{return} [(i, l) \mapsto (i+1, r)] \\
 &\quad \text{where } (l \mapsto r) = K[\varphi] \\
 R[\eta_R e] &= \text{counter} \gg= \lambda i. \text{return} [(i, l) \mapsto (i+1, r)] \\
 &\quad \text{where } (l \mapsto r) = K[\eta_K e]
 \end{aligned}$$

To defunctionalize a recursive expression, one first gets the next fresh label ( $i$ ) and reads the current bindings ( $\beta$ ). In an expanded environment ( $\beta'$ ) that binds the recursive variable ( $\kappa$ ) to its formal parameters ( $v_1, \dots, v_m$ ), the body ( $e$ ) is defunctionalized to a list of rules ( $\rho$ ). Assuming there is a unique label associated with  $\kappa$ , called  $L_\kappa$ , the transition list  $\rho$  is augmented with a transition,  $\text{mkstart } \kappa \ i$ , that serves as the “beginning of the loop”; the augmented list is returned:

$$\begin{aligned}
 R[\text{fix}(\lambda \kappa. \lambda v_1. \dots. \lambda v_l. e)] &= \text{get} \gg= \lambda i. \\
 &\quad \text{rdEnv} \gg= \lambda \beta. \\
 &\quad ( \text{inEnv } \beta' R[e] ) \gg= \lambda \rho. \\
 &\quad \text{return} (\text{mkstart } \kappa \ i : \rho) \\
 \text{where } \beta' &= \beta\{\kappa := [v_1, \dots, v_l]\} \\
 \text{mkstart } \kappa \ i &= (L_\kappa, x_1, \dots, x_c, v) \mapsto (i, x_1, \dots, x_c, v)
 \end{aligned}$$

A recursive call is translated to a transition that takes an input state and moves to a state with label  $L_\kappa$  (defined above) and, in effect, this transition jumps to the head of the “loop”. For the sake of simplifying the presentation, the definition below only gives the case where  $\kappa$  has one argument (i.e.,  $\kappa$  has type  $\tau \rightarrow R\tau'$ ); the full definition is analogous. This recursive call occurs within the body of an expression of form,  $\text{fix}(\lambda \kappa. \lambda x. \text{body})$ . In the following, assume that  $x$  is the inner  $\lambda$ -bound variable represented in the state configuration:

$$\begin{aligned}
 R[\kappa e] &= \text{counter} \gg= \lambda i. \\
 &\quad \text{return} [(i, x_1, \dots, x_c, v) \mapsto (L_\kappa, x_1, \dots, \text{eval } e \ s, \dots, x_c, v)] \\
 \text{where } s &= (x_1, \dots, x_c, v)
 \end{aligned}$$

This presentation is simplified also in that the details of looking up  $x$  in the bindings for  $\kappa$  are suppressed. Defining  $R[-]$  for the bind operations:

$$\begin{aligned}
 R[\gamma \gg_R \chi] &= R[\gamma] \gg= \lambda \rho_1. R[\chi] \gg= \lambda \rho_2. \mathbf{return} (\rho_1 \uparrow\downarrow \rho_2) \\
 R[\gamma \star_R \lambda v. \chi] &= R[\gamma] \gg= \lambda [\rho_1]. R[\chi] \gg= \lambda \rho_2. \mathbf{return} (f v \rho_1 : \rho_2) \\
 \mathbf{where} \\
 f &:: \text{Var} \rightarrow \text{Rule} \rightarrow \text{Rule} \\
 f v ((i, s) \mapsto (i', s')) &= ((i, s) \mapsto (i', \text{upd } v s'))
 \end{aligned}$$

**Example.** Returning to the running example presented in Fig. 3; the relevant portion of the channel code is:

$$\begin{aligned}
 \text{chan} &:: \text{Reg}_1 \rightarrow \text{Reg}_2 \rightarrow R() \\
 \text{chan} &= \text{fix } (\lambda k. \lambda a. \lambda b. \\
 &\quad \text{step } (\text{put}_G a \gg_K \text{actionA} \gg_K \text{get}_G) \star_R \lambda \text{newa}. \\
 &\quad \text{step } (\text{put}_G b \gg_K \text{actionB} \gg_K \text{get}_G) \star_R \lambda \text{newb}. \\
 &\quad k \text{ newa newb })
 \end{aligned}$$

Multiple declarations can be easily accommodated, but rather than elaborate on such details, assume that the two actions stand for particular transitions:

$$\begin{aligned}
 (a, b, \text{newa}, \text{newb}, \text{reg}, \text{val}) \mapsto (a, b, \text{newa}, \text{newb}, \text{reg}+1, ()) &\quad \text{— actionA} \\
 (a, b, \text{newa}, \text{newb}, \text{reg}, \text{val}) \mapsto (a, b, \text{newa}, \text{newb}, \text{reg}-1, ()) &\quad \text{— actionB}
 \end{aligned}$$

The transitions of the state machine produced by defunctionalization are then:

```

initial_state = k
(k,a,b,newa,newb,r,v) -> (1,a,b,newa,newb,r,v)
(1,a,b,newa,newb,r,v) -> (2,a,b,a+1,newb,a+1,a+1)
(2,a,b,newa,newb,r,v) -> (3,a,b,newa,b-1,b-1,b-1)
(3,a,b,newa,newb,r,v) -> (k,newa,newb,newa,newb,r,v)

```

These transitions may be easily reformatted in FSMLang syntax:

```

initial_state = state_k
state_k -> state_1 where
{ }

state_1 -> state_2 where
{ newa' <= a+1;
  r'     <= a+1;
  v'     <= a+1; }

state_2 -> state_3 where
{ newb' <= b-1;
  r'     <= b-1;
  v'     <= b-1; }

state_3 -> state_k where
{ a' <= newa;
  b' <= newb; }

```

This constitutes the **TRANS** section of an FSMLang implementation. The full version includes headers defining a number of initial conditions (e.g., values and sizes of registers, etc.). FSMLang syntax is defined below.

## 5.2 Overview of FSMLang

FSMLang is a domain-specific language (DSL) for describing finite-state machines (FSMs) [3]. FSMLang targets the configurable logic, embedded memories, and soft-core processors that can be found in modern platform Xilinx FPGAs. FSMLang eliminates the need for a programmer to manually control sensitivity lists, state enumerations, FSM reset behavior, and FSM default output behavior. FSMLang descriptions are much smaller, and less cluttered, than equivalent code written in an HDL. Additionally, the FSMLang compiler is re-targetable – while we focus here on producing hardware, it is also capable of producing FSM implementations in software.

The structure and syntax of an FSMLang program is shown in Figure 7. The TRANS section defines the transitions of the state machine. The defunctionalized Cheap Threads program is formatted directly into FSMLang and inserted into the TRANS section of a template with the other sections defined.

<pre>-- Internal state signal names CS: &lt;current_state_signal_name&gt;; NS: &lt;next_state_signal_name&gt;;</pre> <pre>-- Compile-time variables GENERICs:   (&lt;genName&gt;, &lt;type&gt;, &lt;static_value&gt;;)*</pre> <pre>-- Definitions of input/output ports PORTS:   (&lt;portName&gt;, &lt;in out&gt;, &lt;type&gt;;)* CONNECTIONS:   (&lt;outputPortName&gt;  &lt;=  &lt;rhs&gt;;)*</pre> <pre>-- Definitions of memories MEMS:   (&lt;mName&gt;,   &lt;dataWidth&gt;, &lt;addrWidth&gt; [,EXTERNAL];)*</pre>	<pre>-- Definitions of FIFO channels CHANNELS:   (&lt;channelName&gt;, &lt;dataWidth&gt;;)*</pre> <pre>-- Internal FSM signals SIGS:   (&lt;sigName&gt;, &lt;type&gt;;)*</pre> <pre>-- Definition of logic/transitions INITIAL:   &lt;stateName&gt;; TRANS:   (   &lt;curr_st&gt; [   &lt;bool_guard&gt; ] -&gt; &lt;next_st&gt;   [ where { (&lt;lhs&gt;  &lt;=  &lt;rhs&gt;;)* } ]   )*</pre> <pre>-- Native VHDL Definitions VHDL: &lt;un-parsed VHDL code&gt;</pre>
--	--

Fig. 7. FSMLang Program Structure and Syntax

## 6 Application: Huffman Encoding

Cheap Threads serves as a basis for generating efficient software and hardware implementations from a monadic model. The ability to generate both hardware and software components of a complete system is of major benefit when constructing embedded systems because it allows performance-critical elements to be compiled into hardware, capitalizing on the implicit parallelism that that fabric provides, while at the same time allowing less performance-critical components to standard microprocessors.

The Computer Systems Design Laboratory at the University of Kansas has used monad compilation for the implementation of software-defined radios. In contrast to a traditional radio, which generally consists of a series of analog and digital hardware components to implement a specific type of radio (called a

waveform in the nomenclature of the domain), a software-defined radio (SDR) uses the flexibility of the (traditionally software) platform to allow the radio to be reconfigured to support a variety of different waveforms, as application requirements demand [30].

A typical software defined radio will include a variety of components performing digital signal processing such as modulation, spreading, error correction, compression, and encryption. As an example component, Figure 8 shows the definition of a simple Huffman decoder component. Huffman compression is a simple form of data compression which encodes fixed-sized data into stream of variable-sized symbols, and decoding performs the inverse operation. For example, in the decoder definition, the component converts a stream of bits into a stream of integers. The number of bits needed to represent an integer on the input stream varies depending on the frequency that a particular integer occurs in the original (uncompressed) text. This allows integers that occur relatively frequently to be encoded with fewer bits, which in turn reduces the number of bits that must be transmitted.

```

data Node = Emit Int | Branch Int Int
data BMsg = BRead | BWrite Bit | BVal Bit
data IMsg = IRead | IWrite Int | IVal Int
data Bit = Low | High

decoder pos tree input output =
  get tree pos  $\star_R$   $\lambda$ val.
  case val of
    Emit v  $\rightarrow$  signal output (IWrite v)  $\gg_R$  decoder 0 tree input output
    Branch l r  $\rightarrow$  signal input BRead  $\star_R$   $\lambda$ i.
    case i of
      BVal bit  $\rightarrow$  case bit of
        Low  $\rightarrow$  decoder l tree input output
        High  $\rightarrow$  decoder r tree input output
  
```

**Fig. 8.** Huffman Decoder. In this example, the *fix* is implicit and the *R* monad includes another operator, *signal*. See the text for further description.

The decoder uses a tree to represent the encoding of integers. The path from the root of the tree to a leaf identifies the encoding of the integer stored at that leaf. Cheap Threads does not allow recursive data types, so a tree is represented using a state monad, where each value in the state is a *Node*. A *Node* can either be *Emit*, which indicates an integer value, or *Branch*, which indicates an additional bit is needed to encode the values in the sub-trees. The two fields for the *Branch* constructor represent the addresses of the left and right child nodes in the state. To generate the encoding for a node, follow the path from the root of the tree (at address 0). At each *Branch*, if the path enters the left child, then generate a 0 (*Low*) bit, and if the path enters the right child, generate a 1 (*High*) bit.

For example, consider the Huffman tree representing the encoding for arbitrary integers a, b, and c. Given an encoding with a as “0”, b as “10”, and c as “11”, the associated tree written using a recursive tree structure is *Branch* (*Emit a*) (*Branch* (*Emit b*) (*Emit c*)). The non-recursive store representation for this tree using the *Node* structure, mapping integer addresses to *Node* values, is  $0 \mapsto \text{Node } 1$ ,  $2 \mapsto \text{Emit } a$ ,  $3 \mapsto \text{Node } 3$ ,  $4 \mapsto \text{Emit } b$ ,  $5 \mapsto \text{Emit } c$ .

The decoder component in Figure 8 takes advantage of an additional monadic construct, called *signal*, which allows isolated concurrent computations to communicate using a message passing scheme. The semantics of *signal* have been described in detail elsewhere [12], as has the compilation of the construct to VHDL and C [23]. In simple terms, *signal* takes request and passes it to an external entity which interprets the request and generates a response. The computation that generates the request is blocked until a response is generated. As such, the construct models an alternative form of concurrency based on message passing, rather than shared state.

In the implementation of the Huffman decoder, the *signal* construct is used to model the consumption of values from an input stream, one at a time, and to generate values on an output stream. This behavior replaces the common use of lazy lists in languages such as Haskell to model infinite streams, defining the stream transformation notion of computation in the monadic model rather than in the host language.

The decoder function receives the encoded stream one bit at a time. The *pos* argument to the function is used to track the current position of the decoder as an address in the state, identified by the *tree* parameter. If that address contains an *Emit* value, then the decoder sends that value, using the *signal* construct, on the output stream. Alternatively, if the current position contains a *Branch* node, then the decoder will read a value from the input stream, again using the *signal* construct. The decoder then makes a tail call, using the value of the input bit to determine the value of the *pos* parameter.

Having defined the Huffman decoder, the component can be compiled to either a hardware or a software implementation and then integrated into the overall desired radio waveform. Moreover, the monadic representation of the component provides a sound basis for constructing an assurance argument for the correctness of the component, a key capability in the software defined radio domain. Compiling the monadic model directly to an executable implementation eliminates the gap between the model used for verification and the resulting implementation.

## 7 Conclusions

This article presents a foundation for the model-driven engineering of concurrent systems based in semantics-directed compilation. The main vehicle for this approach is the Cheap Threads domain-specific language which encapsulates shared-state concurrency with direct support for state and resumption monads. Cheap Threads may be compiled in a straightforward manner to either a fixed

instruction set or to hardware. The defunctionalization program transformation, furthermore, seems to be well-suited to the generation of hardware implementations of Cheap Threads programs.

Defunctionalization is a well-known technique that remained little used until recently. According to Danvy and Nielsen [8], “compared to closure conversion and to combinator conversion, defunctionalization has been used very little.” We believe that hardware synthesis from a declarative language is an attractive and practical use-case for defunctionalization. The growing prevalence of FPGA and reconfigurable computing technologies means that expressive source languages for mixed hardware/software systems will only become more important, and this article provides evidence that the state machines produced by defunctionalization are a natural fit for hardware synthesis.

A substantial case study of the compilation of monadic programs to hardware and software has been presented as well, in the form of a Huffman decoder for a software-defined radio. Other substantial examples may be found in Kimmell [23]. These case studies indicate that the approach to MDE described here seems to “scale up” and other applications (esp., monadic separation kernels) are currently under development. Monadic modeling provides a formal basis for verifying high assurance properties of the targeted artifacts. It is expected that the semantics-directed basis will yield benefits in this regard, although formal verification has been left for future work.

**Future Work.** Hardware/software co-design presents system designers with a continuum: at one end, a system may be implemented entirely in software, and at the other end, it may be implemented entirely in hardware. The present work has explored the extrema of this continuum. An important question for future research is: can we compile some parts of a Cheap Threads program to hardware, and other parts to software, in an efficient manner? One challenge here is in devising an efficient interface for communication between hardware and software logic. Previous work on the Hybridthreads project [4] at the University of Kansas and the University of Arkansas has explored this problem, using C and the POSIX threads API for mixed-target synthesis.

A particularly interesting follow-on revisits a classic paper by Wand [42] which describes a method for deriving abstract machine instruction sets from source language semantics. Modern FPGA technology may enable us to apply similar techniques to synthesize soft processors that directly implement such a derived instruction set. The defunctionalization-based techniques may well shed new light on this classic research.

The High Assurance Security Kernel (HASK) Lab at the University of Missouri is exploring the use of the techniques described here to implement formally verifiable monadic security kernels [15, 14] in hardware and software. The Cheap Threads language, when extended with support for system calls, protected memory, and asynchronous exceptions [33, 13], will provide a separation kernel modeling language with a clear semantics that supports formal verification.

## References

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2003), pp. 8–19. ACM Press, New York (2003)
2. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science* 342(1), 149–172 (2005); Extended version available as BRICS technical report RS-4-28
3. Agron, J.: Domain-specific language for HW/SW co-design for FPGAs. In: Taha, W.M. (ed.) *DSL 2009*. LNCS, vol. 5658, pp. 262–284. Springer, Heidelberg (2009)
4. Andrews, D., Peck, W., Agron, J., Preston, K., Komp, E., Finley, M., Sass, R.: hthreads: a hardware/software co-designed multithreaded RTOS kernel. In: Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005) (September 2005)
5. Appel, A.W.: *Compiling with Continuations*. Cambridge University Press, Cambridge (1992)
6. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), New York, NY, USA, pp. 174–184 (1998)
7. Dai, J., Huang, B., Li, L., Harrison, L.: Automatically partitioning packet processing applications for pipelined architectures. *SIGPLAN Notices* 40(6), 237–248 (2005)
8. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM International Conference on Principles and Practice of Declarative Programming (PPDP 2001), pp. 162–174 (2001)
9. Filinski, A.: Representing layered monads. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999), pp. 175–188. ACM Press, New York (1999)
10. Hammond, K., Michaelson, G.: Hume: A domain-specific language for real-time embedded systems. In: Pfenning, F., Smaragdakis, Y. (eds.) *GPCE 2003*. LNCS, vol. 2830, pp. 37–56. Springer, Heidelberg (2003)
11. Harrison, W.: A simple semantics for polymorphic recursion. In: Yi, K. (ed.) *APLAS 2005*. LNCS, vol. 3780, pp. 37–51. Springer, Heidelberg (2005)
12. Harrison, W.: The essence of multitasking. In: Johnson, M., Vene, V. (eds.) *AMAST 2006*. LNCS, vol. 4019, pp. 158–172. Springer, Heidelberg (2006)
13. Harrison, W., Allwein, G., Gill, A., Procter, A.: Asynchronous exceptions as an effect. In: Audebaud, P., Paulin-Mohring, C. (eds.) *MPC 2008*. LNCS, vol. 5133, pp. 153–176. Springer, Heidelberg (2008)
14. Harrison, W., Hook, J.: Achieving information flow security through precise control of effects. In: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW 2005), Aix-en-Provence, France, June 2005, pp. 16–30 (2005)
15. Harrison, W., Hook, J.: Achieving information flow security through monadic control of effects. *Journal of Computer Security* (invited submission), 51 (2008) (in press); Extends [14]
16. Harrison, W., Kamin, S.: Metacomputation-based compiler architecture. In: Backhouse, R., Oliveira, J.N. (eds.) *MPC 2000*. LNCS, vol. 1837, pp. 213–229. Springer, Heidelberg (2000)

17. Harrison, W., Kieburtz, R.: The logic of demand in Haskell. *Journal of Functional Programming* 15(5), 837–891 (2005)
18. Harrison, W., Procter, A.: Cheap (but functional) threads. *Higher-Order and Symbolic Computation*, 44 (submitted for publication); extends [12]
19. Hutton, G., Wright, J.: Compiling exceptions correctly. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 211–227. Springer, Heidelberg (2004)
20. Hutton, G., Wright, J.: Calculating an exceptional machine. In: Loidl, H.-W. (ed.) Trends in Functional Programming, February 2006, vol. 5 (2006)
21. Johann, P., Voigtländer, J.: Free theorems in the presence of *seq*. In: Proceedings of the 31st ACM SIGPLAN Symposium on Principles of Programming Languages, January 2004, pp. 99–110 (2004)
22. Kariotis, P., Procter, A., Harrison, W.: Making monads first-class with Template Haskell. In: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell 2008), pp. 99–110 (2008)
23. Kimmell, G.: System Synthesis from a Monadic Functional Language. PhD thesis, University of Kansas (2008)
24. Lawall, J., Muller, G., Duchesne, H.: Language design for implementing process scheduling hierarchies. In: Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM 2004), August 2004, pp. 80–91 (2004)
25. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: TinyOS: An operating system for wireless sensor networks. In: Ambient Intelligence. Springer, Heidelberg (2005)
26. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2006), pp. 189–199 (2007)
27. Liang, S., Hudak, P.: Modular denotational semantics for compiler construction. In: Riis Nielson, H. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 219–234. Springer, Heidelberg (1996)
28. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995), pp. 333–343. ACM Press, New York (1995)
29. Matthews, J., Cook, B., Launchbury, J.: Microprocessor Specification in Hawk. In: Proc. of the Intl. Conf. on Computer Languages (ICCL 1998), pp. 90–101 (1998)
30. Minden, G.J., Evans, J.B., Searl, L., DePardo, D., Petty, V.R., Rajbanshi, R., Newman, T., Chen, Q., Weidling, F., Guffey, J., Datla, D., Barker, B., Peck, M., Cordill, B., Wyglinski, A.M., Agah, A.: KUAR: A Flexible Software-Defined Radio Development Platform. In: 2nd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN), Dublin, Ireland (April 2007)
31. Moggi, E.: An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ. (1990)
32. Mycroft, A., Sharp, R.: A statically allocated parallel functional language. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 37–48. Springer, Heidelberg (2000)
33. Palsberg, J., Ma, D.: A typed interrupt calculus. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 291–310. Springer, Heidelberg (2002)
34. Papaspyrou, N.: A resumption monad transformer and its applications in the semantics of Concurrency. In: Proceedings of the 3rd Panhellenic Logic Symposium (2001); An expanded technical report is available from the author by request

35. Paulson, L.: A semantics-directed compiler generator. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982), pp. 224–233 (1982)
36. Peyton Jones, S. (ed.): Haskell 1998 Language and Libraries, Revised Report. Cambridge Univ. Press, Cambridge (2003)
37. Reynolds, J.: Definitional interpreters for higher order programming languages. In: ACM Conference Proceedings, pp. 717–740 (1972)
38. Roper, M., Olsson, R.: Developing embedded multi-threaded applications with CATAPULTS, a domain-specific language for generating thread schedulers. In: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2005), pp. 295–303 (2005)
39. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2) (February 2006)
40. Sharp, R., Mycroft, A.: The FLaSh compiler: efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Research (2000)
41. Sheard, T., Benaiissa, Z., Pasalic, E.: DSL implementation using staging and monads. In: Proceedings of the 2nd Conference on Domain-Specific Languages, Berkeley, CA, October 3–5, pp. 81–94. USENIX Association (1999)
42. Wand, M.: Semantics-directed machine architecture. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 1982), pp. 234–241. ACM Press, New York (1982)

# A MuDDy Experience—ML Bindings to a BDD Library

Ken Friis Larsen\*

Department of Computer Science, University of Copenhagen  
Njalsgade 132, DK-2300 Copenhagen S, Denmark  
`kflarsen@diku.dk`

**Abstract.** Binary Decision Diagrams (BDDs) are a data structure used to efficiently represent boolean expressions on canonical form. BDDs are often the core data structure in model checkers. MuDDy is an ML interface (both for Standard ML and Objective Caml) to the BDD package BuDDy that is written in C. This combination of an ML interface to a high-performance C library is surprisingly fruitful. ML allows you to quickly experiment with high-level symbolic algorithms before handing over the grunt work to the C library. I show how, with a relatively little effort, you can make a domain specific language for concurrent finite state-machines embedded in Standard ML and then write various custom model-checking algorithms for this domain specific embedded language (DSEL).

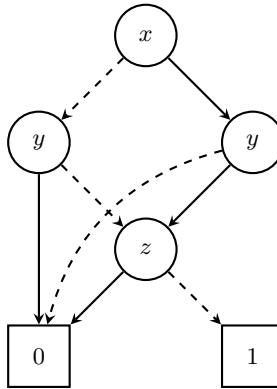
## 1 Introduction

The aim of this paper is twofold. Firstly, I show how a simple domain specific language for specifying concurrent finite state machines can be embedded in Standard ML. The combination of a high-level functional language and a BDD package allows us to make some lightweight experiments with model checking algorithms: lightweight in the sense that the approach has a low start-up cost in terms of programmer time. Furthermore, I show that there is no loss of performance using ML compared to using C or C++.

Secondly, I would like to advocate the data structure *binary decision diagrams* (BDDs). BDDs are used to efficiently represent boolean expression on *canonical form*. The canonical form makes it possible to check whether two expressions are equivalent in constant time. Therefore, BDDs are often a core data structure in model checkers and similar tool. BDDs are naturally presented as a persistent abstract data type, hence it is a good fit for functional programming languages. Because first order logic is generally useful as a symbolic modelling language BDDs can also be used for applications other than model checking such as, for instance, program analysis. However, in this paper I concentrate on examples of simple model checking algorithms. To work with BDDs I use our ML interface,

---

\* Supported by the Danish National Advanced Technology Foundation through the 3gERP project.



**Fig. 1.** Example BDD representing  $(x \leftrightarrow y) \wedge \neg z$ , with variable ordering  $x < y < z$ . Dashed edges represent the case where the variable in a node is false and full edges represent the case where the variable is true. The two special nodes 0 (zero) and 1 (one) represents false and true respectively.

MuDDy, to the BDD C library called BuDDy. MuDDy contains both a Standard ML interface (for MLton and Moscow ML) and an Objective Caml interface. Because MuDDy contains both a Standard ML and an Objective Caml interface, I use “ML” when it does not matter which dialect is used.

The remainder of the paper is structured as follows, Section 2 contains a short introduction to BDDs, Section 3 introduces the domain specific language SGCL and shows how it can be embedded in Standard ML, Section 4 shows how to implement some model check algorithms for SGCL, Section 5 compares the performance of a reachable state-space computation programmed in Standard ML, Objective Caml, C++, and C. Finally, Section 6 reviews related work and Section 7 concludes.

## 2 Binary Decision Diagrams

BDDs are *directed acyclic graphs* (DAGs) used to represent boolean expressions[1]. The main feature of BDDs is that they provide constant time equivalence testing of boolean expressions, and for that reason BDDs are often used in model checkers and theorem provers. The expressions,  $exp$ , represented by BDDs can be described by the following grammar:

$$\begin{aligned} exp ::= & \text{true} \mid \text{false} \mid x \mid exp_1 \text{ con } exp_2 \mid \neg exp \\ & \mid \exists(x_1, x_2, \dots, x_n) \ exp \mid \forall(x_1, x_2, \dots, x_n) \ exp \end{aligned}$$

where  $con$  is one of the usual boolean connectives, and  $x$  ranges over boolean variables, with one unusual requirement: there is a total order of all variables.

Figure 1 shows a graphical representation of a BDD. The important thing to note is that nodes that represent semantically equivalent boolean expressions are always shared.

```

signature bdd =
sig
  type bdd
  type varnum = int
  val TRUE      : bdd
  val FALSE     : bdd
  val ithvar   : varnum -> bdd
  val OR        : bdd * bdd -> bdd
  val AND       : bdd * bdd -> bdd
  val IMP       : bdd * bdd -> bdd
  type varSet
  val makeset  : varnum list -> varSet
  val exist    : varSet -> bdd -> bdd
  val forall   : varSet -> bdd -> bdd
  val equal    : bdd -> bdd -> bool
  ...
end
structure bdd :> bdd = ...

```

**Fig. 2.** Cut-down version of the Standard ML module `bdd` from MuDDy

## 2.1 The MuDDy Interface

Figure 2 shows a cut-down version of the Standard ML signature `bdd` from MuDDy (for now we shall ignore initialization of the library, see Section 2.2 for more details), the structure `bdd` implements the signature `bdd`<sup>1</sup>. The only surprising thing in this interface is the representation of boolean variables, the type `varnum`, as integers and not strings as you might have expected. Thus, to construct a BDD that represents  $x_i$  (the  $i$ 'th variable in the total order of variables) you call the function `ithvar` with  $i$  as argument.

Using this interface it is now straight forward to build a BDD, `b`, corresponding to the tautology  $((x_0 \Rightarrow x_1) \wedge x_0) \Rightarrow x_1$  (assuming that the `bdd` structure has been opened, initialised, and the proper infix declarations have been made):

```

val (x0, x1) = (ithvar 0, ithvar 1)
val b = ((x0 IMP x1) AND x0) IMP x1

```

We can now check that `b` is indeed a tautology. That is, `b` is to (the BDD) `TRUE`:

```

- equal b TRUE;
> val it = true : bool

```

## 2.2 Motivation and Background for MuDDy

Why does it make sense to interface to a BDD package implemented in C rather than write a BDD package directly in ML? The obvious reason is reuse of code.

<sup>1</sup> It is the standard Moscow ML and Objective Caml convention that a structure `A` implements signature `A` (opaque matching) I use that convention throughout the article.

When somebody already has implemented a good library then there is no reason not to reuse it (if possible).

Secondly, high performance is a necessity (if you want to handle problems of an interesting size) because BDDs are used to tackle a NP complete problem. Even a tiny constant factor has a great impact if it is in an operation that is performed exponentially many times. This is the reason why it is hard to write a high performing BDD package in a high-level programming language.

BDDs are *directed acyclic graphs* where all nodes have the same size. Thus, lots of performance can be gained by implementing custom memory management routines for BDD nodes in C.

The three main advantages of the ML wrapping are:

- The garbage collector in ML takes care of the reference counting interface to BuDDy. Thus, freeing the programmer from this burden (see Section 5.2 to see why this is important).
- Compared to the C interface, the ML API is more strongly typed, which makes it impossible to, say, confuse a set of variables with a BDD (which is represented by the same data structure in BuDDy).
- The interactive environment makes it useful in education, because the students can easily work interactively with BDDs.

The first version of MuDDy was written to support state-of-the-art BDDs in the interactive theorem-prover HOL [2]. This usage has influenced many of decisions about how the interface should be, in particular the design principle that there should be as small an overhead as possible for using MuDDy. Thus, convenience functions such as, for instance, mapping variable numbers to human-readable strings should be implemented outside of the library.

### 3 Small Guarded Command Language

This section introduces Small Guarded Command Language (SGCL) as an example of a small yet somewhat realistic example of domain specific language (DSL) that can be used for describing concurrent finite state-machines.

Figure 3 shows the grammar for SGCL. An SGCL program consists of a set of boolean variables, an initialisation and a set of concurrent guarded multi-assignments.

```


$$\begin{array}{ll}
e & ::= \text{true} \mid \text{false} \mid x \mid e_1 \text{ op } e_2 \mid \neg e \\
\text{assignment} & ::= x_1, \dots, x_n := e_1, \dots, e_n \\
\text{command} & ::= e ? \text{assignment} \\
\text{program} & ::= \text{assignment} \\
& \qquad \qquad \qquad \text{command}_1 \parallel \dots \parallel \text{command}_n
\end{array}$$


```

**Fig. 3.** Grammar for SGCL

```

type var = string
datatype boolop = AND | OR | IMP | BIIMP
datatype bexp = BVar of var
  | BBin of bexp * boolop * bexp
  | NOT of bexp
  | TRUE | FALSE
datatype command = CMD of bexp * (var * bexp) list
datatype program = PRG of (var * bexp) list * command list

```

Fig. 4. Abstract syntax for SGCL in Standard ML

```

fun mkBBin opr (x, y) = BBin(x, opr, y)
infix /\ \/ ==> <==>
val (op /\, op \/, op ==>, op <==>) =
  (mkBBin AND, mkBBin OR, mkBBin IMP, mkBBin BIIMP)
infix ::=
val op ::= = ListPair.zip
infix ?
fun g ? ass = [CMD(g, ass)]
infix ||
val op|| = op@
val \$ = BVar

```

Fig. 5. Syntactic embedding of SGCL in Standard ML

### 3.1 SGCL Embedded in Standard ML

To embed SGCL in Standard ML we need two parts: a representation of the abstract syntax of SGCL and an embedding of a concrete syntax for SGML. Figure 4 shows the straightforward representation of abstract syntax trees for SGCL in Standard ML.

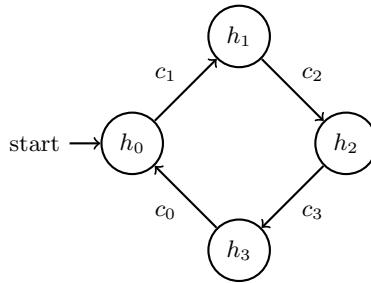
For the syntactic embedding we make heavy use of Standard ML’s `infix` declaration. Figure 5 shows the few declarations we need to make to get a much nicer syntax for SGCL rather than the raw abstract syntax trees. The biggest wart is that we need to distinguish between variables occurring on the left-hand side of an assignment and variables occurring in expressions. We use `$` to lift a variable to an expression.

Thus, if we have a command that monitors when two variable,  $v_1$  and  $v_2$ , are both true and then set them both to false, and a command that does the reverse:

$$\begin{aligned}
 & v_1 \wedge v_2 ? v_1, v_2 := \text{false, false} \\
 \text{|| } & \neg v_1 \wedge \neg v_2 ? v_1, v_2 := \text{true, true}
 \end{aligned}$$

then these two commands can now be written as the following Standard ML expression:

$$\begin{aligned}
 & (\$"v1" /\ \$"v2") ? ([v1, v2] ::= [TRUE, TRUE]) \\
 \text{|| } & (\text{NOT}(\$"v1") /\ \text{NOT}(\$"v2")) ? ([v1, v2] ::= [FALSE, FALSE])
 \end{aligned}$$



**Fig. 6.** Milner's Scheduler with four cyclers

Aside from minor warts like the annoying quotation marks and dollar signs, I consider this to be fully acceptable syntax for SGCL programs. It is definitely nicer to read than the L<sup>A</sup>T<sub>E</sub>X encoding of the SGCL program.

### 3.2 Example: Milner's Scheduler

The driving example in this section is Robin Milner's scheduler taken from [3], the modelling is loosely based on the modelling by Reif Andersen found in [4]. Milner's scheduler is a simple protocol:  $N$  cyclers cooperate on starting  $N$  tasks, one at a time. The cyclers use a token to ensure that all tasks are started in order.

Figure 6 shows Milner's Scheduler with four cyclers. Each of the cyclers  $i$  has three Boolean  $c_i$ ,  $h_i$  and  $t_i$ . The Boolean  $c_i$  is used to indicate that the token is ready to be picked up by cycler  $i$ , the Boolean  $h_i$  indicates that cycler  $i$  has picked up the token and is holding the token, and  $t_i$  indicates that task  $i$  is running.

The initial state of the system is that the token is ready to be picked up by cycler 0 and all tasks are stopped. This can be represented by the following predicate:

$$c_0 \wedge \neg h_0 \wedge \neg t_0 \wedge \left( \bigwedge_{i \in 1 \dots (N-1)} \neg c_i \wedge \neg h_i \wedge \neg t_i \right).$$

When a task is not running, and the token is ready, the cycler can pick up the token and start the task. The cycler can then pass on the token to the next cycler, and the task can stop. For a cycler  $i$  this can be described using two guarded commands:

$$\begin{aligned} \text{cycler}_i = & c_i \wedge \neg t_i ? t_i, c_i, h_i := \text{true}, \neg c_i, \text{true} \\ \text{||} \quad & h_i ? c_{i+1 \bmod N}, h_i := \text{true}, \text{false} \end{aligned}$$

Furthermore we need to model the environment that starts and ends task. This is simply done with one command per task that simply monitors when the right  $t$ -variable becomes true and then sets it to false. That is, for task  $i$ :

$$\text{task}_i = t_i ? t_i := \text{false}$$

```

val (c0, t0, ... , t3, h3) = ("c0", "t0", ... , "t3", "h3")

fun cyclers c t h c' =
  ($c /\ NOT($t)) ? ([t, c, h] ::= [TRUE, NOT($c), TRUE])
  || ($h ? ([c', h] ::= [TRUE, FALSE]))

fun task t = $t ? ([t] ::= [FALSE])

val milner4 =
  PRG( [(c0, TRUE), (t0, FALSE), (h0, FALSE), ... ],
    cyclers c0 t0 h0 c1
    || cyclers c1 t1 h1 c2
    || cyclers c2 t2 h2 c3
    || cyclers c3 t3 h3 c0
    || task t0 || task t1
    || task t2 || task t3
  )

```

**Fig. 7.** Standard ML program that show Milner’s scheduler with four cyclers

All of this can be translated to Standard ML in a straightforward manner. Figure 7 shows the Standard ML code required to model Milner’s Scheduler with four cyclers. The code in Figure 7 is hardwired to four cyclers. However, it is trivial to generalise the code so that it generates a scheduler for arbitrary  $N$ .

## 4 Symbolic Execution of SGCL Programs

One thing that we can do with an SGCL program is to execute it symbolically to find the set of all reachable states from the initial state. That is, we want to compute some kind of representation that denotes the set of all states. One choice is to give an explicit enumeration of all the states. However, for an SGCL program with only a moderate number of variables this representation will quickly explode to enormous sizes. Instead we shall use BDDs to represent the set of reachable states and also to perform the symbolic execution.

The first step is to transform an SGCL program into a pair that consists of a BDD that represents the initial state and a BDD that represents the possible transitions from one set of states to the following set of states (post-states). To model these transitions we shall use two BDD variables, one unprimed and one primed, for each SGCL variable. The primed variables are used to model post-states.

Thus, the interesting part is that we must translate each SGCL command to a predicate, represented as a BDD, that relates a set of states to their post-states. This predicate consists of two parts, one for all the unchanged variables, the variables not assigned in the assignment, that just say that the unchanged variables in the post-state are equivalent to their values in the current state. The other part of predicate is for the variables in the assignment. Figure 8

```

(* commandToBDD: (string * {var: bdd.varnum, primed: bdd.varnum}) list
   -> command
   -> bdd.bdd
*)
fun commandToBDD allVars (CMD(guard, assignments)) =
  let val changed = List.map #1 assignments
      val unchanged =
        List.foldl (fn ((v, {var, primed}), res) =>
                     if mem v changed then res
                     else bdd.AND(bdd.BIIMP(bdd.ithvar primed,
                                              bdd.ithvar var),
                                   res))
                    bdd.TRUE allVars
      val assigns =
        List.foldl (fn ((v,be), res) =>
                     bdd.AND(bdd.BIIMP(primed v, bexp be),
                             res))
                    bdd.TRUE assignments
  in bdd.IMP(bexp guard, assigns) end

```

**Fig. 8.** The function `commandToBDD` translates an SGCL command to a BDD. It uses the helper functions `bexp` that translates an SGCL expression to a BDD and `primed` that translates an SGCL variable to the corresponding primed BDD variable.

shows the function `commandToBDD` that takes a mapping, here as a simple list, of all SGCL variables to the corresponding BDD variable numbers and a command and translates it to a BDD.

Once we have defined `commandToBDD` it's straightforward to define a function, `programToBDDs`, that translates an SGCL program to the two desired BDDs:

```

fun programToBDDs allVars (PRG(init, commands)) =
  let val init =
      List.map (fn (v,be) => bdd.BIIMP(var v, bexp be) init
  in (conj init, disj(List.map commandToBDD commands)) end

```

Again I use some helper functions: `var` translates an SGCL variable to the corresponding unprimed BDD variable, `bexp` translates an expression to a BDD, `conj` makes a conjunction of a list of BDDs, and `disj` makes a disjunction of a list of BDDs.

Now we are ready to compute the set of all reachable states for an SGCL program. Figure 9 shows the function `reachable` that computes the set of reachable states, represented as a BDD, given an initial state, `I`, and a predicate, `T`, relating unprimed variables to primed variables. The interesting part is the nested function `loop` that repetitively expands the set of reachable states until a fix-point is found. This kind of algorithm is one of the cornerstones in most model checkers.

Once we have computed the set of reachable states, it is easy to check whether, for instance, they all satisfy a certain invariant. We just check that the BDD

```

fun reachable allVars I T =
  let val renames =
    List.map (fn(_, {var, primed}) => (var, primed)) allVars
  val pairset = bdd.makepairSet renames
  val unprimed = bdd.makeset(List.map #var allVars)

  open bdd infix OR
  fun loop R =
    let val post = appex T R And unprimed
        val next = R OR replace next pairset
        in if equal R next then R else loop next end
  in loop I end

```

**Fig. 9.** Compute the set of all reachable states for a SGCL program

representing the invariant imply the BDD representing the set of reachable states. Thus, if `R` is the BDD representing the set of reachable states and `Inv` is the BDD representing the invariant we want to check, then we compute:

```
val invSatisfied = bdd.IMP Inv R
```

And if `invSatisfied` is equal to `bdd.TRUE` we know that all reachable states satisfies the invariant.

## 5 ML versus C++ versus C

In this section we will compare model checking using MuDDy with more traditional approaches such as using a C++ or C library directly. We will show fragments of code and we will measure the performance of the different alternatives. Again we use Milner’s scheduler as example.

Model checking is generally NP-complete (due to the exponential number of states). However, BDDs seems to be an efficient representation of the predicates used in Milner’s scheduler. In fact, it turns out that the size of the representation of the state space “only” grows polynomially in the number of cyclers rather than exponentially.

We shall compare implementation in C, C++, Standard ML, and Objective Caml. So as to show what modelling with BDDs looks like in the different languages, I show how to construct the BDD for initial state of Milner’s scheduler in C++ and C. Finally, I show running times for the different languages.

### 5.1 C++

In this section we show a C++ implementation using the BuDDy C++ library. The BuDDy C++ library defines a class `bdd` that represent a bdd, similar to `bdd.bdd` in MuDDy. The class defines bunches of smart operators that makes building Boolean expressions quite easy. Furthermore, all reference counting is managed by the constructor and destructor for the `bdd` class.

```

bdd initial_state(bdd* t, bdd* h, bdd* c)
{
    int i;
    bdd I = c[0] & !h[0] & !t[0];

    for(i=1; i<N; i++)
        I &= !c[i] & !h[i] & !t[i];

    return I;
}

```

The code shows that it is possible to embed a DSEL for boolean expression quite nicely in C++.

## 5.2 C

The C code for building the initial state space predicate lacks the constructor/destructor mechanisms for managing the reference counting. Thus, you must manually keep track of calling the reference counting functions. From hard-earned experience, I guarantee that the code is error prone, unreadable and really hard to maintain. The function `bdd_addrref` references a BDD and then returns the BDD and similar for `bdd_delref`.

```

bdd initial_state(bdd* t, bdd* h, bdd* c)
{
    int i;
    bdd I, tmp1, tmp2, tmp3;

    tmp1 = bdd_addrref( bdd_not(h[0]) );
    tmp2 = bdd_addrref( bdd_apply(c[0], tmp1, bddop_and) );
    bdd_delref(tmp1);

    tmp1 = bdd_addrref( bdd_not(t[0]) );
    I = bdd_apply(tmp1, tmp2, bddop_and);
    bdd_delref(tmp1);
    bdd_delref(tmp2);

    for(i=1; i<N; i++)
    {
        bdd_addrref(I);

        tmp1 = bdd_addrref( bdd_not(c[i]) );
        tmp2 = bdd_addrref( bdd_not(h[i]) );

        tmp3 = bdd_addrref( bdd_apply(tmp1, tmp2, bddop_and) );
        bdd_delref(tmp1);

```

```

bdd_delref(tmp2);

tmp1 = bdd_addr( bdd_not(t[i]) );
tmp2 = bdd_addr( bdd_apply(tmp3, tmp1, bddop_and) );
bdd_delref(tmp3);
bdd_delref(tmp1);

tmp1 = bdd_apply(I, tmp2, bddop_and);
bdd_delref(tmp2);
bdd_delref(I);

I = tmp1;
}

return I;
}

```

### 5.3 Performance

In the following we benchmark the ML implementations of Milner’s scheduler against their C and a C++ counterparts. As benchmark we compute the number of reachable states in Milner’s scheduler for a given number of cyclers.

The source programs were obtained as follows: The C and the C++ implementations are very close to example programs provided with the BuDDy library. We have modified the programs slightly to get them as equal as possible and thereafter translated them into Standard ML code and Objective Caml code. (All implementations are included in the (upcoming) MuDDy distribution.)

You will recall that even though the number of reachable states is exponential in the number of schedulers, it turns out that the representation is only polynomial using BDDs.

Table 1 shows the running times for running the programs with different numbers of cyclers. As we can see, the running times for the ML programs are only *slightly* slower than the C and C++ programs. This is expected because most of the time in all of the programs is spent in the BuDDy library. Yet, it

**Table 1.** Milner’s scheduler benchmark. Running time is in seconds, measured on a 1 Ghz PII with ulimit 156 Mb. The C and C++ programs was compiled using gcc 3.3 with flag `-O3`, `ocamlopt` is native-code from Objective Caml 3.04, `ocaml` is byte-code from Objective Caml 3.04, and `sml` is byte-code from Moscow ML 2.01.

No. of cyclers: 50	100	150	200
<code>c</code> :	1.63	4.69	13.66
<code>cxx</code> :	1.66	4.82	13.89
<code>ocamlopt</code> :	1.71	5.04	14.47
<code>ocaml</code> :	1.74	5.15	14.58
<code>sml</code> :	1.76	5.15	15.12

is reassuring to see that we can write high-level algorithms, such as finding the reachable state-space, in ML without being overly penalised.

## 6 Related Work

Combining BDDs and functional languages is not a new idea. For instance, Lifted-FL [5] is a domain-specific, strongly typed, lazy, functional programming language, where BDDs are available as a build-in type. Lifted-FL is used as a combined model-checking and theorem-proving tool at Intel SCL Logic Team. And the successor to Lifted-FL, the verification framework Forte[6], is built around the strongly-typed ML-like language FL. Like in Lifted-FL BDDs are built into the language, and every Boolean object is represented as a BDD.

Nancy Day et al [7] shows how BDDs and SAT solvers can be nicely wrapped in Haskell so that it feels like a native Haskell structure, hiding the mucky details of an underlying C library—similar to how MuDDy wraps BDDs in ML.

Once you have a nice wrapping of BDDs in a high-level language, you can leverage it to solve a variety of problems. Some examples that have used MuDDy are: Sørensen and Secher show how one can efficiently solve configuration problems with BDDs [8] and Sittampalam et al [9] show how BDDs, for instance, can be used for program transformations. In general, many model checking algorithms should be useful for program analysis [10].

## 7 Conclusion

The combination of a high-level modern language that enables DSEL with a powerful data structure like BDDs makes a pleasant environment in which to learn and experiment, for example model checking algorithms and language design. As an experiment it would be easy, for instance, to add priority specifications to SGCL.

The fact that we have not made a new domain-specific language with BDDs built into it, but have merely made an ML library, means that we can leverage all the existing tools and libraries that exist for Standard ML and Objective Caml. For instance, it enabled us to embed SGCL into Standard ML without problems.

An interesting possibility for the future would be to make an F# [11] interface for MuDDy, because F# has recently acquired powerful reflection mechanisms inspired by *reFLect* [12] (which in turn was inspired by FL). Thus, it should be possible to reap more of the advantages from FL while still staying within a general purpose language.

## Acknowledgments and MuDDy History

The first version of MuDDy was written by Ken Friis Larsen while visiting Mike Gordon at Computer Lab. at University of Cambridge (UK), in autumn 1997 and spring 1998. Jakob Lichtenberg then extended MuDDy to cope with the new

BuDDy features: Finite Domain Blocks (fdds) and Boolean Vectors (bvecs). In 2001–2002 we added support for Objective Caml. In 2003, Ooege de Moor at Computer Lab. at University of Oxford sponsored a research visit to Oxford for Ken Friis Larsen which, amongst other things, resulted in the MLton binding.

It should be stressed that MuDDy is only a type safe ML wrapping around Jørn Lind-Nielsen’s great BuDDy package, and that all the ”hard work” is done by the BuDDy package. Jørn Lind-Nielsen has answered lots of BuDDy questions (and BDD questions), and has been willing to change the BuDDy package to make the ML wrappings more easier.

A special thanks should also go to Peter Sestoft, who has answered numerous of questions about the Moscow ML/Caml Light runtime system, and he even audited the C code at one point to help find a bug.

In addition, over the years many MuDDy users have provided bug-reports, bug-fixes, and useful feedback.

## References

1. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3), 293–318 (1992)
2. Norrish, M., Slind, K.: A thread of HOL development. *Computer Journal* 45(1), 37–45 (2002)
3. Milner, R.: *Communication and Concurrency*. Prentice Hall, Englewood Cliffs (1989)
4. Andersen, H.R.: An introduction to binary decision diagrams (1997), <http://www.itu.dk/people/hra/bdd97-abstract.html>
5. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) *TPHOLs 1999*. LNCS, vol. 1690, p. 323. Springer, Heidelberg (1999)
6. Jones, R.B., O’Leary, J.W., Seger, C.J.H., Aagaard, M.D., Melham, T.F.: Practical formal verification in microprocessor design. *IEEE Design & Test of Computers* 18(4), 16–25 (2001)
7. Day, N.A., Launchbury, J., Lewis, J.: Logical abstractions in Haskell. In: *Proceedings of the 1999 Haskell Workshop*, Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28 (October 1999)
8. Sørensen, M.H., Secher, J.P.: From type inference to configuration. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) *The Essence of Computation*. LNCS, vol. 2566, pp. 436–472. Springer, Heidelberg (2002)
9. Sittampalam, G., Moor, O.D., Larsen, K.F.: Incremental execution of transformation specifications. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 26–38. ACM Press, New York (2004)
10. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Levi, G. (ed.) *SAS 1998*. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
11. Syme, D., Granicz, A., Cisternino, A.: *Expert F#*. Apress (2007)
12. Grundy, J., Melham, T., O’Leary, J.: A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming* 16(2), 157–196 (2006)

# Gel: A Generic Extensible Language

Jose Falcon and William R. Cook

Department of Computer Science

University of Texas at Austin

jofalcon@mail.utexas.edu, wcook@cs.utexas.edu

**Abstract.** Both XML and Lisp have demonstrated the utility of generic syntax for expressing tree-structured data. But generic languages do not provide the syntactic richness of custom languages. Generic Extensible Language (Gel) is a rich generic syntax that embodies many of the common syntactic conventions for operators, grouping and lists in widely-used languages. Prefix/infix operators are disambiguated by white-space, so that documents which violate common white-space conventions will not necessarily parse correctly with Gel. With some character replacements and adjusting for mismatch in operator precedence, Gel can extract meaningful structure from typical files in many languages, including Java, Cascading Style Sheets, Smalltalk, and ANTLR grammars. This evaluation shows the expressive power of Gel, not that Gel can be used as a parser for existing languages. Gel is intended to serve as a generic language for creating composable domain-specific languages.

## 1 Introduction

The traditional approach to implementing concrete syntax for a language is to define a custom grammar and a parser to read the language, and possibly a pretty-printer to output or reformat programs. Examples include programming languages, grammars for parser generators, configuration files, CSS styles, and makefiles.

C/Java: `int m(int[] a) { return o.m(2 * a[x++], !done); }`

CSS: `a:link { font-family: courier; color: #FF0000 }`

Smalltalk: `^ o m: 2 * (a at: x inc) n: done not.`

ANTLR: `call : ID '(' (a=e (',', b=e { a.add(b); })* )? ')' ;`

A custom-designed language is generally easy for humans to read and write, although they must learn specialized syntax and lexical conventions for each language. There are many tools for creating custom languages [19,25,17] and also for creating extensions to custom languages [7].

A second approach is to use a generic language that provides a standard concrete syntax representing generic abstract trees. Examples of this approach include XML [9] and Lisp S-Expressions [23]. A custom language can be defined within a generic language as a *subset*: the Lisp programming language is a subset of S-Expressions, XHTML is a subset of all possible XML documents. Krishnamurthi has called this technique *bicameral parsing* [21]. A language designer can choose how to encode high-level concepts

using the generic syntax. For example, if  $x < 3$  then  $\text{print}(x)$  could be represented this way:

Lisp: (if (< x 3) (print x))

XML: <if><test op="lt"><var name="x"/><const>3</const></test><then><call fun="print"><arg>x</arg></call></then></if>

It is easy to embed or compose different languages in one document. Humans only have to learn one set of syntactic conventions. Parsing, pretty-printing, and other tools can be reused.

A major negative of generic languages is that humans generally find them less appealing to read and write than custom languages. Compared to languages with a custom grammar, both Lisp and XML are impoverished syntactically: a few delimiters and simple syntactic forms are repeated many times.

In this paper we present Generic Extensible Language (Gel), a language that embodies many of the common syntactic conventions of popular languages, including C/Java, Smalltalk [15], Cascading Style Sheets (CSS) [22] and some grammar notations [25]. The goal of this research is to define a generic language based on the syntactic conventions that have evolved over the last 40 years.

Gel has a uniform fixed syntax supporting arbitrary prefix, suffix, and infix operators, lists, grouping, keywords, sequences of adjacent expressions, and string interpolation. It has a novel quoting construct to support meta-languages.

Gel is extensible in the sense that XML is extensible. It allows user-defined operators and flexible use of available syntactic forms. Any practical language using Gel would define a subset of the possible syntactic forms, just as a language defined over text is a subset of all possible strings. Defining a language using Gel involves careful selection of operators with appropriate precedences and selection of keywords and appropriate grouping and expression patterns.

We evaluate Gel by analyzing how well the Gel AST corresponds to the AST created by a traditional parser. Ignoring some conflicts in operator precedence, Gel extracts a good representation of the structure of Java programs, CSS styles, Smalltalk programs, and Corba IDL definitions. These examples demonstrate the expressive power of Gel. The goal is not to create actual parsers for these languages, but to use Gel as a standard input format for future domain-specific languages. The benefit is that Gel can easily support embedding one domain-specific language in another; for example, allowing a Java-like language to include CSS-like fragments directly in an expression, without switching to a different parser.

## 2 Introduction to Gel

This section introduces Gel by example. For reference, an informal summary of Gel syntax in EBNF is given in Figure 1. A formal grammar is given in Section 3. Gel expressions include familiar identifiers, numbers, strings which can be combined with binary operators and grouped in the familiar way.

$e := s \mid t \mid (e) \mid [e] \mid \_ \bullet \_ \mid$	Precedence of expressions, highest first
`e	10. symbol, string, group, op
ee	9. quoted expression
e•e	8. sequence/function application
_•e   e•_   _•e•_	7. binary without spaces
e _ e	6. unary prefix and/or suffix
e _•_ e	5. sequence with space
e _? , _? e	4. binary with spaces
e :   :e   :e :   {e}	3. comma list
e _? ; _? e	2. keyword forms and braces
_	1. semicolon list
$\bullet := ' * [ . , ^ ~ , * / % , + - , @ \# , < > , ! = , & ,   , : ? \$ ] ^ {+}$	arbitrary operators
$s := [a-zA-Z0-9] ^ {+}$	symbols
$t := ' r * ' \mid " p * "$	strings
$r := \backslash x X X \mid \backslash u X X X X \mid \backslash [ t n r ' " \$ ] \mid c h a r$	text encoding
$p := \$ s \bullet \circ g * \mid r$	string interpolation
$g := [e] \mid (e) \mid \{p*\}$	interpolation group
$\_ :=$ white-space or begin/end of group/file	

**Fig. 1.** Informal summary of Gel syntax with expression precedence

```
s1 = x * 3 && c == "str"           (&& (= s1 (* x 3)) (== c "str"))
(s1 = (x * 3)) && (c == "str")    (&& (= s1 (* x 3)^0) (== c "str")^0)
{s1 = [x * 3]} && [c == "str"]    (&& (= s1 (* x 3)^0) (== c "str")^0)
```

The expression on the left is input text. The expression on the right is a Lisp-like presentation of the generic abstract syntax tree (GAST) that results from parsing the text on the left using Gel. The grouping symbols are indicated in GAST by a superscript. The GAST notation preserves the complete structure of the input, even though parentheses are often ignored in later semantic processing of expressions. Remember that Gel only specifies syntax; the semantics of these notations are defined by the particular language encoded using Gel.

Gel interprets any contiguous digits, letters, and underscores as *symbols*. As a result, Gel accepts 3F5BA2, 10pt, 3\_D and 10e23 as symbols. The validity of these symbols is determined by the client program using Gel. Handling of more complex floating point formats is discussed in Section 2.4.

The set of operators is not fixed. Instead, operators are constructed like identifiers: any combination of operator symbols is an operator.

```
{1..9} :-> [c == "str"]           (:->(.. 1 9)^0 (== c "str")^0)
```

Several other languages, including Haskell[18], Scala[24] and Smalltalk[15], allow arbitrary infix operators.

The precedence of most operators is defined by their first character as defined in Figure 2. There is a special case for assignment operators [24] which end in [=] and do not start with [=<>], where [=<>] represents the *set* of characters {!,=,<,>}.

precedence	first character	middle	last	description
13	[.]	any	not [=]	dots
12	[^ ~]	any	not [=]	high
11	[* / %]	any	not [=]	multiplicative
10	[+ -]	any	not [=]	additive
9	[@ #]	any	not [=]	middle
8	[<>]	any	any	relational
7	[!=]	any	any	equality
6	[&]	any	not [=]	and
5	[  ]	any	not [=]	or
4	[: ? \$]	any	not [=]	low
3	not [!=<>] if len>1	any	[=]	assignment
2	[, ]	—	—	comma list
1	[; ]	—	—	semicolon list

where any = [^ ~ \* / % + - @ # != <> & | : ? \$ ` ]

**Fig. 2.** Gel precedence table of operator patterns and precedence levels

The comma, semicolon, and grouping characters are called *punctuation* in Gel. Punctuation symbols do not combine with other operators, and are always taken as single characters. Also, white space is always ignored around punctuation, while it is significant around other operators, as described below.

Multiple uses of the same operator are collected together into an n-ary application, so they have no associativity. Different operators with the same precedence level use right-associativity. While the operators resemble the precedence of many languages, they do not match any perfectly. Although Gel can parse Java code, some operators are given the wrong precedence; the goal of Gel is not to create a better Java parser, but to be able to parse Java-like languages generically. Gel does not support ternary operators, but Java's `c ? a : b` operator can be parsed as a combination of binary operators.

`c ? a : b + 2` (`? c (: a (+ b 2))`)

Many programming languages use comma and semicolon to represent lists of identifiers and lists of statements.

<code>{ 2, 3, 5, 7, 13 }</code> <code>one; two; three</code> <code>(a, 1); (a + 1, b + 2)</code> <code>a, 1; a + 1, b + 2</code>	$(, 2 3 5 7 13)^0$ $(; \text{one two three})$ $(; (, a 1)^0 (, (+ a 1) (+ b 2))^0)$ $(; (, a 1) (, (+ a 1) (+ b 2)))$
---	--

Comma has higher precedence than semicolon, and they both have lower precedence than other operators, so the last two examples above are equivalent. An empty object  $\epsilon$  is inserted when list items are missing, even at the end of a list:

<code>a, , b</code> <code>{ a *= b + 1; }</code>	$(, a \epsilon b)$ $(; (*= a (+ b 1)) \epsilon)^0$
---	---

Operators are treated as symbols in situations where they do not make sense as binary (or unary) operations. Comma and semicolon are always treated as operators, unless they are directly enclosed in a group.

<code>ops = (*, +, -, /)</code>	$(= \text{ops} (* + - /)^0)$
<code>others = [(, ) + (;) + (\$)]</code>	$(= \text{others} (+ , ; \$)^0)$

## 2.1 Unary Operators

Any operator (other than comma and semicolon) can be used as a prefix or suffix unary operator, on any expression:

<code>x?, *p++, !done, pat*</code>	$(, [x]? * [p]++ ! [done] [pat]^*)$
------------------------------------	-------------------------------------

In the abstract notation on the right, unary operators have a special notation. For any expression  $x$  and any operators  $\circ$  and  $\star$ , the prefix form is  $\circ[x]$ , the suffix form is  $[x]\star$  and a combined prefix/suffix form is  $\circ[x]\star$ .

The combination of binary and unary operators allows Gel to represent the typical notation for regular expressions which are also used in modern versions of Extended BNF and other notations for patterns.

<code>(a+   b+)?   x*</code>	$(  [(  [a]+ [b]+ )]? [x]^*)$
------------------------------	-------------------------------

Gel does not support compound grouping symbols, although they can be represented by a prefix and/or suffix operator on a standard group.

<code>@[ "a", "b" ]</code>	$@[(, "a" "b")^0]$
<code>=[x, y, z]=</code>	$=[(, x y z)^0]=$
<code>&lt;{ #2342; @:option** }&gt;</code>	$<[(; # [2342] @:[option]** )^0]>$

## 2.2 Sequences

Gel allows *sequences* of expressions that are not separated by an operator. In Haskell, sequences of expressions denote function application. In Smalltalk, a sequence of identifiers following an expression represent postfix unary operators. In both cases sequences have higher precedence than binary operators.

<code>Haskell: f a 3 + g 10</code>	$(+ (\_ f a 3) (\_ g 10))$
<code>Smalltalk: obj size + item max</code>	$(+ (\_ obj size) (\_ item max))$

In the abstract syntax on the right, a sequence  $a$   $b$  is represented by an whitespace operator:  $(\_ a b)$ . It does not matter to Gel that the interpretation of these syntactic forms is completely different in Haskell and in Smalltalk. What matters is that they follow common syntactic conventions.

Java and C do not have explicit sequence operators, but sequences arise in declarations, statements and in some expressions.

<pre>static int f (int x, bool y) if (x &gt; y) { return x; } (String) x == a [i]</pre>	$(\lrcorner \text{ static int } f (\lrcorner \text{ int } x) (\lrcorner \text{ bool } y))^0$ $(\lrcorner \text{ if } (> x y)^0; (\lrcorner \text{ return } x) \epsilon^0)$ $(== (\lrcorner \text{ String}^0 x) (\lrcorner \text{ a } i^0))$
---	---

Sequences are also used in grammars and regular expressions.

<pre>p ::= id   '(' p ')' ('+'   '-')? ('0' .. '9')+</pre>	$(::= p (  \text{id} (\lrcorner 'p')))$ $(\lrcorner [(\lrcorner '+' '-' )^0] ? [(\lrcorner '0' '9')^0] +)$
--	---

Sequences enable Gel to parse compound expressions without any specific information about what the sequence should contain.

## 2.3 Spaces

The combination of arbitrary infix, prefix and suffix operators with sequences of expressions is highly ambiguous. There would not be any reasonable way to parse the following generic grammar without additional syntactic clues:

$e ::= e \text{ op } e \mid \text{op } e \mid e \text{ op} \mid e \text{ e}$

The simple expression  $a + * b$  can be parsed five different ways (assuming + and \* are separate operators). Gel is based on common conventions for formatting expressions, using white spaces, that distinguish these cases. Parsers are traditionally written to ignore white-space, but humans do not ignore it. Gel uses white-space to distinguish three of the interpretations of this expression (here, an equivalent parenthesized version is provided):

$a + * b \equiv a + (*b)$ $a + * b \equiv (a+) * b$ $a + * b \equiv (a+) (*b)$	$(+ a * [b])$ $(^* [a] + b)$ $(\lrcorner [a] + * [b])$
--	--

Two other interpretations require parentheses in Gel:

$(a+) * b \equiv ((a+) *) b$ $a + (*b) \equiv a (+ (* (b)))$	$(\lrcorner [a] + * b)$ $(\lrcorner a + [* [b]])$
---	--

There is one remaining way to include white-space in the expression. It is not clear how this expression should be parsed.

$a + * b$

One option is to make it an error. However, it is similar to a more common situation with a freestanding operator before or after an expression. In this case, Gel interprets the operator as if it were in parentheses:

$* x + 3 \equiv (* x) + 3$ $a [@ 1] \$ \equiv a [(@ 1) (\$)]$ $[+ -, * /] \equiv [(+) (-), (*) (/)]$ $a + * b \equiv a (+) * b$	$(\lrcorner * (+ x 3))$ $(\lrcorner a (\lrcorner @ 1) \$)$ $(, (\lrcorner + -) (\lrcorner * /))$ $(^* (\lrcorner a +) b)$
--	--

The final example illustrates how the ambiguous expression above is covered by this rule. The last operator is taken as a binary operator, while previous operators are parsed

as symbols. The motivation for this choice is to allow Gel to act as a flexible tokenizer. Gel does not reject expressions that might have a meaningful interpretation.

Spaces *are* significant in most languages. For example, in Java `int x` does not mean the same thing as `intx`. Fortran is the only language we know for which spaces are truly optional [1].

The precedence rules for sequences and operators do not allow Java to be parsed perfectly, even using the pretty-printing conventions. In some cases sequences should have higher precedence than comma, and in other cases comma should have higher precedence:

<code>fun(int x, int y)</code> <code>int x, y;</code>	$(\sqcup \text{ fun} (\sqcup (\sqcup \text{ int } x) (\sqcup \text{ int } y))^0)$ $(; (\sqcup \text{ int } x) y) \epsilon)$
--	--

There is no way to parse both `int x, int y` and `int x, y` correctly with generic precedence for sequences and comma. Gel assigns sequence higher precedence than comma so that function headers, as in the first example, could be parsed correctly. This precedence ordering, however, does not correctly parse the second example. We argue that this is not a defect of Gel, but rather an inconsistency in the precedence of the comma operator in Java.

## 2.4 Spaced and Non-spaced Operators

Spacing also affects the interpretation of binary operators in Gel. The examples in Section 2.2 depend on sequences of expressions having higher precedence than binary operators. However, there are other situations in which binary operators should have higher precedence. One example comes from ANTLR grammars, which use an equal sign as a high-precedence binary operator.

```
exp : a=term ('+' b=term) *
```

Does “`a=b c`” parse as  $(= a (\sqcup b c))$  or as  $(\sqcup (= a b) c)$ ? In this case the desired parse is the latter, but this violates the rule that sequences have higher precedence than binary operators. Note that the convention in ANTLR is to have *no white-space* around the `=` operator, in contrast to the convention when formatting assignment operators in Java.

The solution in Gel is to make operators without white space have higher precedence than operators surrounded by white-space, including sequences separated by white-space. This is clearly a controversial decision. It matches conventions in languages as diverse as Haskell and ANTLR, as examples illustrate below. Using this rule, the ANTLR expression parses correctly in Gel:

```
exp : a=term ('+' b=term) *
      (: exp (\sqcup (= a term)^\diamond [(\sqcup '+' (= b term)^\diamond)]^*) ))
```

The general rule is that a chunk of text with no spaces or punctuation is always parsed as a unit, as if it were parenthesized. These chunks are then combined by any operators with spaces. The same precedence rules are applied to non-spaced and spaced operators. Thus white-space acts as an implicit grouping operator, in effect a kind of

parentheses. This idea is represented explicitly in the abstract representation using a  $\langle \rangle$  as a grouping operator.

Unary prefix and suffix operators can only occur at the beginning or end of a chunk, and they always apply to the result of the entire chunk.

$2 * -3.14^20$	$(* 2 -[(^ (. 3 14) 20)])$
$x=A^*   y=B?$	$(  [(= x A)^* [= y B]?])$
$\&a+b+c^*$	$\&[(+ a b c)]^*$

The first example illustrates how the decimal point in floating point numbers is interpreted as a binary operator. Java breaks sequences of operator characters into tokens, but Gel does not. For example, the Gel expression  $x--*++y$  has the binary operator  $--*++$  but in Java it parses as  $(x--) * (++y)$ . In Gel it must be written  $x-- * ++y$ . Java is not completely consistent in this respect, because it fails to parse  $+++++y$ .

A sequence without spaces, which has high precedence than all other operators, can be used for casting, function application and array access in Java. Note that sequence has higher precedence than dot in Gel, but lower precedence in Java.

$f(x, y)[n]$	$(\lrcorner f (, x y)^0 n^0) \langle \rangle$
$(Integer)a.b$	$(. (\lrcorner Integer^0 a) b) \langle \rangle$
$(Integer) a.b$	$(\lrcorner Integer^0 (. a b) \langle \rangle)$
$o.m(a)$	$(. o (\lrcorner m a^0)) \langle \rangle$
$o.m (a)$	$(\lrcorner (. o m) \langle \rangle a^0)$

These examples illustrate how spaces affect the grouping of operators. The punctuation characters (parentheses, brackets, braces, comma and semicolon) are always interpreted the same whether or not they have white-space around them.

Gel can also parse typical email addresses and URLs, although it does not conform to the full specification of either.

$wcook@cs.utexas.edu$	$(@ wcook (. cs utexas edu)) \langle \rangle$
$http://google.com/search?query=Gel&n=1#m$	
$(// http (? (/ (. google com) search) (# (& (= query Gel) (= n 1)) m))) \langle \rangle$	

This example is only meant to be suggestive of the kinds of notations that Gel could parse, in more restricted contexts. The actual email and URL standards [11,2] allow many other characters that would be interpreted as operators in Gel and ruin the parse.

The Haskell period symbol uses a special case of the general rule for spaces and operators. Without spaces, the period between identifiers represents module paths, but with spaces it is a binary operator, as seen in this one-line implementation of the Unix sort command:

```
(sequence . map putStrLn . List.sort . lines) = << getContents
```

Gel parses this Haskell expression correctly:

```
(=<< (. sequence (\lrcorner map putStrLn) (. List sort) \langle \rangle lines)^0 getContents))
```

## 2.5 Keywords and Curly Braces

A keyword is a special identifier often used to indicate a particular syntactic structure. In most languages keywords are reserved words that cannot be used for any other purpose. One common use is to identify control flow structures, for example `for`, `while`, `if/else`, `switch/case`, `try/catch` and `return`. Some keywords act as operators, for example `new` and `instanceof` in Java. The set of keywords differs from language to language. Some languages, including Smalltalk, do not have any keywords.

Many uses of keywords in Java can be parsed in Gel without any specific information about keywords.

<code>while (!b) { b = next(); }</code>	$(\sqcup \text{while } ![\text{b}]^0 (\text{; } (= \text{b } (\sqcup \text{next } \epsilon^0)^0)^0)^0)$
<code>p = new Point(3, 4)</code>	$(= \text{p } (\sqcup \text{new } (\sqcup \text{Point } (, 3 4)^0)^0)^0)$
<code>if (a&gt;b) f(i); else return;</code>	$(\text{; } (\sqcup \text{if } (> \text{a } \text{b})^0 (\sqcup \text{f } \text{i}^0)^0)^0 (\sqcup \text{else } \text{return}))$
<code>if (e instanceof Point) m(e)</code>	$(\sqcup \text{if } (\sqcup \text{e instanceof Point})^0 (\sqcup \text{m } \text{e}^0)^0)^0)$
<code>for (i = 9; i &gt; 1; i--) f(i)</code>	$(\sqcup \text{for } (\text{; } (= \text{i } 9)^0 (> \text{i } 1)^0 [\text{i}]--)^0 (\sqcup \text{f } \text{i}^0)^0)^0)$

This is not a general solution, however. The statement `return x + y` parses incorrectly as `(return x) + y` because sequence has higher precedence than `+`. A similar situation happens in ML or Haskell, which do not require parentheses as in Java and C, so control flow statements do not parse correctly in Gel.

<code>if a = b then 1 else 2</code>	$(= (\sqcup \text{if } \text{a}) (\sqcup \text{b then } 1 \text{ else } 2))$
-------------------------------------	--

These examples illustrate a common purpose for keywords — to label or combine expressions to form statements. When viewed from this perspective, keywords can be understood as a kind of low-precedence operator. In Gel, keywords are identified by a prefix or suffix unary colon operator. Keywords enable more of Java to be parsed correctly with Gel:

<code>return: x + y;</code>	$(; (\sqcup \text{return}]: (+ \text{x } \text{y})) \epsilon)$
<code>if: a = b then: 1 else: 2</code>	$(\sqcup \text{[if]}: (= \text{a } \text{b}) [\text{then}]: 1 [\text{else}]: 2)$

Keywords have precedence greater than semicolon but less than comma. In the abstract syntax (on the right) keywords are combined by a double-barred sequence operator, `=`. Gel generalizes the notion of a keyword to allow any expression with a prefix or suffix colon operator to be a keyword.

<code>n-val: 23; (test): 5</code>	$(; (\sqcup \text{[(- } \text{n } \text{val}])]: 23) (\sqcup \text{[test}^0]: 5))$
-----------------------------------	--

In addition, groups in curly braces are also treated as keywords. This convention mirrors usage in C/Java and CSS, where such groups are not included in sequences. Compare these examples:

<code>a + b [ more ] ≡ a+(b[more])</code>	$(+ \text{a } (\sqcup \text{b } \text{more}^0))$
<code>a + b { more } ≡ (a+b){more}</code>	$(\sqcup (+ \text{a } \text{b}) \text{more}^0)$

As a result, Gel parses these forms correctly:

```
class: C implements: A, B { ... }
(= [class]: C [implements]: (, A (.. B ...^0))

.info, h1 { color: #6CADDFF }
(= (, .[info] h1) (= [color]: #[6CADDFF])^0)

if: (b) { ... } a = 3;
(= (= [if]: (.. b^0 ...^0)) (= a 3) ε)
```

If these groups were not treated the same as keywords, they would parse as

```
class: C implements: A, (B { ... })
```

and

```
if: (((b) { ... } a) = 3);
```

The last example above illustrates a final special case: when a curly group is inside a semicolon operator, the group has an implicit semicolon added after it. In C++ the semicolon is required after a class declaration, but not after a method body. This special case for curly groups affects some other languages badly. For example, many parser generators use curly groups to enclose parser actions, so they do not parse correctly in Gel. The solution is to add a unary operator to the group, as in \*{ ... }, or to use a different grouping operator.

Gel also cannot meaningfully parse languages that use keywords for grouping, e.g. begin/end or if...end if, although more modern languages tend to use {} for indicating block structures. Parsing these examples correctly would require specific knowledge of the structure of statements.

There is a special case for keywords or curly braces that are the direct argument of a binary operator. In this case they keyword is nested inside the binary operator.

```
p = new: Point(3, 4)          (= p (= [new]: (.. Point (, 3 4)^0))^0)
x = {a} + b * test: x        (= (= x (+ a^0 (* b (= [test]: x)))))^0
b * k1: k2: 99               (* b (= [k1]: [k2]: 99))
```

The design of keywords is the most difficult part of Gel. We explored the option of user-defined keywords in a document or block header, but this complicated the language and interrupted the flow of content in a document. The colon marker is lightweight and explicit.

## 2.6 Quoting

Quoting is useful to indicate that a syntactic form has a special meaning. In Lisp, any expression can be quoted. Syntactically, this wraps the expression in a list beginning with the symbol **quote**, which tells the Lisp interpreter to use the expression as a literal data value. Quotes are also useful in defining grammars. They can be used to distinguish the syntax being defined from the meta-syntax of the grammar definition language. To illustrate, first consider a conventional presentation of the syntax of EBNF in EBNF:

```
grammar EBNF {
  grammar ::= "grammar" id "{" rule (";" rule)* "}" ;
  rule     ::= id "::=" pat ;
  pat      ::= id | str | pat pat | pat "|" pat | pat "*" ;
  id       ::= letter+ ;
  str      ::= quote any* quote
}
```

This is a typical grammar for parsing text streams, in which the tokens of the language being defined are enclosed in quotes. It assumes that the patterns `letter` and `quote` are predefined. This grammar is highly ambiguous, requiring significant work to resolve these ambiguities. More work would be needed to deal with white-space.

Gel suggests another possibility where the operators `"|"` and `"*"` are parsed as actual operators rather than strings. The operators that are part of the language being defined are marked with a backquote character:

```
id | `id | pat pat | pat`|pat | pat`*
```

This is a tree grammar [14] that recognizes Gel trees that represent EBNF patterns. The expression `pat`|pat` is written as a chunk (without spaces) so that it will have higher precedence than the other `|` operators. In the example below it is parenthesized instead. The full grammar is below:

```
grammar EBNF {
  grammar ::= (`grammar ID { rule* }) ;
  rule     ::= (ID `::= pat) ;
  pat      ::= ID | `ID | pat pat | (pat `| pat) | pat`* ;
}
```

In Gel any expression or operator can be quoted. A quoted operator has exactly the same precedence as its unquoted version. That is, Gel will create the same structure for a quoted expression and an unquoted version.

```
(`grammar EBNF
  (`grammar ID [rule]* `))
  (rule `::= ID pat `)
  (pat `| ID `ID (pat pat) (`| pat pat) [pat]`*))`)
```

Gel quoting can also be combined with a prefix operator to implement back-quote substitution as in Lisp. This kind of structural substitution has a counterpart in strings as defined in the next section.

## 2.7 Strings and Interpolation

Many languages allow variables or expressions to be embedded inside a string, a technique called *string interpolation*. For example, `"the $nth word"` is equivalent to `"the " + nth + " word"`. String interpolation is a short-hand for string concatenation. In Gel the `$` character can be followed by an optional symbol, then an optional operator, and then any number of groups. The parenthesis and square bracket

groups contain Gel, while the curly bracket groups enclose *strings*. That is, the text inside  $\$\{ \dots \}$  is implicitly quoted and can contain additional interpolations.

```
"$heading [2+n] {Section $n} equation: $={2+n}"  
(+ („ heading (+ 2 n)0 ("Section " n0)0)0 " equation: " („ = "2+n"0)0)0
```

Note that Gel's interpolations generalized both Perl notation and also  $\text{\TeX}$  [20]. After substituting  $\$$  for  $\backslash$ , Gel can extract meaningful structure from many (but not all)  $\text{\TeX}$  documents. Gel could be used for a Latex-like formatting language, but the Gel operator syntax could be used for math instead of text encoding as in  $\text{\TeX}$ .

### 3 Gel Specification

Gel is defined by a concrete grammar, an abstract syntax, and a set of rewrite rules to handle keywords. The grammar of Gel is given in Figure 3. As is standard,  $x^*$  means zero or more repetitions of  $x$ ,  $x^+$  is one or more, and  $x^?$  means zero or one copy of  $x$ . A set of characters in brackets  $[abc]$  represents exactly one character from the set. Character sets preceded by a  $\neg$  symbol represents exactly one character that is not in the character set, and sets superscripted by a number  $n$  represent  $n$  repetitions. Ranges may also appear in superscripts as  $n - m$ . White-space tokens are not ignored, but are represented explicitly in the grammar as  $[~]$ . Comments can only occur in conjunction with white-space. The reference parser for Gel is defined using Rats! [17], a Parsing Expression Grammar system [13]. Syntactic predicates are needed in the rule for  $B_{n+1}$  to identify extra operators as defined at the end of Section 2.3. More details and the Gel implementation are available for download at <http://www.utexas.edu/users/wcook/Gel>.

The first three productions represent lists, separated by semicolon ( $op_1$ ) and comma ( $op_2$ ), of optional items. The nonterminals  $B_3$  through  $B_n$  represent binary expressions with  $op_i$  surrounded by spaces. The  $[~]$  terminal represents any number of white-space characters, including single spaces, tabs, return feeds and new lines. These nonterminals have lower precedence than *sequence*, which is a list of chunks that do not contain spaces. The  $B_{n+1}$  rule allows operators to be part of a sequence, when they cannot be interpreted as binary operators. The nonterminals  $C_i$  are analogous to  $B_i$  except the operators do not have spaces. The  $C_{n+1}$  rule allows sequences of primaries that are not separated by spaces.

Compound operators are composed of any sequence of operator characters. The precedence order of operators is given by the table in Figure 2. For most operators the precedence is given by the precedence of the first character. There is a special case for *assignment operators*, which end with equal [=] and do not begin with [= <>].

A primary is a symbol, string, or group. The *string1* and *string2* rules define strings with single and double quotes, respectively. Both strings allow Java-style escaping with backslash. The  $[\$]$  character is an interpolation character in double-quoted strings. It allows interpolation of Gel expressions into a string. The single back-quote character (') is used for quoting. Any operator or primary may be quoted.

The behavior of keywords in Gel is not implemented by the parser, but is handled by the rewrite rules in Figure 4 during construction of the abstract syntax tree. The first rule combines operators to eliminate associativity. Keywords in a sequence are moved

```

expression ::= list quote? [ ; ] expression | list
list ::= optional quote? [ , ] list | optional
optional ::= [ _ ]? B3? [ _ ]?
Bi ::= Bi+1 [ _ ] opi [ _ ] Bi | Bi+1 for i ∈ {3..n}
Bn+1 ::= op [ _ ]? | (op [ _ ])* sequence ([ _ ] op )*
sequence ::= chunk ([ _ ] chunk)*
chunk ::= op? C3 op?
Ci ::= Ci+1 opi Ci | Ci+1 for i ∈ {3..n}
Cn+1 ::= primary+
opi ::= quote? [ ` :$@? | & ! = < > + - * / \ % ~ ^ # . ]+
where i is the precedence as defined in Figure 2
op ::= op1 | ... | opn
quote ::= [ ` ]+
primary ::= quote? (symbol | group | string1 | string2)
symbol ::= [a-zA-Z0-9_-]+
group ::= [{} expression {}] | exprGroup
string1 ::= [ ' ] (escape | ¬[ ' ] )* [ ' ]
string2 ::= [ " ] (escape | interpolate | ¬[ " ] )* [ " ]
escape ::= [ \ ] [u][0-9A-F]4 | [ \ ] [0-7]1-3 | [ \ ] -
interpolate ::= [ $ ] symbol? op? (string3 | exprGroup)*
string3 ::= [{} (escape | interpolate | ¬[{} ] )* {}]
exprGroup ::= [ ( ) expression () ] | [ [ ] expression [ ] ]
ignore ::= [ / ][ / ] (¬newline)* newline | [ / ][ * ] any* [ * ][ / ]

```

**Fig. 3.** Gel grammar, where  $n$  is the number of operator precedence levels

$(\star (\star \bar{x}) x) \Rightarrow (\star \bar{x} x)$ $(\underline{\omega} k x) \Rightarrow (\underline{\omega} k x)$ $(\circ (\underline{\omega} \bar{x} v_1) v_2) \Rightarrow (\underline{\omega} \bar{x} (\circ v_1 v_2))$ $(\underline{\omega} (\underline{\omega} \bar{x}_1) (\underline{\omega} \bar{x}_2)) \Rightarrow (\underline{\omega} \bar{x}_1 \bar{x}_2)$ $(\underline{\omega} (\underline{\omega} \bar{x}) x) \Rightarrow (\underline{\omega} \bar{x} x)$ $(; \bar{x}_1 (\underline{\omega} \bar{x}_2 x^0 \bar{x}_3) \bar{x}_4) \Rightarrow (; \bar{x}_1 (\underline{\omega} \bar{x}_2 x^0) (\underline{\omega} \bar{x}_3) \bar{x}_4)$	$(\star x (\star \bar{x})) \Rightarrow (\star x \bar{x})$ $(\underline{\omega} x k) \Rightarrow (\underline{\omega} x k)$ $(\circ v_1 (\underline{\omega} v_2 \bar{x})) \Rightarrow (\underline{\omega} (v_1 v_2) \bar{x})$ $(\underline{\omega} x (\underline{\omega} \bar{x})) \Rightarrow (\underline{\omega} x \bar{x})$
--	---

$\star, \circ \in \text{op}$ ,  $\circ \notin [ ; ]$ ,  $x$  is any Gel,  $k \in \{ [x]:, :[x], x^0 \}$ ,  $v \notin \{ x:, :x, x^0, (\underline{\omega} \bar{x}) \}$

**Fig. 4.** Keyword rewrite rules

$$\begin{aligned}
x \in \text{Gel} &= \text{symbol} \mid \text{"str"} \mid (\star x_1 \dots x_n) \mid \star[x] \mid [x]\star \mid \star[x]\star \mid 'x \mid x^G \mid \epsilon \\
\text{symbol} &\in [a-zA-Z0-9_-] +
\end{aligned}$$

$$\star \in ; \mid , \mid [ ` :$@? \mid & ! = < > + - * / \ % ~ ^ # . ]+ \mid \underline{\omega} \mid \underline{=}$$

$$G \in \text{Group} = () \mid \{ \} \mid [] \mid \langle \rangle$$

where  $\underline{\omega}$  means sequence,  $\underline{=}$  means keyword sequence,  $\langle \rangle$  means chunk

**Fig. 5.** Gel abstract syntax

outside of other operators. The last rule adds an implicit semicolon after a group. The abstract syntax of Gel is defined in Figure 5.

## 4 Evaluation

We evaluate Gel by testing how well it can extract the structure from existing languages that are defined by a custom grammar. It is not enough to determine whether Gel accepts a given input, because Gel accepts almost any input with balanced grouping operators. The key question is whether Gel can extract meaningful structure from typical documents that follow standard formatting conventions. These tests were instrumental in designing Gel.

Let  $S$  be a source file of a language  $L$ , and let  $L(S)$  be the AST of  $S$  created by the  $L$  parser. The same source file  $S$  can be parsed with Gel to produce a GAST,  $Gel(S)$ . The goal is to determine if  $Gel(S)$  has the same structure as  $L(S)$ .

However, the  $L(S)$  AST cannot be compared to the GAST because each uses a different abstract syntax. To overcome this problem, we apply the idea that the structure of an abstract tree can be made explicit in concrete syntax by adding parentheses at every level of the tree. The implementation of this idea starts with a printer  $P$  for language  $L$  having the property that  $L(P(T)) = T$  for any abstract tree  $T$  in  $L$ . We then convert  $P$  into a *parenthesizing printer*  $P'$  that prints a tree  $T$  while adding parentheses around every abstract node as it is printed. The output  $P'(T)$  may not be a valid instance of language  $L$ , but it can be parsed with Gel. The extra parentheses force Gel to create a parse tree that mirrors the structure of  $L(S)$ . Gel has captured the structure of  $L$  if  $Gel(P'(L(S))) = Gel(S)$  ignoring parentheses. As an example, consider this Smalltalk fragment and its parenthesized versions:

```
x min to: args size * 2 do: aBlock
      (= („ x min) [to]: (* („ args size) 2) [do]: aBlock)
((x) min) to: (((args) size) * (2)) do: (aBlock)
      (= („ x0 min) [to]: (* („ args0 size) 20) [do]: aBlock0)
```

Not all languages follow Gel's syntactic standards. Although there can be significant differences, sometimes the differences are small, for example comment markers and operator choices may conflict. Smalltalk separates statements with a period, which is a high-precedence binary operator in Gel. If Smalltalk used a semicolon, as in Java, Gel would parse it more accurately. We handle minor syntactic issues by converting symbols before parsing with Gel. This change preserves the key characteristics of Smalltalk; it just uses a different symbol. A fixup transformation  $T$  for language  $L$  is applied to the files before they are parsed by Gel. These transformations are simple character or reserved word substitutions. With transformation, the comparison is  $Gel(T(P'(L(S)))) = Gel(T(S))$ . We have successfully applied this technique to Smalltalk, Java, CSS and CORBA IDL. For a small set of representative sample documents, Gel extracts the exact same structure as the custom parser, in all but a few cases as mentioned below. There may be other syntactic mismatches that did not show up in our test documents.

## 4.1 Java

Gel operators and precedence are based on Java, but Gel does not have exactly the same operator precedences, so it will not parse Java precisely. We tested Gel against Java documents whose operators align with Gel precedence. Other issues in Java are related to sequences, where two syntactic structures are placed next to each other with just a space between them.

- Declarations of multiple variables do not parse correctly, as described at the end of Section 2.4.
- Java keywords do not parse correctly unless they are marked as Gel keywords as mentioned in Section 2.5.
- The grammar we used for Java parses  $o.m()$  as  $(\omega (o m) \epsilon^0)$ , while Gel parses it as  $(o (\omega m \epsilon^0))$ . It is debatable which of these is correct.
- Generics in Java are declared using the `<` and `>` characters, as in `Stack<String>`. We translated these to [...] before parsing.
- Typical white-space conventions must be followed: using white-space after a colon, and around binary operators. Typical white-space means that `int [] x` must not be written `int [] x` although this is legal in Java, it violates coding conventions. Similarly, `p = *p2` must not be written `p =* p2`. We found that the reformat command in Eclipse corrects most spacing issues in Java documents so that they parse correctly with Gel.

A preprocessor used a total of 12 modifications rules to make the above changes to all test cases. We tested Gel on a 300 line program and a 5740 line program generated by the Rats! parser generator.

## 4.2 Smalltalk

The Gel syntax closely resembles and generalizes Smalltalk grammar. Keywords in Smalltalk are identified by a colon suffix. Arbitrary binary operators use infix notation, and have higher precedence than keywords. Unary messages are represented by a sequence of symbols separated by spaces, with higher precedence than binary operators. Parentheses, braces, and brackets are used for grouping.

There are problems with parsing using Gel:

- Statements are terminated or separated by periods. We translated these semicolons before parsing with Gel.
- Cascaded message sends are separated by semicolons. These become ambiguous if period is replaced by semicolon. We insert a special “previous” token after the semicolon to make reuse of the previous message target explicit. These message sends must also be enclosed in parentheses if the target object is returned.
- Binary operators in Smalltalk all have the same precedence.
- The conventional storage format for Smalltalk programs (the method change list) does not have grouping constructs that can be parsed by Gel.
- Typical white-space conventions must be followed: using white-space after a colon, and around binary operators.

A preprocessor used a total of 2 modifications to make the above changes on a test case of about 100 lines.

### 4.3 CSS

Most of CSS follows a typical structure with semi-colons and braces. CSS also uses keywords tagged with colon. It uses a variety of prefix and infix operators. However, there are problems with parsing CSS with Gel:

- Identifiers that include hyphens, e.g. `background-color`, parse as chunks in Gel. This works reasonably well, although Gel is breaking up more tokens than are necessary.
- Typical white-space conventions must be followed: using white-space after a colon, and not separating prefix operators from their.
- Pseudo-classes look like binary colon operators, of the form `link:visible`. According to one CSS grammar they should be parsed as `link (:visible)` but Gel parses them as `( : link visible)`. This does not seem like a major issue.
- The use of numbers with a dimension, as in `16pt`, is handled in Gel as an identifier, not as a sequence of number 16 and `pt`. It is simple to process these tokens to extract the dimension.

A total of 4 modification rules made the above changes on 617 lines of CSS code taken from various websites.

### 4.4 Python

Although Python does not adhere to many of the conventions discussed, Gel is able to parse Python programs. The following problems must be addressed to parse Python correctly:

- In Gel, logical blocks of code can only be created using the three types of grouping operators. However, Python uses indentation to specify logical blocks of code. This is currently handled by a pre-processor, which inserts `{ . . . }` groups according to indentation rules of Python. This preprocess is a lexical transformation.
- Many statement constructs in Python use the colon character, as in `if x is True : . . .`. These can be discarded once grouping operators are created around the logical block.
- Python uses newline to separate statements. However, these would parse as white-space tokens in Gel, so semicolons must be inserted.

No formal tests have been done on Python.

### 4.5 ANTLR and Other Parser Generators

We have used Gel to parse grammar specification languages, including ANTLR [25] and Rats! [17]. These languages use `{ . . . }` as parser actions within a rule. A prefix or suffix must be added to prevent actions from terminating the expression (according to the keyword rule in Section 2.5). In addition, Rats! uses `[A-Z]` as a character class, in effect quoting an arbitrary set of characters, as in `[ ( { }`. These must quoted as strings, or converted to the form used by ANTLR: `'A' . . . 'Z'`.

No modifications were made to ANTLR source files. We tested Gel on a 1000 line ANTLR implementation of the Java language.

## 5 Related Work

Gel is related to other generic and extensible languages, include Lisp, XML and JSON. Gel can parse Lisp-like data [23], if single-quote is converted to backqoute, comma to \$, comments to // . Common lisp atoms \*val-list\* are converted to Gel chunks \*[(-val list)]\* with prefix/suffix \* operators, which means that they have been over-analyzed but are still recognizable. Any sequence of non-punctuation characters without spaces can be parsed as Gel. Operators are the main problem, since Lisp always treats them as ordinary symbols, but Gel may parse them as binary operators. Thus (a + b) is incorrectly parsed as (+ a b) in Gel. To express something like the correct Lisp structure ( $\sqcup$  a + b) the operator must be changed, for example enclosed in a group (a {+} b).

XML [9] cannot be parsed by Gel at all. It uses < and > as grouping characters, and tags as grouping for large-scale units. To parse XML-like structures, a more C-like notation is needed.

```
<tag attr="value" ...>...</tag> ⇒ tag: attr="value" ... { ... }
text ⇒ "text"
```

Alternatively, Gel could simulate the text-oriented nature of XML and its history in HTML by using an interpolation-based translation:

```
<tag attr="value" ...>...</tag> ⇒ $tag(attr="value" ...){ ... }
```

The JavaScript Object Notation (JSON) is a subset of JavaScript that is frequently used as a generic data encoding language [12]. Correct parsing of JSON depends on consistent white-space conventions. It works well if colon is treated as a binary operator.

```
"val" : 3, "name" : "Test"   (, (: "val" 3) (: "name" "Test"))
"val": 3, "name": "Test"     (= ["val"]:(, 3 (= ["name"]:"Test")))
"val": 3; "name": "Test"     (; (= ["val"]:) 3) (= ["name"]:"Test"))
```

The keyword notation in the second example groups the values awkwardly: the second keyword is within the body of the first keyword because of the comma. If JSON used semi-colons then Gel could parse the keyword form more naturally, as in the third example.

Another approach to extensible languages involves languages whose syntax can be extended with additional rules. This approach has the advantage that specific syntax is recognized and checked during parsing. Brabrand and Schwartzbach [5] provide a detailed summary and comparison of different systems for syntax extension [10,6]. Extensible grammars and macros are an active research area [27,8,16]. Gel does not compete with these approaches, but may be complementary. Since any language defined using Gel is a subset of all Gel inputs, it is still necessary to recognize or validate the input as legal. The issues in defining extensible languages (subsets) within Gel are similar to those faced by traditional extensible grammars. The primary difference is if Gel handles the initial syntactic recognition phase (including precedence and nesting), these issues do not need to be addressed at the level of language extensions. More work is needed to experiment with Gel as the initial phase of a complete language engineering framework. Examples of such systems include  [4,5] and Stratego [28]. Lisp and Scheme macros provide a similar benefit in the context of the generic syntax

of Lisp S-Expressions. Gel is not yet part of a complete system for language definition and syntactic extension, so it is difficult to compare its effectiveness at this level. Given that Gel is essentially a syntactic variant of Lisp S-Expressions, the techniques developed for Lisp/Scheme should work for Gel as well. This kind of validation will not be possible until other researchers experiment with using Gel in their own systems.

## 6 Conclusions

In this paper we have discussed Gel, a generic extensible language: generic because it has a fixed syntax based on common syntactic conventions, and extensible in that it supports arbitrary unary and infix operators, arbitrary keywords and arbitrary sequencing of expressions. We defined a set of *operator* precedence rules chosen to closely mimic those of common languages, and a larger set of *expression* precedence rules to handle the introduction of the sequence and keyword operators. Lastly, we developed a form of string interpolation for extracting meaningful structure from Latex-like formatting languages.

Gel is designed to be used as a front-end for domain-specific languages. To define a language within Gel, appropriate operators and syntactic forms are chosen, and a structure grammar is defined. The output tree from Gel must then be parsed to verify that it matches the DSL structure. This process is very much like validating against an XML Schema [26,3] but is beyond the scope of this paper. Gel enables easy syntactic composition or embedding of different languages within each other. It may also be possible to define a generic pretty-printer for Gel.

One argument against Gel may be that its use of white spaces makes it too fragile for casual use. However, most programming languages are sensitive to adding new arbitrary spaces, or completely removing spaces. Gel accepts nearly every input document without error, as long as grouping symbols are balanced. When used for a specific DSL, error messages will come from later phases, when the output of Gel is validated against the DSL structure.

During the design of Gel numerous alternatives were tried. We have worked hard to eliminate special cases. Currently the only special cases are for assignment operators and curly braces. These special cases are relatively simple for users and provide useful options to language designers when designing a new notation. We have resisted allowing the grammar to be customized, for example by allowing external definition of a set of keywords. We plan to gather feedback on Gel for a short period of time before fixing the language specification.

## References

1. Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Herrick, H.L., Hughes, R.A., Mitchell, L.B., Nelson, R.A., Nutt, R., Sayre, D., Sheridan, B.P., Stern, H., Ziller, I.: Fortran Automated Coding System For the IBM 704. International Business Machines Corporation, New York (1956)
2. Berners-Lee, T., Masinter, L., McCahill, M.: Uniform Resource Locators (URL). RFC 1738, Internet Engineering Task Force (December 1994),  
<http://ds.internic.net/rfc/rfc1738.txt> (accessed August 23, 1997)

3. Biron, P.V., Malhotra, A.: XML Schema part 2: Datatypes. The World Wide Web Consortium (May 2001), <http://www.w3.org/TR/xmlschema-2/>
4. Brabrand, C., Møller, A., Schwartzbach, M.I.: The *<bigwig>* project. ACM Trans. Interet Technol. 2(2), 79–114 (2002)
5. Brabrand, C., Schwartzbach, M.I.: Growing languages with metamorphic syntax macros. In: Proceedings of Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2002, pp. 31–40. ACM Press, New York (2002)
6. Brabrand, C., Schwartzbach, M.I.: The metafront system: Safe and extensible parsing and transformation. Sci. Comput. Program 68(1), 2–20 (2007)
7. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. a language and toolset for program transformation. Sci. Comput. Program 72(1-2), 52–70 (2008)
8. Bravenboer, M., Visser, E.: Designing syntax embeddings and assimilations for language libraries. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 34–46. Springer, Heidelberg (2007)
9. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. (eds.): Extensible Markup Language (XML) 1.0. W3C Recommendation. W3C, 4th edn. (August 2003)
10. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. Technical report, Research Report 121, Digital SRC (1994)
11. Crocker, D.H.: Standard for the Format of ARPA Internet Text Messages. University of Delaware, Department of Electrical Engineering, Newark, DE 19711 (August 1982), <http://www.faqs.org/rfcs/rfc822.html>
12. Crockford, D.: Rfc 4627. the application/json media type for javascript object notation (json) (2006), <http://www.json.org/>
13. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Symposium on Principles of Programming Languages, pp. 111–122 (2004)
14. Gladky, A.V., Melčuk, I.A.: Tree grammars (=  $\Delta$ -grammars). In: Proceedings of the 1969 Conference on Computational linguistics, Morristown, NJ, USA, pp. 1–7. Association for Computational Linguistics (1969)
15. Goldberg, A., Robson, D.: Smalltalk-80: the Language and Its Implementation. Addison-Wesley, Reading (1983)
16. Grimm, R.: Practical packrat parsing. New York University Technical Report, Dept. of Computer Science, TR2004-854 (2004)
17. Grimm, R.: Better extensibility through modular syntax. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, pp. 38–51. ACM, New York (2006)
18. Hudak, P., Jones, S.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburz, D., Nikhil, R., Partain, W., Peterson, J.: Report on the programming language Haskell: a non-strict, purely functional language version 1.2. SIGPLAN Not. 27(5), 1–164 (1992)
19. Johnson, S.C.: Yacc: Yet another compiler compiler. In: UNIX Programmer’s Manual, vol. 2, pp. 353–387. Holt, Rinehart, and Winston, New York (1979)
20. Knuth, D.E.: The *T<sub>E</sub>Xbook*. Addison-Wesley, Reading (1984)
21. Krishnamurthi, S.: Programming Languages: Application and Interpretation (2006), <http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/>
22. Lie, H.W., Bos, B.: Cascading Style Sheets, level 1. W3c recommendation, W3C (January 1999)
23. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM 3(4), 184–195 (1960)
24. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A comprehensive step-by-step guide. Artima Inc. (August 2008)

25. Parr, T., Quong, R.: ANTLR: A Predicated-LL(k) parser generator. *Journal of Software Practice and Experience* 25(7), 789–810 (1995)
26. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema part 1: Structures. The World Wide Web Consortium (May 2001),  
<http://www.w3.org/TR/xmlschema-2/>
27. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
28. Visser, E.: Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)

# A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications

Zoé Drey<sup>1</sup>, Julien Mercadal<sup>2</sup>, and Charles Consel<sup>2</sup>

<sup>1</sup> Thales / LaBRI-Université de Bordeaux, France

<sup>2</sup> INRIA / LaBRI-Université de Bordeaux, France

`{drey, mercadal, consel}@labri.fr`

**Abstract.** Various forms of pervasive computing environments are being deployed in an increasing number of areas including healthcare, home automation, and military. This evolution makes the development of pervasive computing applications challenging because it requires to manage a range of heterogeneous entities with a wide variety of functionalities.

This paper presents Pantagruel, an approach to integrating a taxonomical description of a pervasive computing environment into a visual programming language. A taxonomy describes the relevant entities of a given pervasive computing area and serves as a parameter to a sensor-controller-actuator development paradigm. The orchestration of area-specific entities is supported by high-level constructs, customized with respect to taxonomical information.

We have implemented a visual environment to develop taxonomies and orchestration rules. Furthermore, we have developed a compiler for Pantagruel and successfully used it for applications in various pervasive computing areas, such as home automation and building management.

**Keywords:** Visual Rule-Based Language, Pervasive Computing.

## 1 Introduction

Various forms of pervasive computing environments are being deployed in an increasing number of areas including healthcare [1], home automation [2], building management [3] and military [4]. This trend is fueled by a constant flow of innovations in devices forming ever richer pervasive computing environments. These devices have increasingly more computing power offering high-level interfaces to access rich functionalities. Access to their functionalities can be done remotely because most devices are now networked.

The advent of this new generation of devices enables the development of pervasive computing systems to abstract over low-level embedded systems intricacies. This development is now mainly concerned with the programming of the orchestration of the entities, whether hardware (sensors and actuators) or software (*e.g.*, databases and calendars). Yet, the nature of pervasive computing systems makes this programming very challenging. Indeed, orchestrating networked heterogeneous entities requires expertise in a number of areas, including

distributed systems, networking, and multimedia. There exist middlewares and programming frameworks that are aimed to facilitate this task; examples include Gaia [5], Olympus [6], One.World [7], and Aura [8]. However, these approaches do not fill the semantic gap between an orchestration logic and its implementation because they rely on a general-purpose language and use large APIs. This situation makes the programming of the orchestration logic costly and error-prone, impeding evolutions to match user's requirements and preferences.

To circumvent this problem, visual approaches to programming the orchestration logic have been proposed, based on storyboards and rules [9,10,11]. These approaches enable the programmers to express the orchestration logic using intuitive abstractions, facilitating the development process. However, this improvement is obtained at the expense of expressivity. Specifically, existing visual approaches are limited to a given area (*e.g.*, CAMP magnetic poetry for the Smart Home domain [10]), a pre-defined set of categories of entities (*e.g.*, in iCap rule-based interactive tool [11]), or abstractions that do not scale up to rich orchestration logic (*e.g.*, Oscar service composition interface [12]).

## 1.1 This Paper

This paper presents Pantagruel, an expressive approach to developing orchestration logic that is parameterized with respect to a *taxonomy* of entities describing a pervasive computing environment. Specifically, our approach consists of a two-step process: (1) a pervasive computing environment is described in terms of its constituent entities, their functionalities and their properties; (2) the development of a pervasive computing application is driven by a taxonomy of entities and consists of orchestrating them using high-level constructs.

The environment description allows our approach to be instantiated with respect to a given application area. This description defines the classes of entities that are relevant to the target area. For each class, it specifies an interface to access its functionalities. Because the orchestration logic is written with respect to the environment description, entities are combined in compliance with their description. To facilitate the programming of the orchestration logic, we have developed a visual tool that uses a sensor-controller-actuator paradigm. This paradigm is suitable for programmers with standard skills, or even for novice programmers, as demonstrated by its use in various fields such as computer games (*e.g.*, Blender<sup>1</sup>) and robots (*e.g.*, Altaira [13] or LegoSheets [14] for Lego Mindstorms<sup>2</sup>). Like a game logic, an orchestration logic collects context data from sensors, combines them with a controller, and reacts by triggering actuators. This paradigm is intuitive and makes programs readable. Furthermore, our visual programming environment offers the developer an interface that is customized with respect to the environment description. Information about the environment entities is exploited to guide the programmer in defining sensor-controller-actuator rules.

---

<sup>1</sup> <http://www.blender.org>

<sup>2</sup> <http://mindstorms.lego.com/>

Because Pantagruel is a very high-level language, its compilation could be challenging, leading to intricate and error-prone processings. To alleviate this problem, we have developed a compiler for Pantagruel that leverages an architecture description language (ADL) dedicated to distributed systems, named DiaSpec [15]. Given an architecture description, the compiler for this ADL, named DiaGen, generates a dedicated programming framework in Java, which provides extensive support to discover and interact with distributed entities. The compilation process of Pantagruel consists of two stages: (1) an environment description is translated into a DiaSpec description (2) orchestration rules are compiled into Java code, supported by a DiaGen-generated programming framework. Leveraging on DiaSpec enables Pantagruel to gain access to a large number of heterogeneous entities, available in our Lab's smart space. They include hardware entities (*e.g.*, IP phones, webcams, displays, and X10 controllers) and software entities (*e.g.*, calendars, presence agents and instant messaging clients).

The contributions of this paper are as follows.

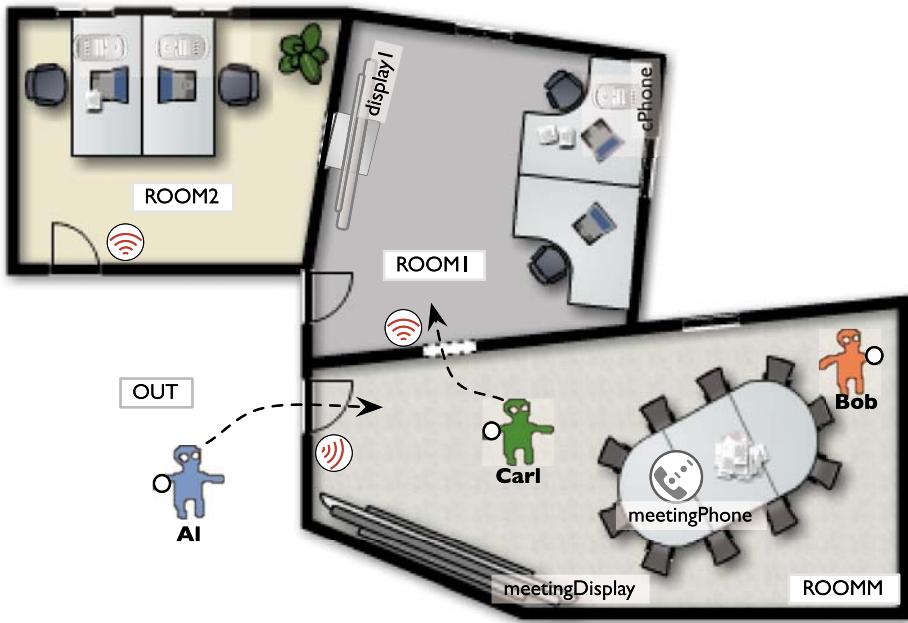
- *Area-specific approach.* We introduce a novel approach to visual programming of pervasive computing applications that is parameterized with respect to a description of a pervasive computing environment. This makes our approach applicable to a range of areas.
- *Area-specific visual programming.* We extend the sensor-controller-actuator paradigm to allow the programming of the orchestration logic to be driven by an environment description. This extended approach eases programming and enables verifications.
- *Preliminary validation.* We have implemented a compiler and successfully used it for applications in various pervasive computing areas such as home automation and building management.

## 1.2 Outline

To motivate our approach, Section 2 presents an example of a meeting application. Section 3 then introduces our taxonomical approach to defining descriptions of pervasive computing environments. Section 4 presents a visual environment to develop applications that orchestrate entities, defined in a taxonomy. The implementation of Pantagruel is examined in Section 5 and a study of its use is presented in Section 6. The related work is detailed in Section 7. Concluding remarks and future work are provided in Section 8.

## 2 Working Example

To motivate our approach, we consider as an example a meeting management application. This application area orchestrates various kinds of entities, consisting of RFID readers, presence agents, an LCD display, laptops, IP phones, a shared agenda, and an instant messaging server. An example of a physical layout for an office space is displayed in Figure 1.



**Fig. 1.** The physical layout of an office space

A RFID reader is placed in every room (ROOM1, ROOM2, and ROOMM) to detect the location of users (AI, Bob and Carl) wearing an RFID tag. A shared agenda stores each meeting with additional information such as the date, the starting and ending times, and the participants.

Numerous orchestration scenarios can be defined to manage meetings. Let us examine an example that focuses on starting meetings on time with all participants. For simplicity, we suppose that meetings occur in the same room, say the meeting room. Specifically, if a participant is not present in the meeting room but connected via instant messaging, (s)he receives a reminder message shortly before the meeting time. To make the meeting room available on time for the next scheduled meeting, a message is sent to the LCD display indicating the ending time of the current meeting and the list of participants that need to leave before the next meeting starts. When a new meeting starts, the organizer's laptop switches over to the LCD display of the meeting room, making it possible to share documents and presentations with the participants. Participants attending the meeting from a remote location but connected by instant messaging are invited to start an audio session using their IP phone. Simultaneously, the meeting's presentations are displayed on their laptop.

Many other scenarios for meeting management could be imagined. More scenarios combining meeting management with related activities could be introduced. In fact, for a given environment, it should be possible to define various orchestration scenarios, adapting to users' requirements and preferences, and

reacting to users' feedback. Because these scenarios can have a tremendous impact on people's life, it is critical to ease the creation and improvement of orchestration logic. In doing so, orchestration logic becomes understandable to the widest audience and close to the users' informal specification.

Also, our example application area illustrates the richness of the entities that are commonly available today, requiring expressivity to combine them. Finally, the office space environment consists of entities for which numerous variations are available, requiring an approach to defining the orchestration logic that abstracts over these differences.

### 3 Defining a Pervasive Computing Environment

To abstract over the variations of entities, we introduce a declarative approach to defining a taxonomy of entities relevant to a given area. The entity declarations form an environment description that can then be instantiated for a particular setting.

#### 3.1 Environment Description

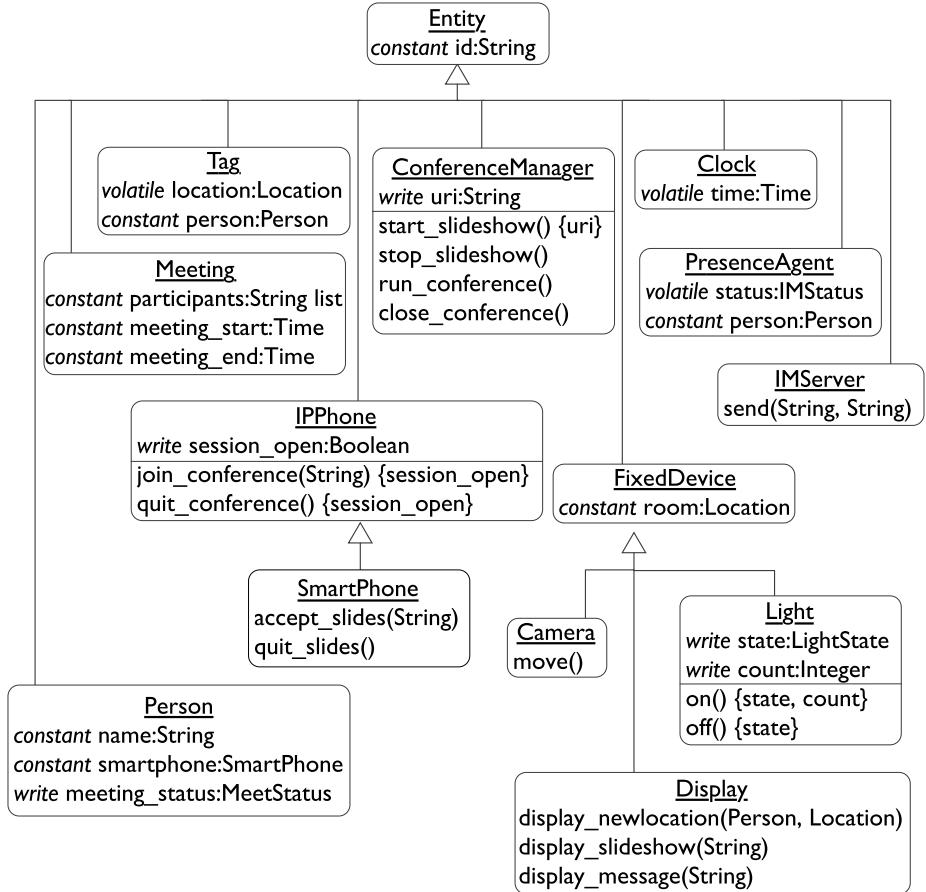
An environment description consists of declarations of entity classes, each of which characterizes a collection of entities that share common functionalities. The declaration of an entity class lists how to interact with entities belonging to this class. The generality of these declarations makes it possible to abstract over a range of variations, enabling the re-use of an environment description.

Furthermore, the declaration of an entity class consists of attributes defining a context and methods accessing the entity functionalities. Entity classes are organized hierarchically, allowing attributes and methods to be inherited. Figure 2 displays a hierarchy of entity classes for the meeting management area in an office space. Starting at the root, this hierarchy breaks down the entity classes into increasingly specific elements. Each successive element adds new attributes and methods.

*Entity context.* Context awareness is a critical issue in programming a pervasive computing application. If addressed with inappropriate abstractions, it may bloat the application code with conditionals and data structure operations. In our approach, an entity class defines attributes, each of which holds some element of the context of a pervasive computing system. A context element may either be constant, external or applicative. A context element is tested in the sensor part of an orchestration rule. Let us examine each kind of context element.

A *constant* context element is an attribute whose value does not change over time. For example, the `FixedDevice` entity class declares a `room` attribute which is constant for a given setting. As such, an instance of a `Camera` entity class is assumed to have a constant location.

An external context element is aimed to acquire context information from the outside. This kind of context element may correspond to sensors (*e.g.*, a



**Fig. 2.** A hierarchy of classes of entities for our working example

device reporting RFID tag location) or software components (*e.g.*, a calendar reporting meeting events). To model an external context element that varies over time, we introduce the notion of *volatile* attribute. To communicate context information to a Pantagruel program, an external entity updates the value of this attribute. An updated value may then trigger an orchestration rule. As an example of volatile attribute, consider the Tag class entity in Figure 2. Each instance of Tag holds the Location of its owner. This location is updated as the owner moves around and is detected by RFID readers. In our example, the location is an enumeration type over the rooms of the office space. As can be noticed, this information is higher level than the raw information captured by an RFID reader. This level of abstraction requires an expert to wrap the RFID reader such that when a tag is detected, it retrieves its owner and updates the corresponding Pantagruel object. This wrapper is very similar to a device driver, keeping device intricacies separate from entity orchestration.

Lastly, an applicative context element corresponds to context data computed by the application. To address applicative context elements, we introduce *write* attributes. These attributes can be written in the actuator part of an orchestration rule. In our example, Figure 2 shows the `meeting_status` attribute that can either be `PRESENT`, `REMOTE` or `ABSENT`, depending on the status of the meeting participant. This attribute is updated depending on the participant's tag location and presence status.

By raising context elements to abstractions in the Pantagruel language, we facilitate their manipulation, improve readability, and enable program verification.

*Methods.* The functionalities of an entity class correspond to method interfaces. They are typed to further enable verification. Methods are introduced to perform actions that are out of the scope of Pantagruel. For example, Pantagruel is not suited to program the sequence of operations needed to rotate a camera. As a consequence, the `Camera` entity class includes a `move` method signature to access this functionality. This method is invoked in the actuator part of an orchestration rule.

When a method is invoked, it may modify the applicative context. Consider the `Light` entity class. Instances of this class have a limited life expectancy that depends on the number of times they are switched on and off. To allow the programmer to design rules that would control the use of lights, we modeled `Light` with two attributes: `state`, holding the light status (`ON` or `OFF`), and `count`, counting the number of times the light is activated. Additionally, the `Light` entity class includes the `on()` and `off()` method signatures. Turning on/off a light affects the `Light` attributes and thus the applicative context. When this method is invoked in an actuator, the Pantagruel developer needs to be aware of these side-effects to define the orchestration logic. To do so, we require side-effecting methods to declare the list of attributes they may alter. For example, the `on()` method declaration includes the list of attributes that are updated by this method, namely `{state, count}`.

### 3.2 Instantiating an Environment Description

Once the environment description is completed, it is used to define concrete environments by instantiating entity classes. Figure 3 gives an excerpt of the concrete entities used in our meeting management example. Each entity has a name and refers to its entity class. For instance, we created the `meetingPhone` entity of the `SmartPhone` class; it is noted `meetingPhone:SmartPhone`. This entity corresponds to the phone that is located in the meeting room. Because it plays a specific role in our meeting management scenario, it needs to be created at this stage to allow the programmer to use it in the orchestration logic.

Figure 3 also includes examples of meeting events registered in the shared agenda; they are instances of the `Meeting` entity class. As such, their attributes are constant. Entity instances can be created dynamically in a given environment (*e.g.*, meeting events and RFID tags). As discussed in the next section, the orchestration logic can be written so as to apply to all instances of an entity

meetingDisplay: <u>Display</u>	clock: <u>Clock</u>	callconfManager: <u>ConferenceManager</u>
room:=ROOMM		
display1: <u>Display</u>	imServer: <u>IMServer</u>	meetingPhone: <u>SmartPhone</u>
room:=ROOM1		
al: <u>Person</u>	bob: <u>Person</u>	carl: <u>Person</u>
name:='Al'	name:='Bob'	name:='Carl'
smartphone:=aPhone	smartphone:=bPhone	smartphone:=cPhone
aPhone: <u>SmartPhone</u>	bPhone: <u>SmartPhone</u>	cPhone: <u>SmartPhone</u>
aTag: <u>Tag</u>	bTag: <u>Tag</u>	cTag: <u>Tag</u>
person:=al	person:=bob	person:=carl
aAgent: <u>PresenceAgent</u>	bAgent: <u>PresenceAgent</u>	cAgent: <u>PresenceAgent</u>
person:=al	person:=bob	person:=carl
m1: <u>Meeting</u>	m2: <u>Meeting</u>	m3: <u>Meeting</u>
meeting_start:=15h	meeting_start:=12h	meeting_start:=10h
meeting_end:=16h	meeting_end:=13h	meeting_end:=11h
participants:=['Bob', 'Carl']	participants:=['Al', 'Carl']	participants:=['Al', 'Bob']

**Fig. 3.** An excerpt of a concrete environment

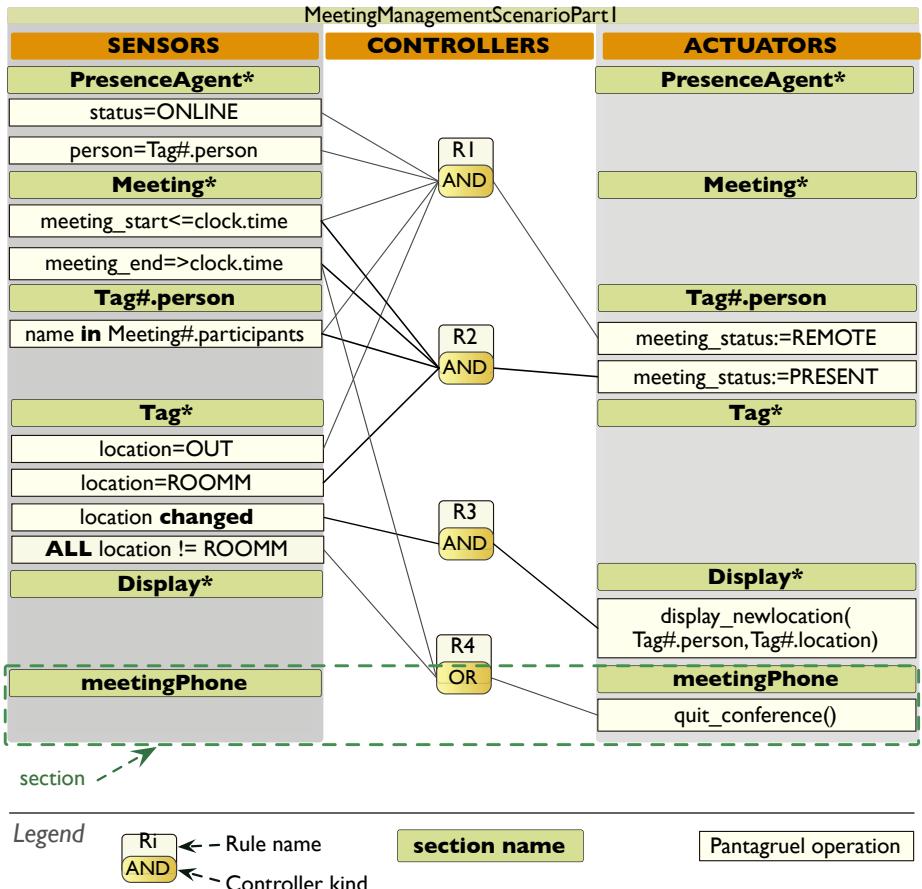
class, or to a specific instance (*e.g.*, `meetingPhone`). Notice that applicative and external context elements are initially undefined.

Pantagruel allows simple datatypes to be defined. An example of an enumeration type is given by `Location` in Figure 3; it ranges over the rooms of our example office space.

## 4 Defining Orchestration Rules

We now present a visual environment dedicated to the development of orchestration rules. Following our paradigm, the Pantagruel development environment offers a panel divided in three columns: sensors, controllers, and actuators. This panel is shown in Figures 4 and 5. These two figures present the orchestration rules of our working example of meeting manager (Section 2).

To develop an application, the programmer starts by defining some conditions on context elements in the sensor column, combining them in the controller column, and triggering actions in the actuator column. For readability, rules are

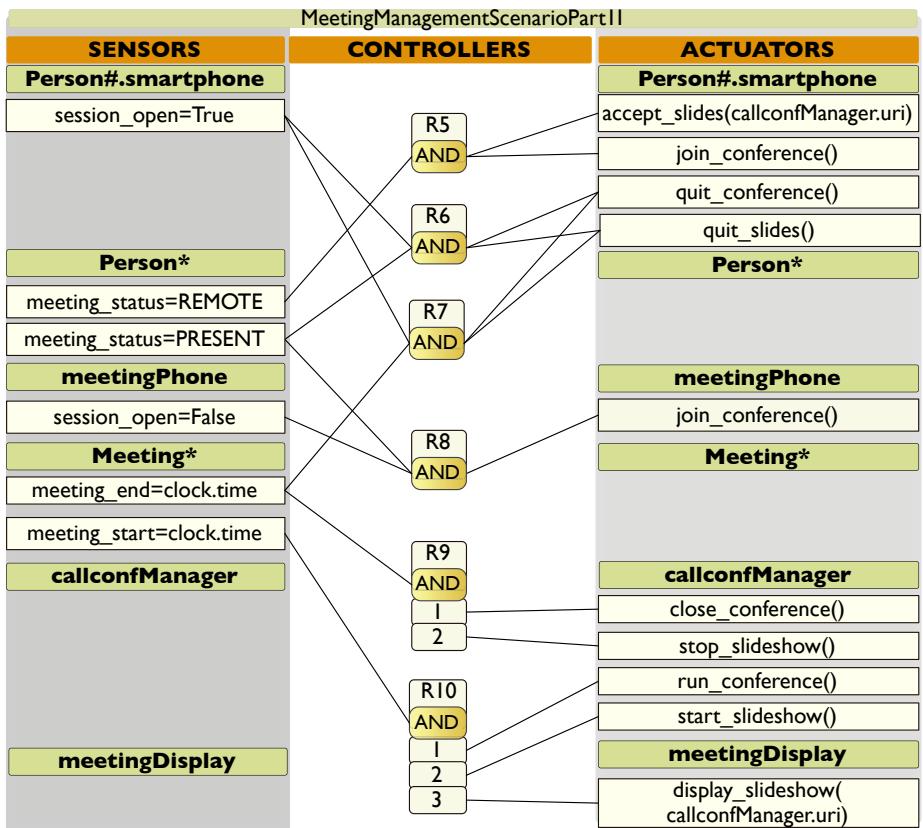


**Fig. 4.** An example program in the Pantagruel development environment (Part 1)

numbered in the controller column (*e.g.*, R1). A key feature of our approach is to drive the development of orchestration rules with respect to an environment description. In doing so, the development environment provides the programmer with contextual menus and one-the-fly verification to guide the definition of rules.

#### 4.1 Sections as Constituent Parts of a Rule

To further structure the definition of orchestration rules, the development panel is divided into horizontal sections, one for each entity instance or entity class involved in the application. This is illustrated by the first two sections of the meeting manager application shown in Figure 4. These sections orchestrate the *PresenceAgent* and *Meeting* entity classes. As well, the last section of Figure 4 is defined for the *meetingPhone* instance of the *SmartPhone* entity class.



**Fig. 5.** An example program in the Pantagruel development environment (Part 2)

Within a section, Pantagruel operations are in the scope of the corresponding entity class. For example, the `meeting_start` attribute, defined in the environment description, can be manipulated within the `Meeting` section. In contrast, the `time` attribute needs to be explicitly accessed via the `clock` entity.

## 4.2 Referring to Entity Instances

Because they orchestrate actual entities, orchestration rules need to refer to instances of entity classes deployed in a given environment. Doing so requires the ability to discover entities over which rules are defined. Rules thus need to be defined over and applied to all instances of a given entity class. Each of these instances may require rules to further refine the processing. As well, specific rules may be required for specific entity instances (*e.g.*, the phone of the meeting room).

To address these issues, we introduce notations to define a section that refers to all entity instances of an entity class. The name of such a section is composed of the entity class name (that starts with an uppercase letter) followed by the

‘\*’ symbol. When an orchestration rule includes an element (sensor or actuator) coming from such a section, it is executed over all the instances of the corresponding entity class. For example, Figure 4 includes the **Tag\*** section. One of the conditions defined in this section determines whether the tag of a person is located in the meeting room (*i.e.*, `location=ROOM`), acting as a filter towards collecting the set of meeting participants that are present. As we collect this set, we sometimes need to define rules that manipulate sub-parts of instances being collected. This is typically required to further refine the filtering of instances and trigger some actions. For example, when a person is present, we further want to test whether (s)he is participating to an upcoming meeting. To do so, we define a section over a sub-part of a tag entity that holds information about its owner. Specifically, the **Tag#.person** section of Figure 4 refers to the **person** sub-part of a tag being collected in the **Tag\*** section. A tag instance is collected if (1) it is present in the meeting room ( $2^{nd}$  condition in the **Tag\*** section), (2) the person’s name belongs to the list of participants of a meeting (condition in the **Tag#.person** section), (3) the meeting is occurring (both conditions in the **Meeting\*** section). As illustrated in the **Tag#.person** section, the ‘#’ notation is used to introduce a section that refers to each filtered element of an entity class. Elements of a class are filtered in a ‘\*’ section, as exemplified by the **Tag\*** section. In the **Tag#.person** section, the ‘#’ notation is also used to check the current person against the list of participants of the current meeting (*i.e.*, `Meeting#.participants`), collected in the **Meeting\*** section.

Lastly, Figure 4 illustrates a section for the **meetingPhone** entity instance (whose name starts with a lowercase letter). This is needed to operate this specific phone, when the meeting is over. Notice that, **meetingPhone** is started by the action of Rule R8 in Figure 5.

### 4.3 Defining Context Conditions

Sensors consist of conditions defined over context elements, whether constant, external or applicative. Pantagruel provides notations and operators to easily express conditions. Values of context elements can be tested with comparison and set operators. Attributes of entities are accessed using the conventional dot notations. When filtering an entity class, a condition may need to hold for all of its instances. Such condition is expressed by prefixing a predicate by the keyword **ALL** (**NONE** for the inverse). This keyword is illustrated in Rule R4 in Figure 4, where **meetingPhone** is closed when all tag owners have left the meeting room.

A specific construct called **changed** deals with external and applicative context elements. This construct turns true when the value of such attribute changes. As a result, the orchestration rules are completely insulated from the implementation details of the context change. They only focus on the logic to react to a context change.

### 4.4 Defining Actions

Context conditions are grouped with AND/OR operators, forming a controller. When a controller evaluates to true, either a unique action (*i.e.*, a method) is

executed, as in Rule R1, or several actions. In the latter case, the actions may be executed in any order (Rule R5) or sequentially (Rule R9). Actions may correspond to attribute assignments, typically needed to update an entity status, as in the `Tag#.person` section in Figure 4. Only write attributes can be assigned values. Volatile attributes have their value updated externally to Pantagruel. The action part of a rule may also involve method invocations, as required to operate entities from Pantagruel. These invocations have to conform to the type signature declared in the environment description. When the method of an instance is invoked, it may update one of its attributes as declared in the environment description. For example, when the `join_conference` method is invoked on `meetingPhone`, it may set the `session_open` to True, indicating that `meetingPhone` has joined an audio session.

## 5 Implementation

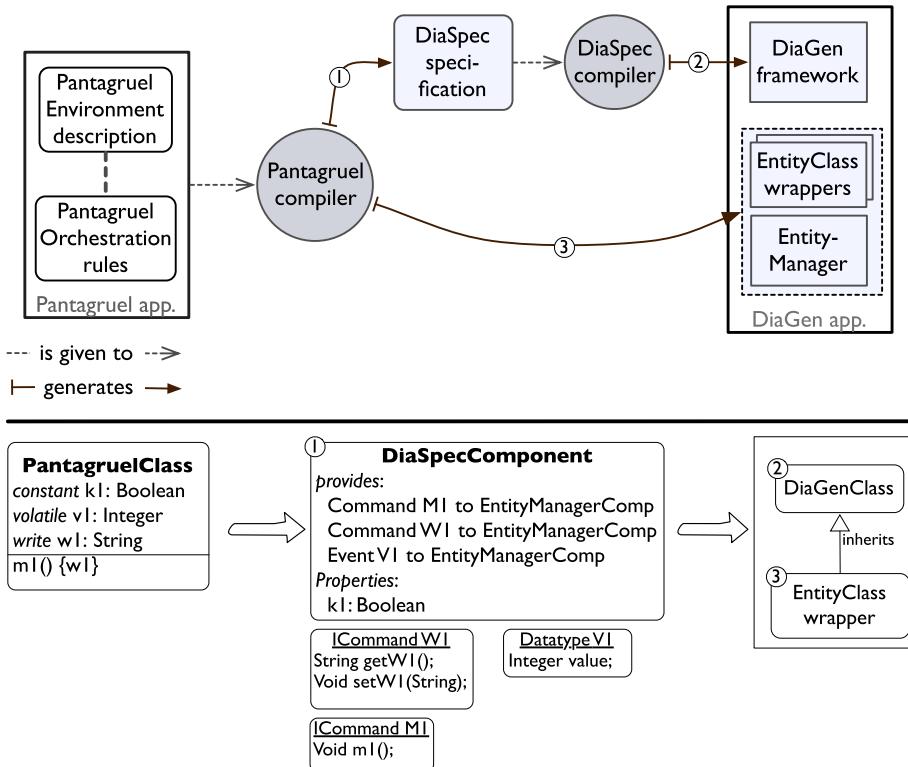
We now present an overview of the implementation of Pantagruel. This implementation is based on the denotational semantics of the language. This formal definition has been used to precisely specify the semantics of the orchestration rules with respect to a taxonomy of entities. As well, it is a basis to formulate various verifications of Pantagruel programs; we briefly discuss this topic at the end of this section.

Because of the nature of Pantagruel, its compilation is rather challenging, requiring to bridge the gap between a high-level language and some underlying middleware. This situation usually leads to an intricate compilation process, involving various stages to gradually map high-level abstractions into general-purpose ones.

In contrast, our strategy consists of leveraging a high-level software development tool. Specifically, our approach relies on DiaSpec [16], a general-purpose architecture description language (ADL), targeting distributed systems. This ADL integrates a component-and-connector architecture into Java [17]. It comes with DiaGen, a generator that produces programming frameworks, customized with respect to an ADL description. DiaSpec and its generator enable us to abstract over distributed systems technologies and mechanisms to interact with networked entities. A Pantagruel environment description is thus mapped into a DiaSpec description.

More specifically, the interaction modes offered by Pantagruel to communicate with entities are mapped into DiaSpec connectors for distributed components. To do so, volatile attributes are expressed in terms of DiaSpec events. That is, when an entity notifies an event, the volatile attribute of the corresponding Pantagruel entity is updated. Method invocations of Pantagruel entities are compiled into DiaSpec commands, which can be viewed as remote method invocations. Write attributes are mapped into getters and setters, implemented as DiaSpec commands.

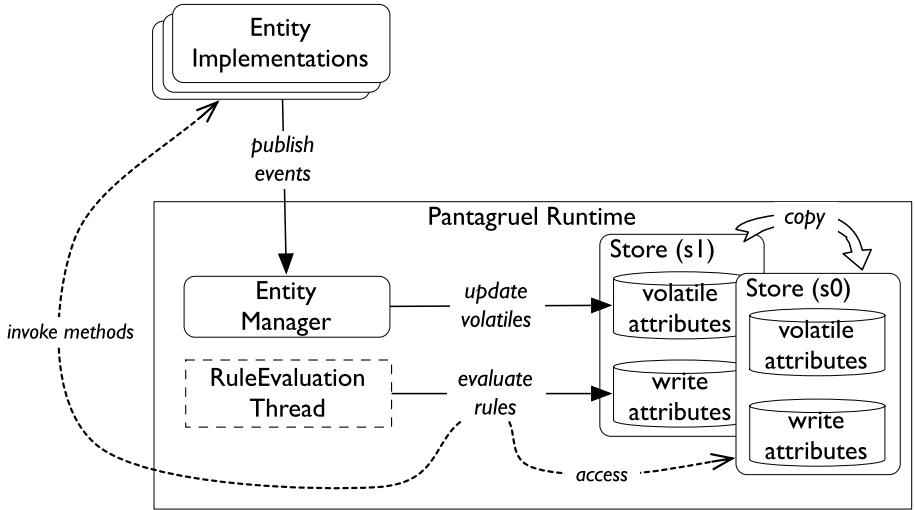
*Compilation.* The overall compilation process of Pantagruel is displayed in Figure 6 (upper part). The first step consists of mapping a Pantagruel



**Fig. 6.** Pantagruel compilation. Upper part: an overview of the compilation process. Lower part: example of compilation steps.

environment description into a DiaSpec description (Figure 6 – lower part). An entity class is compiled into a component declaration, mapping Pantagruel interaction modes into DiaSpec ones. When this mapping is completed, the ADL description is passed to DiaGen to produce a programming framework, dedicated to the input taxonomy (step 2). This framework includes support for the declared events and commands; it is generated once and for all for a given taxonomy. The third step of the compilation process consists of generating Java code from the Pantagruel rules, given the customized programming framework. Only this code needs to be regenerated and compiled when the Pantagruel rules change; the customized programming framework is re-used as is. The implementation of a Pantagruel application is completed by introducing wrappers for the environment entities. A wrapper implements the interface required by a Pantagruel entity class. It corresponds to a Java class implementing (1) methods that receive data from some entities and publish them as events, (2) methods that access entity functionalities, and (3) methods managing write attributes.

To generate the wrappers, we allow the taxonomy to be annotated with implementation declarations. These declarations map the taxonomy elements to



**Fig. 7.** Execution process

existing DiaSpec components and their interaction modes (that is, events or commands). During the compilation of a Pantagruel program, the generated DiaSpec specification is connected to the existing DiaSpec components. In doing so, we leverage the existing DiaSpec specifications for which wrapper implementations already exist.

*Execution.* Once compiled into Java, a Pantagruel application is deployed in the target pervasive computing environment. Executing an application amounts to evaluating rules given the current store, modeling the context of the pervasive computing environment, to produce a new store, reflecting the actions performed. A store consists of volatile attributes and write attributes. A detailed view of the execution process is depicted in Figure 7. At the center lies the RuleEvaluation thread that loops over the evaluation of the orchestration rules. At each iteration, the rules are executed with an initial store. When the predicates of a rule hold, its actions are triggered and their effects performed on the new store. Volatile attributes that change during an iteration are handled by the EntityManager component. This component uses the new store to update their value as their changes get notified by events. When the iteration is completed, the new store becomes the initial store of the next iteration. This dual-store strategy allows rules to be evaluated independently of the others, making Pantagruel programs easier to understand for the developers.

As can be noticed, DiaSpec enables the compilation of Pantagruel to be rather high level. As well, our approach makes it possible for Pantagruel to benefit from the numerous entities supported by DiaGen and installed in our Lab's smart space. Indeed, it gives Pantagruel access to a range of hardware and software entities.

To facilitate testing of Pantagruel programs, we leverage another tool from DiaSpec: a simulator for pervasive computing applications, called DiaSim [18], which has been developed by our research group. This simulator is coupled with a 2D-renderer. It allows the developer to create a visual layout of a physical environment, including representations of entities. The execution of a compiled Pantagruel program, equipped with simulated entities, is visualized and driven by a set of stimuli and their evolution rules. Existing simulated entities can be re-used by a Pantagruel program, as is done with wrapper implementations of actual entities.

*Verification.* Verification of Pantagruel programs is based on the formal definition of the language and facilitated by its dedicated nature. Specifically, Pantagruel programs can be represented as finite-state systems, making them amenable to model checking tools. Safety and liveness properties can be specified and proved for a Pantagruel program. To do so, we define a straightforward compilation process for Pantagruel programs into the TLA+ specification language [19] and use the TLC model checker [20] to verify properties. The semantics of TLA formulas is based on sequences of states, similar to the Pantagruel semantics. In essence, our verification process consists of three phases. Firstly, we generate a TLA+ specification from a Pantagruel program. Secondly, we provide abstraction functions that abstract over the possible states of write and volatile attributes to prevent an explosion of the state-space. Thirdly, we express a set of domain-specific properties as TLA+ formulas. As an example, a property for our meeting scenario can be specified as follows: a conference that was open on a smartphone (`join_conference`) must eventually be closed (`quit_conference`). In other words, we can formulate properties to ensure that, if some sensor conditions hold, then some actuators will eventually be triggered. Specifying and verifying such properties can greatly help developers to ensure the safety of their orchestration logic.

## 6 Preliminary Evaluation

We now present a preliminary evaluation of Pantagruel. This evaluation relies on a prototype of a graphical editor that we developed in the Eclipse IDE<sup>3</sup>. Figure 8 gives a snapshot of the Pantagruel graphical editor. It implements the coupling between the taxonomy and the orchestration logic, providing a graphical guide to build rules that conform to a taxonomy. Using this editor, we defined simple scenarios for three application areas: home automation, surveillance management, and professional communication management.

This experimental work is a first step towards determining Pantagruel's applicability. In particular, we want to assess Pantagruel's ability to fulfill four key requirements: (1) to model a variety of application areas, using the concepts of our taxonomy-based approach; (2) to model a range of applications for a given area; (3) to easily create variations of an application, driven by user preferences,

---

<sup>3</sup> <http://www.eclipse.org> - Eclipse Interface Development Environment.

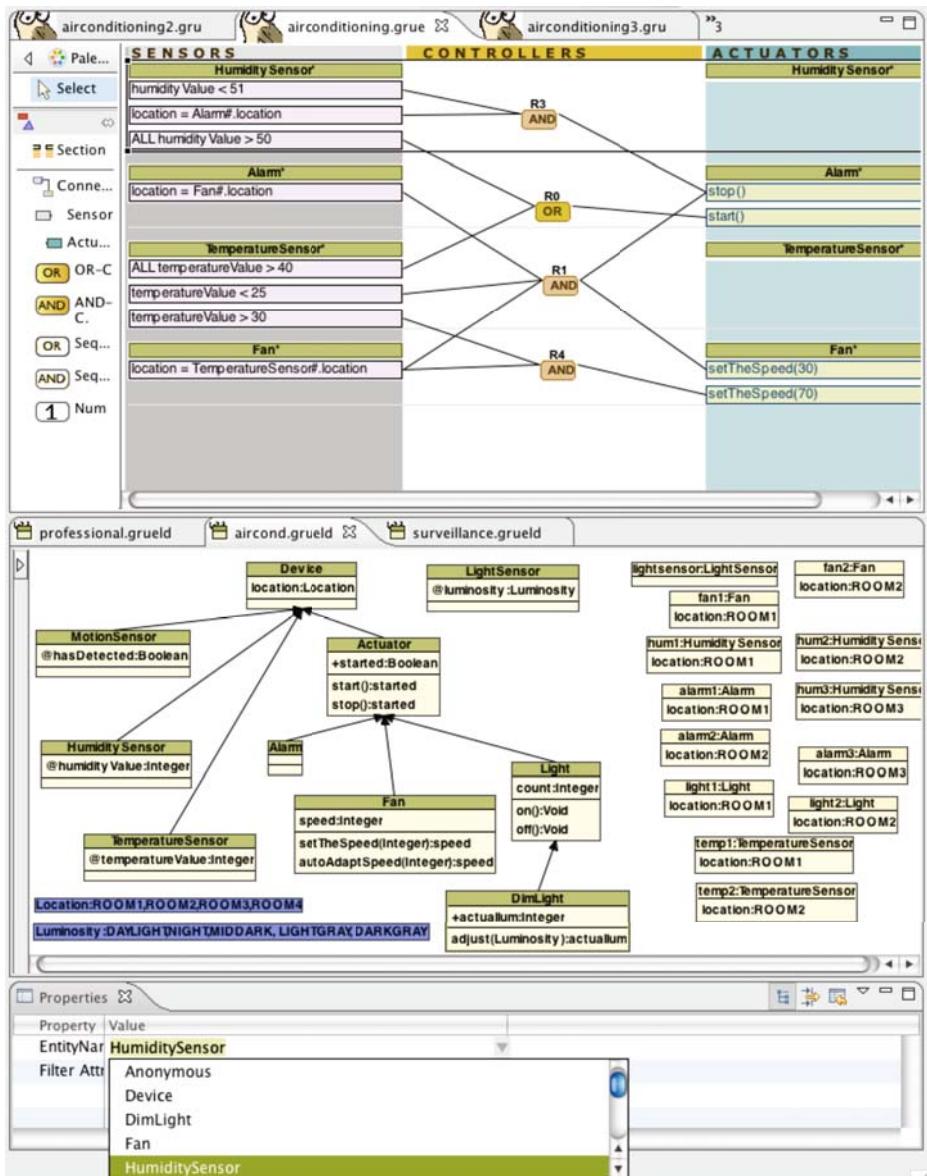


Fig. 8. A screenshot of the Pantagruel graphical editor

as discussed in our working example (Section 2); (4) to validate the insulation of Pantagruel programs from the heterogeneous implementations of entity classes. The latter requirement was partly addressed by the generation of wrappers presented in the previous section.

To evaluate the first three requirements, we developed three taxonomies for the above-mentioned application areas. These taxonomies factorize the knowledge on three different pervasive application areas. Their differences give a preliminary measure of the expressivity of our declarative language for taxonomies (requirement 1). Each taxonomy specifies the building blocks needed to develop a range of applications within an area (requirement 2). In the rest of this section, for each application area, we present a taxonomy and describe two different applications. We also give a flavor of the ease to adapt one of these applications to specific needs (requirement 3).

## 6.1 Home Automation

We introduce an application to manage the air conditioning and one to manage lights. They involve typical devices found in the home automation area.

These applications imply a set of sensors that capture the external conditions, and a set of actuators to react to these conditions to regulate indoor ambience, including temperature, luminosity, or humidity. A taxonomy for this aspect of the home automation area is composed of seven classes, as shown in Figure 8: `MotionSensor` with the `has_detected` volatile, `TemperatureSensor` with the `temperatureValue` volatile, `Fan`, `HumiditySensor`, `LightSensor`, `Light`, `DimLight` that refines `Light`, and `Alarm`. We also provide two datatypes: `Location` that specifies the location of sensors and lights, and `Luminosity` that gives a measure of the luminosity.

Our first application controls the fan speed according to the temperature measured by the temperature sensors whose room is given by the `Location` attribute, inherited from the `Device` class. Also, temperature sensors trigger alarms in rooms if the captured temperature exceeds a given threshold. This triggering is done by testing each instance (*e.g.*, `temperatureValue > 50`) of the `TemperatureSensor*` entity class.

The second application controls the lights of the rooms, according to the outside luminosity measured by a light sensor. For example, when the luminosity changes, the light intensity is adjusted accordingly. We also have a rule that controls the lights of a room according to its luminosity and whether presence is detected.

These applications define some basic tasks that involve entities sensing external events and actuating devices, like lights. The sensors are conveniently mapped into a unique volatile attribute. The rules are simple and readable in that they combine two or three sensors to trigger an actuator.

## 6.2 Building Surveillance

The building surveillance detects various forms of intrusions. The taxonomy for the building surveillance area consists of the following entity classes: `Clock`, `Light`, `Camera`, `Alarm`, `MailServer` and `RestrictedLocation`. We also reuse an entity class from the previous application area, namely `MotionSensor`. The camera takes pictures of an intruder. The `RestrictedLocation` class is associated

to each room, whose location is defined by the `loc` constant attribute. It defines the `is_restricted` write attribute that can be set to true to restrict the access of a given room.

The intrusion management application sends a message to the security guard when a sensor detected a motion outside the working hours. As well, it triggers the alarms, takes pictures and turns all the lights on. Doing so relies on two rules that respectively activate and deactivate the motion sensors depending on previously defined working hours.

The second surveillance manager monitors restricted rooms at specific times during the day. In case a restricted room is entered by an employee at a prohibited time, a message is sent to the guard. When accessible, the `is_restricted` attribute of a restricted room is set to false.

In this application, by appropriately defining the entity classes, we enable the definition of variations of applications. For example, instances of the `Restricted-Location` entity class can be parameterized to change the restrictions of the rooms. The `Clock` sensor can be configured to define new time intervals.

### 6.3 Professional Communication Management

We defined a taxonomy towards managing professional communications, namely, call transfer, when somebody is unavailable, and reminder, when a meeting is upcoming. In contrast with the previous taxonomies, this one essentially involves entities corresponding to software components. Our taxonomy contains the following set of classes: `PresenceAgent` defines the `status` volatile, indicating the user availability; `SmartPhone` consists of the `availability` write attribute, the `callreceived` volatile, and a method to control the phone ring; `TelephonyServer` has a forward-call method; `SMSServer` sends SMS messages; and, `Agenda` defines a `meeting` volatile, indicating a meeting status: upcoming, ongoing or completed. All instances of these entity classes, except the servers, are associated with a user (*e.g.*, Bob).

The first application is composed of three rules. One rule changes the availability status of a person to false if she is attending a meeting or her presence agent has a `BUSY` status. Another rule does the inverse when her agent status is `ONLINE` and no meeting is planned. A third rule invokes the `forwardSecretary` method of the telephony server with the caller name the original callee.

A second application sends a reminder to the participant of an upcoming meeting, using the `meeting` volatile of her agenda. It also turns off the smart phone ring when the meeting has started, and turns it back on at the end of the meeting.

### 6.4 Discussion

*Expressivity.* From this preliminary study we draw the following conclusions. Pantagruel fulfills the first requirement in that it makes it possible to define entities ranging from physical sensors, like a temperature sensor, to software components, like an agenda. Because of our model of entity interaction, the orchestration logic abstracts over the nature of the attributes (*i.e.*, volatile, write or

constant) that define the rule contexts, raising further the level of programming. We examined the Pantagruel programs written for the target application areas and observed that they had a small number of rules and did not illustrate any limitations. However, our applications remain simple; larger scale developments are needed to further assess Pantagruel's expressivity and scalability.

*Benefits of a visual language.* Visual programming languages and textual languages essentially differ in that visual languages offer a multi-dimensional representation of programs [21]. For example, one particular dimension is represented by spatial relationships between objects. In Pantagruel, we offer a spatial structuring of entities, rules, sensors, and actuators that is customized to the domain of pervasive computing. Carrying out a multi-dimensional representation of a program requires a close integration between a visual language and its programming environment. This integration facilitates the management of the constituent elements of a program, improving the development process.

Because it is driven by a taxonomy, the rule editor of Pantagruel allows to directly identify which entities are involved in an orchestration logic, and what purpose they serve. We plan to further emphasize this benefit by improving our graphical selection mechanisms to filter entities, rules, sensors and actuators.

Finally, the visual nature of Pantagruel should contribute to making this language accessible to novice programmers. In the future, we plan to explore the combination between the Pantagruel language and the DiaSim simulator to provide programmers with a better visual feedback of the behavior of their orchestration rules.

## 7 Related Work

Frameworks or middlewares have been proposed to separate the description of the environment entities from the application (*i.e.*, orchestration) logic. Kurkani *et al.* [22] describe entities using an XML-based representation, and develop the orchestration logic using a specification language. However, the relationship between the entities and the application logic is achieved via references to XML files. This approach is low level and makes verification difficult. In our approach the orchestration logic is driven by the entity class descriptions, which enable the verification of Pantagruel programs. In the Olympus programming model [6], the orchestration logic relies on an ontological description of entities. However programming the orchestration logic is done in a general-purpose programming language, which is not constrained by the ontological descriptions and may lead to programs that do not conform to these descriptions.

Visual prototyping tools are gaining interest within the pervasive computing community. The visual nature of these tools enables a simplified, intuitive representation of the orchestration logic. However, to the best of our knowledge, none of the tools reported in the literature propose a taxonomy-based approach to developing orchestration logic. As a consequence, existing approaches are either limited to a specific pervasive computing area or cannot be customized

with respect to a given area. For example, CAMP [10] is an end-user, rule-based tool. It is based on the magnetic poetry metaphor, proposing to visually compose sentences. However, its vocabulary is restricted to the home automation area. Another related tool is iCap [11], it provides a fixed set of generic entity classes: objects, activities, time, locations and people. Unlike our approach, these pre-defined entity classes cannot be extended with new attributes and methods. Moreover, iCap does not provide a uniform means to express rules over a group of entities besides a group of persons. In contrast, our approach provides syntactic abstractions to define filters on all instances of an entity class, as well as specific entities, enabling the orchestration logic to address a dynamically changing environment, as well as static entities.

The visual language VisualRDK [23] contrasts with the previous tools in that it targets novice or experienced programmers. VisualRDK is a programmatic approach, offering language-inspired constructs such as processes, conditional cascades and signals. These constructs mimic conventional programming, obscuring the domain-specific aspects of entity orchestration. Pantagruel differs from this approach in that rules are driven by the entities, connecting sensors to actuators, as reflected by the three-column panel of its development environment. Other visual prototyping tools like OSCAR [12] and Jigsaw [24] propose an approach to discovering, connecting and controlling services and devices, dedicated to domestic spaces. However, they offer limited expressivity to access the functionalities of the entities. Finally, tools based on the storyboard paradigm like Topiary [9] express the orchestration logic as control flows. However, Topiary does not allow scenarios where actions do not have a visual representation.

## 8 Conclusion and Future Work

Pervasive computing is reaching an increasing number of areas, creating a need to factorize knowledge about the entities that are relevant to each of these areas. This paper presents Pantagruel, an approach and a tool that integrate a taxonomical description of a pervasive computing environment into a visual programming language. Rules are developed using a sensor-controller-actuator paradigm, parameterized with respect to a taxonomy of entities. We have used Pantagruel to develop a number of scenarios, demonstrating the benefits of our taxonomy-based approach.

In the future, we will further evaluate our taxonomy-based approach by widening the scope of the target pervasive computing areas. To do so, we are collecting coordination scenarios that have been reported in the literature. Specifically, we plan to study the management of hospital patients [1], technological assistance to persons with intellectual deficiencies [25], and museum tour guide systems [3]. These studies will further assess how much is factorized by our taxonomy-based approach and how expressive is our rule-based language. While developing new scenarios, we intend to refine the syntax and semantics of our language.

Tackling these scenarios will go beyond the mere definition of a taxonomy and applications in these areas. In fact, we will test our Pantagruel programs

by leveraging DiaSim, our pervasive computing simulator [18]. In doing so, we are also enable to explore the expressivity of Pantagruel programs in areas that would otherwise be out of reach.

Another direction for future work is to conduct a usability study of Pantagruel. The goal is to determine what skills are needed to define a taxonomy in a given area and whether it can be defined by a domain expert without a programming background. Similarly, we want to determine what training is required to define coordination rules and whether these rules can be defined by non-programmers.

Finally, we are developing various analyses for Pantagruel programs to guarantee properties such as termination and non-interference of coordination rules. This work builds upon the formal semantics of Pantagruel. These verifications will be integrated in the Pantagruel development environment, providing early feedback to the developer.

## Acknowledgments

The authors would like to thank Alex Consel for introducing them to Game Blender, a sub-application of Blender dedicated to make games. Alex shared his enthusiasm for Game Blender and answered numerous questions about it. This tool has been a major source of inspiration for Pantagruel.

This work was partially funded by the *Conseil Régional d'Aquitaine*.

## References

1. Shulman, B.: RFID for Patient Flow Management in Emergency Unit. Technical report, IBM Corporation (2006), <http://www.ibm.com/news/fr/fr/2006/03/cp1851.html>
2. Keith Edwards, W., Grinter, R.E.: At home with ubiquitous computing: Seven challenges. In: Abowd, G.D., Brumitt, B., Shafer, S. (eds.) UbiComp 2001. LNCS, vol. 2201, pp. 256–272. Springer, Heidelberg (2001)
3. Kuflik, T., Sheidin, J., Jbara, S., Goren-Bar, D., Soffer, P., Stock, O., Zancanaro, M.: Supporting small groups in the museum by context-aware communication services. In: 12th Int'l. Conference on Intelligent User Interfaces (IUI), pp. 305–308. ACM, New York (2007)
4. Adkins, M., Kruse, J., Younger, R.: Ubiquitous computing: Omnipresent technology in support of network centric warfare. In: 35th Hawaii Int'l. Conference on System Sciences (HICSS), p. 40. IEEE Computer Society, Los Alamitos (2002)
5. Roman, M., Campbell, R.H.: Gaia: enabling active spaces. In: 9th ACM SIGOPS European Workshop, pp. 229–234. ACM, New York (2000)
6. Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R.H., Mickunas, M.D.: Olympus: A high-level programming model for pervasive computing environments. In: 3rd Int'l. Conference on Pervasive Computing and Communications (PerCom), pp. 7–16. IEEE Computer Society, Los Alamitos (2005)
7. Grimm, R.: One. world: Experiences with a pervasive computing architecture. IEEE Pervasive Computing 3(3), 22–30 (2004)
8. Garlan, D., Siewiorek, D.P., Steenkiste, P.: Project Aura: Toward distraction-free pervasive computing. IEEE Pervasive Computing 1, 22–31 (2002)

9. Li, Y., Hong, J.I., Landay, J.A.: Topiary: a tool for prototyping location-enhanced applications. In: 17th Symposium on User Interface Software and Technology (UIST), pp. 217–226. ACM, New York (2004)
10. Truong, K.N., Huang, E.M., Abowd, G.D.: CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In: Davies, N., Myatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 143–160. Springer, Heidelberg (2004)
11. Dey, A.K., Sohn, T., Streng, S., Kodama, J.: iCAP: Interactive prototyping of context-aware applications. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) PERVASIVE 2006. LNCS, vol. 3968, pp. 254–271. Springer, Heidelberg (2006)
12. Newman, M.W., Elliott, A., Smith, T.F.: Providing an integrated user experience of networked media, devices, and services through end-user composition. In: Indulska, J., Patterson, D.J., Rodden, T., Ott, M. (eds.) PERVASIVE 2008. LNCS, vol. 5013, pp. 213–227. Springer, Heidelberg (2008)
13. Pfeiffer Jr., J.J.: Altaira: A rule-based visual language for small mobile robots. *Journal of Visual Languages and Computing* 9(2), 127–150 (1998)
14. Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O., Repenning, A.: LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO programmable brick. In: 11th Symposium on Visual Languages (VL), pp. 172–179. IEEE Computer Society, Los Alamitos (1995)
15. Jouve, W., Lancia, J., Palix, N., Consel, C., Lawall, J.: High-level programming support for robust pervasive computing applications. In: 6th Int'l. Conference on Pervasive Computing and Communications (PerCom), pp. 252–255 (2008)
16. Jouve, W., Palix, N., Consel, C., Kadionik, P.: A SIP-based programming framework for advanced telephony applications. In: Schulzrinne, H., State, R., Niccolini, S. (eds.) IPTComm 2008. LNCS, vol. 5310, pp. 1–20. Springer, Heidelberg (2008)
17. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26(1), 70–93 (2000)
18. Jouve, W., Bruneau, J., Consel, C.: DiaSim: A parameterized simulator for pervasive computing applications. Technical report, INRIA/Labri (2008)
19. Lamport, L.: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3), 872–923 (1994)
20. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)
21. Burnett, M.M., McIntyre, D.W.: Visual programming - guest editors' introduction. *IEEE Computer* 28(3), 14–16 (1995)
22. Kulkarni, D., Tripathi, A.: Generative programming approach for building pervasive computing applications. In: 1st Int'l. Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (SEPCASE), p. 3. IEEE Computer Society, Los Alamitos (2007)
23. Weis, T., Knoll, M., Ulbrich, A., Muhl, G., Brandle, A.: Rapid prototyping for pervasive applications. *IEEE Pervasive Computing* 6(2), 76–84 (2007)
24. Humble, J., Crabtree, A., Hemmings, T., Åkesson, K.-P., Koleva, B., Rodden, T., Hansson, P.: Playing with the Bits user-configuration of ubiquitous domestic environments. In: Dey, A.K., Schmidt, A., McCarthy, J.F. (eds.) UbiComp 2003. LNCS, vol. 2864, pp. 256–263. Springer, Heidelberg (2003)
25. Svensk, A.: Design for cognitive assistance. In: Human Factors and Ergonomics Society Europe Annual Meeting (HFES) (2003)

# LEESA: Embedding Strategic and XPath-Like Object Structure Traversals in C++

Sumant Tambe and Aniruddha Gokhale

Electrical Engineering and Computer Science Department,  
Vanderbilt University, Nashville, TN, USA  
`{sutambe,gokhale}@dre.vanderbilt.edu`

**Abstract.** Traversals of heterogeneous object structures are the most common operations in *schema-first* applications where the three key issues are (1) separation of traversal specifications from type-specific actions, (2) expressiveness and reusability of traversal specifications, and (3) supporting structure-shy traversal specifications that require minimal adaptation in the face of schema evolution. This paper presents **L**anguage for **E**mbedded qu**E**ry and traver**S**Al (LEESA), which provides a generative programming approach to address the above issues. LEESA is an object structure traversal language embedded in C++. Using C++ templates, LEESA combines the expressiveness of XPath's axes-oriented traversal notation with the genericity and programmability of Strategic Programming. LEESA uses the object structure meta-information to statically optimize the traversals and check their compatibility against the schema. Moreover, a key usability issue of *domain-specific error reporting* in embedded DSL languages has been addressed in LEESA through a novel application of *Concepts*, which is an upcoming C++ standard (C++0x) feature. We present a quantitative evaluation of LEESA illustrating how it can significantly reduce the development efforts of schema-first applications.

## 1 Introduction

Compound data processing is commonly required in applications, such as program transformation, XML document processing, model interpretation and transformation. The data to be processed is often represented in memory as a heterogeneously typed hierarchical object structure in the form of either a tree (*e.g.*, XML document) or a graph (*e.g.*, models). The necessary type information that governs such object structures is encoded in a schema. For example, XML schema [1] specifications are used to capture the vocabulary of an XML document. Similarly, metamodels [2] serve as schema for domain-specific models. We categorize such applications as *schema-first* applications because at the core of their development lie one or more schemas.

The most widespread technique in contemporary object-oriented languages to organize these schema-first applications is a combination of the Composite and Visitor [3] design patterns where the composites represent the object structure

and visitors traverse it. Along with traversals, *iteration*, *selection*, *accumulation*, *sorting*, and *transformation* are other common operations performed on these object structures. In this paper, we deal with the most general form of object structures, *i.e.*, object graphs, unless stated otherwise.

Unfortunately, in many programming paradigms, object structure traversals are often implemented in a way such that the traversal logic and type-specific computations get entangled. Tangling in the functional programming paradigm has been identified in [4]. In object-oriented programming, when different traversals are needed for different visitors, the responsibility of traversal is imposed on the visitor class coupled with the type-specific computations. Such a tight coupling of traversal and type-specific computations adversely affects the reusability of the visitors and traversals equally.

To overcome the pitfalls of the Visitor pattern, domain-specific languages (DSL) that are specialized for the traversals of object structures have been proposed [5, 6]. These DSLs separate traversals from the type-specific computations using *external* representations of traversal rules and use a separate code generator to transform these rules into a conventional imperative language program. This two step process of obtaining executable traversals from external traversal specifications, however, has not enjoyed widespread use. Among the most important reasons [7, 8, 9, 10] hindering its adoption are (1) high upfront cost of the language and tool development, (2) their extension and maintenance overhead, and (3) the difficulty in integrating them with existing code-bases. For example, development of language tools such as a code generator requires the development of at least a lexical analyzer, parser, back-end code synthesizer and a pretty printer. Moreover, Mernik et al. [7] claim that language extension is hard to realize because most language processors are not designed with extension in mind. Finally, smooth integration with existing code-bases requires an ability of not only choosing a subset of available features but also incremental addition of those features in the existing code-base. External traversal DSLs, however, lack support for incremental addition as they tend to generate code in bulk rather than small segments that can be integrated at a finer granularity. Therefore, programmers often face a *all-or-nothing* predicament, which limits their adoption. *Pure embedding* is a promising approach to address these limitations of external DSLs.

Other prominent research on traversal DSLs have focused on Strategic Programming (SP) [4, 11, 12] and Adaptive Programming (AP) [13, 14] paradigms, which support advanced separation of traversal concerns from type-specific actions. SP is a language-independent generic programming technique that provides a design method for programmer-definable, reusable, generic traversal schemes. AP, on the other hand, uses static meta-information to optimize traversals and check their conformance with the schema. Both the paradigms allow “structure-shy” programming to support traversal specifications that are loosely coupled to the object structure. We believe that the benefits of SP and AP are critical to the success of a traversal DSL. Therefore, an approach that combines

them in the context of a pure embedded DSL while addressing the *integration challenge* will have the highest potential for widespread adoption.

To address the limitations in the current state-of-the-art, in this paper we present a *generative programming* [15] -based approach to developing a pure embedded DSL for specifying traversals over object graphs governed by a schema. We present an expression-based [8] pure embedded DSL in C++ called **Language for Embedded quEry and traverSAI** (LEESA), which leverages C++ templates and operator overloading to provide an intuitive notation for writing traversals. LEESA makes the following novel contributions:

- It provides a notation for traversal along several object structure axes, such as *child*, *parent*, *sibling*, *descendant*, and *ancestor*, which are akin to the XML programming idioms in XPath [16] – an XML query language. LEESA additionally allows composition of type-specific behavior over the axes-oriented traversals without tangling them together.
- It is a novel incarnation of SP using C++ templates, which provides a combinator style to develop programmer-definable, reusable, generic traversals akin to the classic SP language Stratego [17]. The novelty of LEESA’s incarnation of SP stems from its use of static meta-information to implement not only the regular behavior of (some) primitive SP combinators but also their customizations to prevent traversals into unnecessary substructures. As a result, efficient *descendant* axis traversals are possible while simultaneously maintaining the schema-conformance aspect of AP.
- One of the most vexing issues in embedded implementations of DSLs is the lack of mechanisms for intuitive *domain-specific error reporting*. LEESA addresses this issue by combining C++ template metaprogramming [18] with concept checking [19, 20] in novel ways to provide intuitive error messages in terms of *concept violations* when incompatible traversals are composed at compile-time.
- Finally, its embedded approach allows incremental integration of the above capabilities into the existing code-base. During our evaluation of LEESA’s capabilities, small segments of verbose traversal code were replaced by succinct LEESA expressions in a step by step fashion. We are not aware of any external C++ code generator that allows integration at comparable granularity and ease.

The remainder of the paper is organized as follows. Section 2 describes LEESA’s notation for object structure traversal and its support for strategic programming; Section 3 presents how we have realized the capabilities of LEESA; Section 4 describes how concept checking is used in novel ways to provide domain-specific error diagnostics; Section 5 evaluates the effectiveness of LEESA; Section 6 and Section 7 present related work and conclusions, respectively.

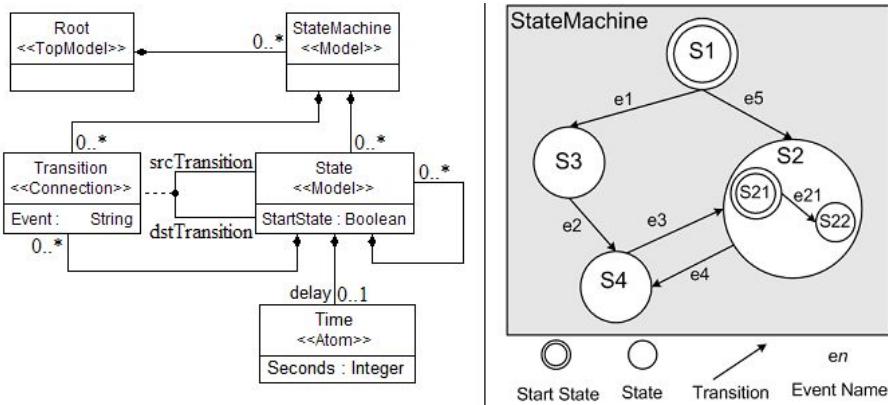
## 2 The LEESA Language Design

In this section we formally describe the syntax of LEESA and its underlying semantic model in terms of axes traversals. To better explain its syntax and

semantics, we use a running example of a domain-specific modeling language for hierarchical finite state machine (HFSM) described next.

## 2.1 Hierarchical Finite State Machine (HFSM) Language: A Case-Study

Figure 1 shows a metamodel of a HFSM language using a UML-like notation. Our HFSM metamodel consists of **StateMachines** with zero or more **States** having directional edges between them called **Transitions**. States can be marked as a “start state” using a boolean attribute. States may contain other states, transitions, and optionally a **Time** element. A **Transition** represents an association between two states, where the source state is in the **srcTransition** role and the destination state is in the **dstTransition** role with respect to a **Transition** as shown in Figure 1. **Time** is an indivisible modeling element (hence the stereotype `<<Atom>>`), which represents a user-definable delay in seconds. If it is absent, a default delay of 1 second is assumed. **Delay** represents the composition role of a **Time** object within a **State** object. All other composition relationships do not have any user-defined composition roles but rather a default role is assumed. The **Root** is a singleton that represents the root level model.



**Fig. 1.** Metamodel of Hierarchical Finite State Machine (HFSM) language (left) and a simple HFSM model (right)

To manipulate the instances of the HFSM language, C++ language bindings were obtained using a code generator. The generated code consists of five C++ classes: **Root**, **StateMachine**, **State**, **Transition**, and **Time** that capture the vocabulary and the relationships shown in the above metamodel. We use these classes throughout the paper to specify traversals listings.

## 2.2 An Axes-Oriented Notation for Object Structure Traversal

Designing an intuitive domain-specific notation for a DSL is central to achieving productivity improvements as domain-specific notations are closer to the

problem domain in question than the notation offered by general-purpose programming languages. The notation should be able to express the key abstractions and operations in the domain succinctly so that the DSL programs become more readable and maintainable than the programs written in general-purpose programming languages. For object structure traversal, the key abstractions are the objects and their typed collections while the basic operations performed are the navigation of associations and execution of type-specific actions.

When designing a notation for an embedded DSL, an important constraint imposed by the host language is to remain within the limits of the programmer-definable operator syntax offered by the host language. Quite often, trade-offs must be made to seek a balance between the expressiveness of the embedded notation against what is possible in a host language.

```

Statement   : Type { (Operator      Type)  |  (LRShift  visitor-object)  |
                  (LRShift      Action)  |  (>>  Members)           |
                  (>> | >>=  Association) }+
Type        : class-name '(' ')'
Operator    : LRShift | >>= | <<=
LRShift     : >> | <<
Action      : "Select" | "Sort" | "Unique" | "ForEach" | (and more ...)
Association : "Association" '(' class-name :: role-name ')'
Members     : "MembersOf" '(' Type { ','  Statement }+ ')'

```

**Listing 1.** Grammar of LEESA expressions

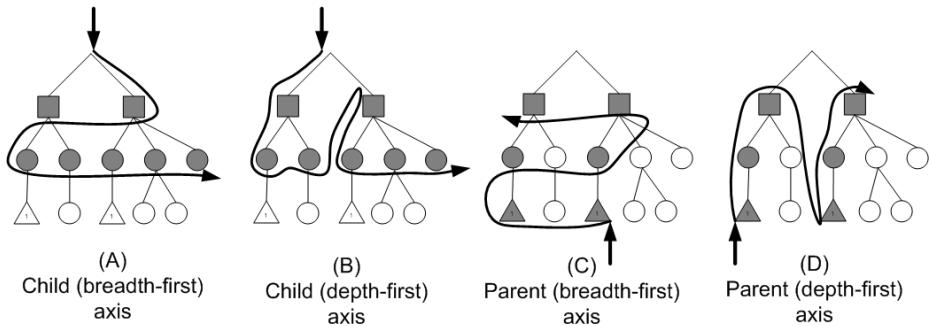
Listing 1 shows LEESA's syntax represented in the form of a grammar. *Statement* marks the beginning of a LEESA expression, which usually contains a series of *types*, *actions*, and visitor objects separated by *operators*. The first *type* in a LEESA statement determines the type of object where the traversal should begin. The four operators ( $\gg$ ,  $\ll$ ,  $\gg=$ ,  $\ll=$ ) are used to choose between *children* and *parent* axes and variations thereof. This traversal notation of LEESA resembles XPath's query syntax, however, unlike XPath, the LEESA expressions can be decorated with visitor objects, which modularize the type-specific actions away from the traversals.

The *association* production in Listing 1 represents traversal along user-defined association roles (captured in the metamodel) whereas *members* represent traversal along *sibling* axis. Actions are generic functions used to process the results of intermediate traversals. The parameters accepted by these actions, which are implemented as C++ function templates, are not shown. Instead, only the string literals sufficient for illustration are shown. Finally, instances of programmer-defined visitor classes can be added in the place of *visitor-object* that simply dispatch the type-specific actions. It conveniently allows accumulation of information during traversal without tangling the type-specific computations and the traversal specifications.

We now present concrete examples of LEESA expressions with their semantics in the context of the HFSM language case-study in Section 2.1.

**Table 1.** Child and parent axes traversal using LEESA (v can be replaced by an instance of a programmer-defined visitor class)

Axes	LEESA expressions and their semantics
(A) Child (breadth first)	<b>Root() &gt;&gt; StateMachine()</b> >> v >> <b>State()</b> >> v Visit <i>all</i> state machines followed by all their immediate children states.
(B) Child (depth first)	<b>Root() &gt;&gt;= StateMachine()</b> >> v >>= <b>State()</b> >> v Visit <i>a</i> state machine and all its immediate children states. Repeat this for the remaining state machines.
(C) Parent (breadth first)	<b>Time() &lt;&lt; v &lt;&lt; State()</b> << v << <b>StateMachine()</b> << v Visit a given set of time objects followed by their immediate parent states followed by their immediate parent state machines.
(D) Parent (depth first)	<b>Time() &lt;&lt; v &lt;&lt;= State()</b> << v <<= <b>StateMachine()</b> << v For a given set of time objects, visit <i>a</i> Time object followed by visit its parent state followed by visit its parent state machine. Repeat this for the remaining time objects.



**Fig. 2.** Outlines of child/parent axes traversals (Squares are statemachines, circles are states, triangles are time objects, and shaded shapes are visited)

**Child and Parent Axes.** Child and parent axes traversals are one of the most common operations performed on object structures. LEESA provides a succinct and expressive syntax in terms of “ $\gg$ ” and “ $\ll$ ” operators for child and parent axes traversals, respectively. Two variations, *breadth-first* and *depth-first*, of both the axes are also supported. Presence of the “ $=$ ” operator after the above operators turns a breadth-first strategy into a depth-first.<sup>1</sup> Table 1 shows four LEESA traversal expressions using child and parent axes notations. Figure 2 illustrates the graphical outlines corresponding to the examples shown in Table 1.

<sup>1</sup> In C++, “ $\ll=$ ” and “ $\gg=$ ” are bitwise shift left & assign and shift right & assign operators, respectively.

Breadth-first and depth-first variations of the axes traversal strategies are of particular interest here because of the ease of control over traversal provided by them. The breadth-first strategy, if applied successively (as in examples (a) and (c) in Table 1), visits all the instances of the specified type in a *group* before moving on to the next group of objects along an axis. Essentially, this strategy simulates multiple looping constructs in a sequence. The depth-first strategy, on the other hand, selects a single object of the specified type at a time, descends into it, executes the remaining traversal expression in the context of that single object, and repeats the same with the next object, if any. Therefore, successive application of the depth-first strategy (as in examples (b) and (d) in Table 1), traverses the edges of the object tree unlike the breadth-first strategy. Essentially, the depth-first strategy simulates nested looping constructs.

LEESA uses the Visitor [3] design pattern to organize the type-specific behavior while restricting traversals to the LEESA expressions only. To invoke type-specific computations, LEESA expressions can be decorated with instances of programmer-defined visitor classes as shown in Table 1. If a visitor object *v* is written after type *T*, LEESA invokes *v.Visit(t)* function on every collected object *t* of type *T*. LEESA expressions can be used not only for visitor dispatch but also for obtaining a collection of the objects of the type that appears last in the expression. Such a collection of objects can be processed using conventional C++. For instance, example (a) in Table 1 returns a set of **States** whereas example (c) returns a set of **StateMachines**.

**Descendant and Ancestor Axes.** LEESA supports *descendant* and *ancestor* axes traversal seamlessly in conjunction with child/parent axes traversals. For instance, Listing 2 shows a LEESA expression to obtain a set of **Time** objects that are recursively contained inside a **StateMachine**. This expression supports a form of structure-shy traversal in the sense that it does not explicitly specify the intermediate structural elements between the **StateMachine** and **Time**.

```
Root() >> StateMachine() >> DescendantsOf(StateMachine(), Time())
```

**Listing 2.** A LEESA expression showing descendant axis traversal

Two important issues arise in the design and implementation of the structure-shy traversal support described above. First, how are the objects of the target type located *efficiently* in a hierarchical object structure? and second, at what stage of development the programmer is notified of impossible traversal specifications? In the first case, for instance, it is inefficient to search for objects of the target type in composites that do not contain them. Whereas in the second case, it is erroneous to specify a target type that is not reachable from the start type. Section 3.3 and Section 4 present solutions to the efficiency and the error reporting issues, respectively.

**Sibling Axis.** Composition of multiple types of objects in a composite object is commonly observed in practice. For example, the HFSM language has a composite called **StateMachine** that consists of two types of children that are siblings

of each other: **State** and **Transition**. Support for object structure traversal in LEESA would not be complete unless support is provided for visiting multiple types of siblings in a programmer-defined order.

```
ProgrammerDefinedVisitor v;
Root() >> StateMachine() >> MembersOf(StateMachine(), State()      >> v,
                                             Transition() >> v)
```

**Listing 3.** A LEESA expression for traversing siblings: **States** and **Transitions**

Listing 3 shows an example of how LEESA supports sibling traversal. The sample expression visits all the **States** in a **StateMachine** before all the **Transitions**. The types of visited siblings and their order is programmer-definable. The **MembersOf** notation is designed to improve readability as its first parameter is the common parent type (*i.e.*, **StateMachine**) followed by a comma separated list of LEESA subexpressions for visiting the children in the given order. It effectively replaces multiple *for* loops written in a sequence where each *for* loop corresponds to a type of sibling.

**Association Axis.** LEESA supports traversals along two different kinds of user-defined associations. First, *named composition roles*, which use user-defined roles while traversing composition instead of the default composition role. For instance, in our HFSM modeling language, **Time** objects are composed using the **delay** composition role inside **States**. Second, *named associations* between different types of objects that turn tree-like object structures into graphs. For example, **Transition** is a user-defined association possible between any two **States** in the HFSM language described in Section 2.1. Moreover, **srcTransition** and **dstTransition** are two possible roles a **State** can be in with respect to a **Transition**.

LEESA provides a notation to traverse an association using the name of the association class (*i.e.*, **class-name** in Listing 1) and the desired role (*i.e.*, **role-name** in Listing 1). Listing 4 shows two independent LEESA expressions that traverse two different user-defined associations. The first expression returns a set of **Time** objects that are composed immediately inside the top-level **States**. The expression traverses the **delay** composition role defined between states and time objects. This feature allows differentiation (and selection) of children objects that are of the same type but associated with their parent with different composition roles.

```
Root() >> StateMachine() >> State() >> Association(State::delay) ... (1)
Root() >> StateMachine() >> Transition()
          >> Association(Transition::dstTransition) ... (2)
```

**Listing 4.** Traversing user-defined associations using LEESA

The second expression returns all the top-level states that have at least one incoming transition. Such a set can be conceptually visualized as a set of states

that are at the *destination* end of a transition. The second expression in Listing 4 up to `Transition()` yields a set of transitions that are the immediate children of `StateMachines`. The remaining expression to the right of it traverses the user-defined association `dstTransition` and returns `States` that are in the *destination* role with respect to every `Transition` in the previously obtained set.

In the above association-based traversals, the operator “`>>`” does not imply child axis traversal but instead represents continuation of the LEESA expression in a breadth-first manner. As described before, breadth-first strategy simulates loops in sequence. Use of “`>>=`” turns the breadth-first strategy over association axis into a depth-first strategy, which simulates nested loops. Expressions with associations can also be combined with visitor objects if role-specific actions are to be dispatched.

## 2.3 Programmer-Defined Processing of Intermediate Results Using Actions

Writing traversals over object structures often requires processing the intermediate results before the rest of the traversal is executed (*e.g.*, filtering objects that do not satisfy a programmer-defined predicate, or sorting objects using programmer-defined comparison functions). LEESA provides a set of *actions* that process the intermediate results produced by the earlier part of the traversal expression. These actions are in fact higher-order functions that take programmer-defined predicates or comparison functions as parameters and apply them on a collection of objects.

```
int comparator (State, State) { ... } // A C++ comparator function
bool predicate (Time) { ... } // A C++ predicate function
Root() >> StateMachine() >> State() >> Sort(State(), comparator)
                    >> Time() >> Select(Time(), predicate)
```

**Listing 5.** A LEESA expression with actions to process intermediate results

Listing 5 shows a LEESA expression that uses two predefined actions: `Sort` and `Select`. The `Sort` function, as the name suggests, sorts a collection using a programmer-defined comparator. `Select` filters out objects that do not satisfy the programmer-defined predicate. The result of the traversal in Listing 5 is a set of `Time` objects, however, the intermediate results are processed by the actions before traversing composition relationships further. `Sort` and `Select` are examples of higher-order functions that accept conventional functions as parameters as well as stateful objects that behave like functions, commonly known as *functors*.

LEESA supports about a dozen different actions (*e.g.*, `Unique`, `ForEach`) and more actions can be defined by the programmers and incorporated into LEESA expressions if needed. The efforts needed to add a new action are proportional to adding a new class template and a global overloaded operator function template.

## 2.4 Generic, Recursive, and Reusable Traversals Using Strategic Programming

Although LEESA's axes traversal operators ( $\gg$ ,  $\ll$ ,  $\gg=$ ,  $\ll=$ ) are reusable for writing traversals across different schemas, they force the programmers to commit to the vocabulary of the schema and therefore the traversal expressions (as whole) cannot be reused. Moreover, LEESA's axes traversal notation discussed so far lacked support for *recursive* traversal, which is important for a wide spectrum of domain-specific modeling languages that support *hierarchical* constructs. For example, our case study of HFSM modeling language requires recursive traversal to visit deeply nested states.

A desirable solution should not only support recursive traversals but also enable higher-level reuse of *traversal schemes* while providing complete control over traversal. Traversal schemes are higher level control patterns (*e.g.*, top-down, bottom-up, depth-first, etc.) for traversal over heterogeneously typed object structures. Strategic Programming (SP) [4, 11, 12] is a well known generic programming idiom based on programmer-definable (recursive or otherwise) traversal abstractions that allow separation of type-specific actions from reusable traversal schemes. SP also provides a *design method* for developing reusable traversal functionality based on so called *strategies*. Therefore, based on the observation that LEESA shares this goal with that of SP, we adopted the SP design method and created a new incarnation of SP on top of LEESA's axes traversal notation. Next, we describe how LEESA leverages the SP design method to meet its goal of supporting generic, recursive, and reusable traversals. For a detailed description of the foundations of SP, we suggest reading [4, 11, 12] to the readers.

LEESA's incarnation of the SP design method is based on a small set of *combinators* that can be used to construct new combinators from the given ones. By combinators we mean reusable C++ class templates capturing basic functionality that can be composed in different ways to obtain new functionality. The basic combinators supported in LEESA are summarized in Table 2. This set of combinators is inspired by the *strategy* primitives of the term rewriting language Stratego [17].

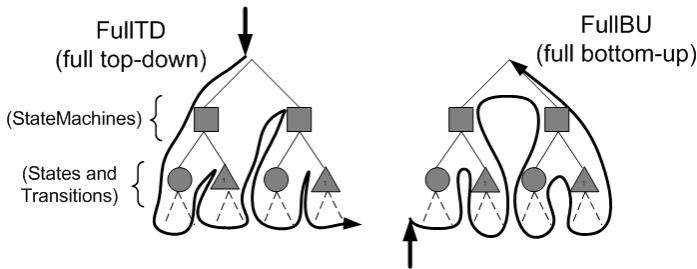
**Table 2.** The set of basic class template combinators

Primitive combinators	Description
Identity	Returns its input datum without change.
Fail	Always throws an exception indicating a failure.
Sequence $\langle S_1, S_2 \rangle$	Invokes strategies $S_1$ and $S_2$ in sequence on its input datum.
Choice $\langle S_1, S_2 \rangle$	Invokes strategy $S_2$ on its input datum only if the invocation of $S_1$ fails.
All $\langle S \rangle$	Invokes strategy $S$ on all the immediate children of its input datum.
One $\langle S \rangle$	Stops invocation of strategy $S$ after its first success on one of the children of its input datum.

```
FullTD<Strategy> = Sequence<Strategy, All<FullTD> >
FullBU<Strategy> = Sequence<All<FullBU>, Strategy>
```

**Listing 6.** Pseudo-definitions of the class templates of the predefined traversal schemes (Strategy = Any primitive combinator or combination thereof, TD = top-down, BU = bottom-up)

**All** and **One** are *one-layer traversal* combinator, which can be used to obtain full traversal control, including recursion. Although none of the basic combinator are recursive, higher-level traversal schemes built using the basic combinator can be recursive. For instance, Listing 6 shows a subset of predefined higher-level traversal schemes in LEESA that are recursive. The (pseudo-) definition of **FullTD** (full top-down) means that the parameter **Strategy** is applied at the root of the incoming datum and then it applies itself recursively to all the immediate children of the root, which can be of heterogeneous types. Figure 3 shows a graphical illustration of **FullTD** and **FullBU** (full bottom-up) traversal schemes. Section 3.3 describes the actual C++ implementation of the primitives and the recursive schemes in detail.



**Fig. 3.** Graphical illustration of FullTD and FullBU traversal schemes. (Squares, circles, and triangles represent objects of different types).

```
Root() >> StateMachine() >> FullTD(StateMachine(), VisitStrategy(v));
```

**Listing 7.** Combining axes traversal with strategic programming in LEESA. (v can be replaced by a programmer-defined visitor.)

Listing 7 shows how **FullTD** recursive traversal scheme can be used to perform full top-down traversal starting from a **StateMachine**. Note that the heterogeneously typed substructures (**State**, **Transition**, and **Time**) of the **StateMachine** are not mentioned in the expression. However, they are incorporated in the traversal automatically using the static meta-information in the metamodel. This is achieved by *externalizing* the static meta-information in a form that is understood by the C++ compiler and in turn the LEESA expressions. Later in Section 3.2 we describe a process of externalizing the static meta-information

from the metamodel (schema) and in Section 3.3 we show how it is used for substructure traversal.

Finally, the `VisitStrategy` in Listing 7 is a predefined LEESA strategy that can not only be configured with programmer-defined visitor objects but can also be replaced by other programmer-defined strategies. We envision that LEESA’s `VisitStrategy` will be used predominantly because it supports the *hierarchical visitor* pattern [21] to keep track of depth during traversal. This pattern is based on a pair of type-specific actions: `Visit` and `Leave`. The prior one is invoked while *entering* a non-leaf node and the latter one is invoked while *leaving* it. To keep track of depth, the visitor typically maintains an internal stack where the `Visit` function does a “push” operation and `Leave` function does a “pop”.

## 2.5 Schema Compatibility Checking

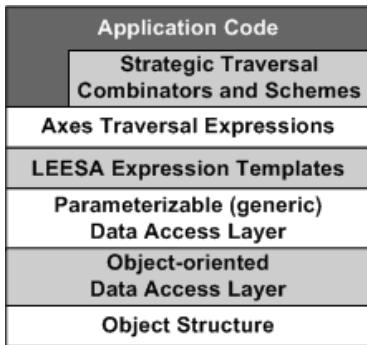
Every syntactically correct traversal expression in LEESA is statically checked against the schema for type errors and any violations are reported back to the programmer. Broadly, LEESA supports four kinds of checks based on the types and actions participating in the expression. First, only the types representing the vocabulary of the schema are allowed in a LEESA expression. The visitor instances are an exception to this rule. Second, impossible traversal specifications are rejected where there is no way of reaching the elements of a specified type along the axis used in the expression. For example, the child-axis operators ( $\gg$ ,  $\gg=$ ) require (immediate) parent/child relationship between the participating types whereas `DescendantsOf` requires a transitive closure of the child relationship. Third, the argument type of the intermediate results processing actions must match to that of the result returned by the previous expression. Finally, the result type of the action must be a type from the schema if the expression is continued further. Section 4 describes in detail how we have implemented schema compatibility checking using C++ Concepts.

## 3 The Implementation of LEESA

In this section we present LEESA’s layered software architecture, the software process of obtaining the static meta-information from the schema, and how we have implemented the strategic traversal combinators in LEESA.

### 3.1 The Layered Architecture of LEESA

Figure 4 shows LEESA’s layered architecture. At the bottom is the in-memory *object structure*, which could be a tree or a graph. An *object-oriented data access layer* is a layer of abstraction over the object structure, which provides schema-specific, type-safe interfaces for iteratively accessing the elements in the object structure. Often, a code generator is used to generate language bindings (usually a set of classes) that model the vocabulary. Several different types of code generators such as XML schema compilers [22] and domain-specific modeling



**Fig. 4.** Layered View of LEESA’s Architecture (Shading of blocks shown for aesthetic reasons only)

tool-suites [23] are available that generate schema-specific object-oriented data access layer from the static meta-information.

To support generic traversals, the schema-specific object-oriented data access layer must be adapted to make it suitable to work with the generic implementation of LEESA’s C++ templates. The *parameterizable data access layer* is a thin generic wrapper that achieves this. It treats the classes that model the vocabulary as type parameters and hides the schema-specific interfaces of the classes. This layer exposes a small generic interface, say, `getChildren`, to obtain the children of a specific type from a composite object and say, `getParent`, to obtain the parent of an object. For example, using C++ templates, obtaining children of type T of an object of type U could be implemented<sup>2</sup> as `U.getChildren<T>()`, where U and T could be any two classes modeling the vocabulary that have parent/child relationship. This layer can also be generated automatically from the object structure schema.

*Expression Templates* [24] is the key idea behind embedding LEESA’s traversal expressions in C++. Using operator overloading, expression templates enable *lazy evaluation* of C++ expressions, which is otherwise not supported natively in C++. Lazy evaluation allows expressions – rather than their results – to be passed as arguments to functions to extract results lazily when needed. LEESA overloads the `>>`, `<<`, `>>=`, and `<<=` operators using the design method of expression templates to give embedded traversal expressions a look and feel of XPath’s axes-oriented traversal specifications. Moreover, LEESA expressions can be passed to other generic functions as arguments to extract results lazily. LEESA’s expression templates map the traversal expressions embedded in a C++ program onto the parameterizable data access layer. They raise the level of abstraction by hiding away the iterative process of accessing objects and instead focus only on the *relevant* types in the vocabulary and different strategies (breadth-first and depth-first) of traversal. LEESA’s expression templates

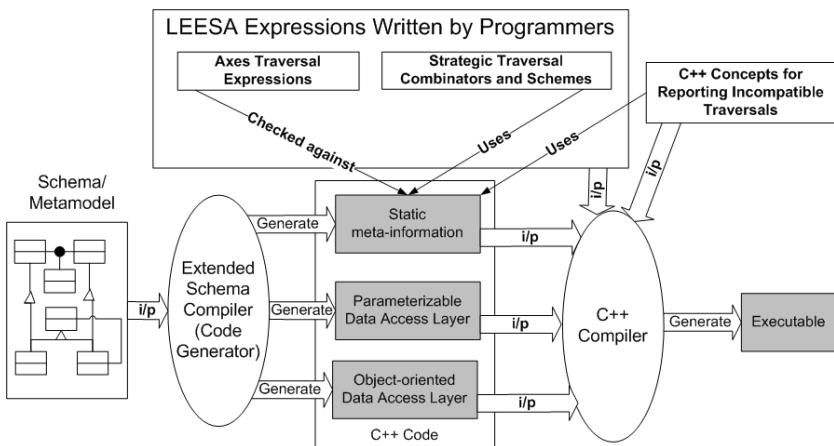
<sup>2</sup> A widely supported, standard C++ feature called “template explicit specialization” could be used.

are independent of the underlying vocabulary. Schema-specific traversals are obtained by instantiating them with schema-specific classes. For more details on LEESA's expression templates, including an example, the readers are directed to our previous work [25].

Finally, LEESA programmers use the *axes traversal expressions* and *strategic traversal combinators and schemes* to write their traversals as described in Section 2. The axes traversal expressions are based on LEESA's expression templates. The strategic traversal combinators use an externalized representation of the static meta-information for their generic implementation. Below we describe the process of externalizing the static meta-information.

### 3.2 Externalizing Static Meta-information

Figure 5 shows the software process of developing a schema-first application using LEESA. The object-oriented data access layer, parameterizable data access layer, and the static meta-information are generated from the schema using a code generator. Conventional [23, 22] code generators for language-specific bindings generate the object-oriented data access layer only, but for this paper we extended the Universal Data Model (UDM) [23] – a tool-suite for developing domain-specific modeling languages (DSML) – to generate the parameterizable data access layer and the static meta-information. The cost of extending UDM is amortized over the number of schema-first applications developed using LEESA. While the static meta-information is used for generic implementations of the primitive strategic combinators, C++ Concepts [19, 20] shown in Figure 5 are used to check the compatibility of LEESA expressions with the schema and report the errors back to the programmer at compile-time. C++ Concepts allow the error messages to be succinct and intuitive. Such a diagnosis capability is



**Fig. 5.** The software process of developing a schema-first application using LEESA. (Ovals are tools whereas shaded rectangular blocks represent generated code).

of high practical importance as it catches programmer mistakes much earlier in the development lifecycle by providing an additional layer of safety.

The Boost C++ template metaprogramming library (MPL) [18] has been used as a vehicle to represent the static meta-information in LEESA. It provides easy to use, readable, and portable mechanisms for implementing metaprograms in C++. MPL has become a de-facto standard for metaprogramming in C++ with a collection of extensible compile-time algorithms, typelists, and metafunctions. Typelists encapsulate zero or more C++ types (programmer-defined or otherwise) in a way that can be manipulated at compile-time using MPL metafunctions.

**Using Boost MPL to Externalize the Static Meta-information.** The static meta-information (partial) of the HFSM metamodel (Section 2.1) captured using Boost MPL typelists is shown below.

```
class StateMachine {
    typedef mpl::vector < State, Transition > Children;
};

class State {
    typedef mpl::vector < State, Transition, Time > Children;
};

class Transition {
    typedef mpl::vector < > Children           // Same as class Time
};                                         // empty

mpl::contains <StateMachine::Children, State>::value //...(1) true
mpl::front   <State::Children>::type                //...(2) class State
mpl::pop_front <State::Children>::type               //...(3) mpl::vector<Transition, Time>
```

Each class has an associated type called `Children`, which is a MPL typelist implemented using `mpl::vector`. The typelist contains a list of types that are children of its host type. A MPL metafunction called `mpl::contains` has been used to check existence of a type in a MPL typelist. For example, the statement indicated by (1) above checks whether typelist `StateMachine::Children` contains type `State` in it or not. It results in a compile-time constant *true* value. Metafunctions `mpl::front` and `mpl::pop_front`, indicated by (2) and (3), are semantically equivalent to “car” and “cdr” list manipulation functions in Lisp. While `mpl::front` returns the first type in the typelist, `mpl::pop_front` removes the first type and returns the remaining typelist.

We leverage this metaprogramming support provided by MPL to represent children, parent, and descendant axes meta-information in C++. We have extended the UDM tool-suite to generate Boost MPL typelists that capture the static meta-information of these axes.

### 3.3 The Implementation of Strategic Traversal Schemes

In LEESA’s implementation of SP, `All` and `One` are *generative* one-layer combinatorics because their use requires mentioning the type of only the start element where the strategy application begins. The children and descendant (in case of recursive traversal schemes) types of the start type are automatically incorporated into the traversal using the externalized static meta-information and the LEESA’s metaprograms that iterate over it.

```

template <class Strategy>
class All {
    Strategy strategy_>;
public:
    All (Strategy s) : strategy_(s) {} // Constructor
    template <class T>
    void apply (T arg) { // Every strategy implements this member template function.
        // If T::Children typelist is empty, calls (B) otherwise calls (A)
        children(arg, typename T::Children());
    }
private:
    template <class T, class Children>
    void children(T arg, Children) {
        typedef typename mpl::front<Children>::type Head; // ..... (A)
        typedef typename mpl::pop_front<Children>::type Tail; // ... (2)
        for_each c in arg.getChildren<Head>() // ... (3)
            strategy_.apply(c);
        children(arg, Tail()); // ... (4)
    }
    template <class T>
    void children(T, mpl::vector<> /* empty typelist */) {} // .... (B)
};

-----
template <class S1, class S2>
class SEQ {
    S1 s1_; S2 s2_;
public:
    SEQ(S1 s1, S2 s2)
        : s1_(s1), s2_(s2) {}
    template <class T>
    void apply (T arg) {
        s1_.apply(arg);
        s2_.apply(arg);
    }
};

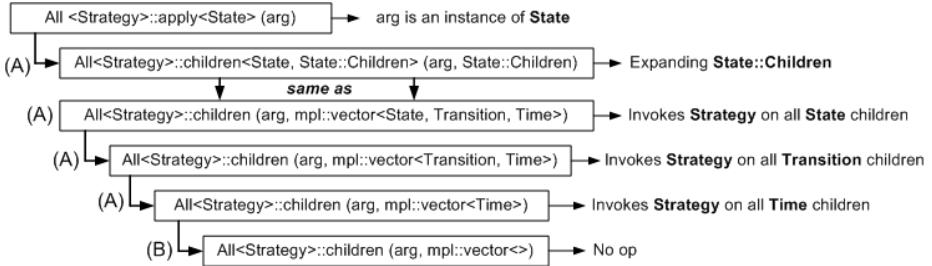
template <class Strategy>
class FullTD {
    Strategy st_;
public:
    FullTD(Strategy s) : st_(s) {}
    template <class T>
    void apply (T arg) {
        All<FullTD> all(*this);
        SEQ<Strategy, All<FullTD> > seq(st_, all);
        seq.apply(arg);
    }
};

```

**Listing 8.** C++ implementations of `All` and `SEQ` (Sequence) primitive combinators and the `FullTD` recursive traversal scheme

Listing 8 shows the C++ implementation of the `All` and `Sequence` primitive combinators and the `FullTD` recursive traversal scheme in LEESA. `All` is a class template that accepts `Strategy` as a type parameter, which could be instantiations of other combinators or other instantiations of `All` itself. Execution of `All` begins at the `apply` function, which delegates execution to another member template function called `children`. `All::children` is instantiated as many times as there are children of type `T`. From the `T::Children` typelist, repeated instantiation of the `children` member template function are obtained using the metaprogram indicated by statements (1), (2), and (4) in Listing 8.

Similar to list processing in functional languages, statement (1) yields the first type (`Head`) in the typelist whereas statement (2) yields the remaining typelist (`Tail`). Statement (4) is a compile-time recursive call to itself but with `Tail` as its second parameter. This compile-time recursion terminates only when `Tail` becomes empty after successive application of `mpl::pop_front` metafunction. When `Tail` is an empty typelist, `children` function marked by (B) is invoked terminating the compile-time recursion. Figure 6 shows graphical illustration of this recursive instantiation process. Multiple recursive instantiations of function `children` are shown in the order they are created with progressively smaller and



**Fig. 6.** Compile-time recursive instantiations of the `children` function starting at `All<Strategy>::apply<State>(arg)` when `arg` is of type `State`

smaller typelist as its second parameter. Finally, the statement marked as (3) is using the parameterizable data access interface `T::getChildren`, which returns all the `Head` type children of `arg`.

**Efficient Descendant Axis Traversal.** Compile-time customizations of the primitive combinator `All` and in turn `FullTD` traversal scheme are used for efficient implementation of the *descendant* axis traversal. LEESA can prevent traversals into unnecessary substructures by controlling the types that are visited during recursive traversal of `FullTD` and `FullBU` schemes. LEESA customizes the behavior of the `All` primitive combinator using the descendant types information that is obtained from the schema. The `T::Children` typelist in the `All::apply` function is manipulated using C++ template metaprogramming such that the schema types that have no way of reaching the objects of the target type are eliminated before invoking the `All::children` function. This is achieved using compile-time boolean queries over the list of descendant types implemented using MPL's metafunction `mpl::contains` as described in Section 3.2. All these metaprograms are completely encapsulated inside the C++ class templates that implement recursive traversal schemes and are not exposed to the programmers.

While static meta-information can be used for efficient traversal, the same meta-information can be used to check the LEESA expressions for their compatibility with the schema. We describe that next.

## 4 Domain-Specific Error Reporting Using C++ Concepts

In DSL literature [7, 8], embedded DSLs have been criticized for their lack of support for domain-specific error reporting. The importance of intuitive error messages should not be underestimated as it directly affects the programmer's effectiveness in locating and correcting errors in a DSL program. This issue is all the more important for embedded DSLs since their compiler is the same as the host language compiler, and hence the error reports are often in terms of the host language artifacts instead of domain-specific artifacts that are relevant to the problem. Moreover, for embedded DSLs in C++ that are implemented

using templates, the problem is further exacerbated because templates lack early modular (separate) type-checking.

#### 4.1 Early Type-Checking of C++ Templates Using Concepts

C++ Concepts [19] have been proposed in the latest C++ programming language standard, C++0x [26], to address the problem of late type-checking of templates during compilation. Concepts express the syntactic and semantic behavior of types and constrain the type parameters in a C++ template, which are otherwise unconstrained. Concepts allow separate type-checking of template definitions from their uses, which makes templates easier to use and easier to compile. The set of constraints on one or more types are referred to as *Concepts*. Concepts describe not only the functions and operators that the types must support but also other accessible types called *associated types*. The types that satisfy the requirements of a concept are said to *model* that concept. When a concept constrained C++ template is instantiated with a type that does not model the concept, an error message indicating the failure of the concept and the type that violates it are shown at the call site in the source code. An experimental support for C++ Concepts has been implemented in the ConceptGCC [27] compiler<sup>3</sup>.

#### 4.2 Schema Compatibility Checking Using Concepts and Metaprogramming

We have defined several C++ Concepts in LEESA that must be satisfied by different types participating in a LEESA expression. These Concepts are related primarily to child, parent, descendant, and ancestor axes traversals and the invocation of actions for intermediate results processing. For example, each type in a child axis traversal expression must model a `ParentChildConcept` with respect to its preceding type. An implementation of the `ParentChildConcept` is shown below.

```
concept ParentChildConcept <typename Parent, typename Child> {
    typename Children = typename Parent::Children;
    typename IsChild = typename mpl::contains<Children, Child>::type;
    requires std::SameType<IsChild, true_type>;
};
```

The concept is parameterized with two types and essentially requires that the `Child` type be present in the list of children of the `Parent` type. This is achieved by (1) obtaining the result of the application of MPL metafunction `contains` on `Parent`'s associated type `Children` and (2) enforcing the type of the result to be the same as `true_type`, which signifies success. If the Concept does not hold, a short error message is produced stating the failure of the Concept and the types that violate it. The error is reported at the first occurrence of the type that violates it regardless of the length of the expression. As the length of

<sup>3</sup> Library level support for concept checking is available [20] for pre-C++0x compilers.

the erroneous LEESA expression grows, the error output grows linearly due to increasing size of the recursively constructed type using expression templates. However, the reason and location are always stated distinctly in the form of concept violation.

For example, consider a LEESA expression, “`StateMachine() >> Time()`”, which is incorrect with respect to the metamodel of the HFSM modeling language because `Time` is not an immediate child of `StateMachine` and therefore, does not satisfy the `ParentChildConcept` described before. Below, we have shown the actual error message produced by the ConceptGCC [27] compiler, which is only four lines long, and clearly states the reason and the location (both on the fourth line) of the error.

```
t.cc: In function 'int main()':
t.cc:99: error: no match for 'operator>>' in 'StateMachine() >> Time()'
t.cc:85: note: candidates are: R LEESA::operator>>(const L&, const R&
                  [with L = StateMachine, R = Time] <requirements>
t.cc:99: note: no concept map for requirement
                  'LEESA::ParentChildConcept<StateMachine, Time>'
```

## 5 Evaluation

In this section, we present quantitative results on LEESA’s effectiveness in reducing efforts while programming traversals compared to the third generation object-oriented languages.

**Experimental setup.** We conducted the experiments using our open-source domain-specific modeling tool-suite: CoSMIC.<sup>4</sup> CoSMIC is a collection of domain-specific modeling languages (DSML), interpreters, code generators, and model-to-model transformations developed using Generic Modeling Environment (GME) [28], which is a meta-programmable tool for developing DSMLs. CoSMIC’s DSMLs are used for developing distributed applications based on component-based middleware. For instance, the Platform Independent Component Modeling Language (PICML) [29] is one of the largest DSMLs in CoSMIC for modeling key artifacts in all the life-cycle phases of a component-based application, such as interface specification, component implementation, hierarchical composition of components, and deployment. A PICML model may contain up to 300 different types of objects. Also, PICML has over a dozen model interpreters that generate XML descriptors pertaining to different application life-cycle stages. All these interpreters are implemented in C++ using UDM as the underlying object-oriented data access layer.

**Methodology.** The objective of our evaluation methodology is to show the reduction in programming efforts needed to implement commonly observed traversal patterns using LEESA over traditional iterative constructs.

To enable this comparison, we refactored and reimplemented the traversal related parts of PICML’s deployment descriptor generator using LEESA. This generator exercises the widest variety of traversal patterns applicable to PICML.

---

<sup>4</sup> <http://www.dre.vanderbilt.edu/cosmic>

**Table 3.** Reduction in code size (# of lines) due to the replacement of common traversal patterns by LEESA expressions

Traversal Pattern	Axis	Occurrences	Original #lines (average)	#Lines using LEESA (average)
A single loop iterating over a list of objects	Child	11	8.45	1.45
	Association	6	7.50	1.33
5 sequential loops iterating over siblings	Sibling	3	41.33	6
2 Nested loops	Child	2	16	1
Traversal-only visit functions	Child	3	11	0
Leaf-node accumulation using depth-first	Descendant	2	43.5	4.5
Total traversal code	-	All	414 (absolute)	53 (absolute)

It amounts to little over 2,000 lines<sup>5</sup> of C++ source code (LOC) out of which 414 (about 21%) LOC perform traversals. It is organized using the Visitor [3] pattern where the accept methods in the object structure classes are non-iterating and the entire traversal logic along with the type-specific actions are encapsulated inside a monolithic visitor class. Table 3 shows the traversal patterns we identified in the generator. We replaced these patterns with their equivalent constructs in LEESA. This procedure required some refactoring of the original code.

**Analysis of results.** Table 3 shows a significant reduction in the code size due to LEESA’s succinct traversal notation. As expected, the highest reduction (by ratio) in code size was observed when two ad-hoc implementations of depth-first search (*e.g.*, searching nested components in a hierarchical component assembly) were replaced by LEESA’s adaptive expressions traversing the descendant axis. However, the highest number of reduction in terms of the absolute LOC (114 lines) was observed in the frequently occurring traversal pattern of a *single loop*. Cumulatively, leveraging LEESA resulted in 87.2% reduction in traversal code in the deployment descriptor generator. We expect similar results in other applications of LEESA.

**Incremental Adoption of LEESA.** It is worth noting here that due to its pure embedded approach, applying LEESA in the existing model traversal programs is considerably simpler than external DSLs that generate code in bulk. Incremental refactoring of the original code-base was possible by replacing one traversal pattern at a time while being confident that the replacement is not changing the behavior in any unexpected ways. Such incremental refactoring using *external* traversal DSLs that use a code generator would be extremely hard, if not impossible. Our pure embedded DSL approach in LEESA allows us to distance ourselves from such *all-or-nothing* predicament, which could

<sup>5</sup> The number of lines of source code is measured excluding comments and blank lines.

potentially be a serious practical limitation. We expect that a large number of existing C++ applications that use XML data-binding [22] can start benefiting from LEESA using this incremental approach provided their XML schema compilers are extended to generate the parameterizable data access layer and the meta-information.

## 6 Related Work

In this section we place LEESA in the context of a sampling of the most relevant research efforts in the area of object structure traversal.

XPath 2.0 [16] is a structure-shy XML query language that allows node selection in a XML document using downward (children, descendant), upward (parent, ancestor), and sideways (sibling) axes. In general, XPath supports more powerful node selection expressions than LEESA using its untyped unconstrained (*i.e.* `axis::*`) axes. XPath’s formal semantics describe how XML schema could be used for *static type analysis* to detect certain type errors and to perform optimizations. However, contemporary XPath programming APIs for C++ use string encoded expressions, which are not checked against the schema at compile-time. Moreover, unlike XPath, type-specific behavior can be composed over the axes-oriented traversals using LEESA.

Adaptive Programming (AP) [13,14] specifies structure-shy traversals in terms of milestone classes composed using predicates, such as *from*, *to*, *through*, and *bypass*. It uses static meta-information to optimize traversals as well as to check their compatibility against the schema. While LEESA focuses on accumulation of nodes using its axes-oriented notation and programmability of traversals using its strategic combinator style, AP focuses on collection of unique paths in the object graph specified using the above mentioned predicates. The use of visitors in LEESA to modularize type-specific actions is similar in spirit to the code wrappers in AP.

Strategic Programming (SP) [4, 17], which began as a term rewriting [17] language has evolved into a language-interparadigmatic style of programming traversals and has been incarnated in several other contexts, such as functional [12], object-oriented [11], and embedded [10]. The strategic traversal expressions in LEESA are based on a new embedded incarnation of SP in an *imperative* language, C++. Unlike [10], however, no compiler extension is necessary. Also, all the expressions are *statically* checked against the schema, unlike visitor combinators [11].

Scrap++ [30] presents a C++ templates-based approach for implementing Haskell’s “Scrap Your Boilerplate” (SYB) design pattern, which is remarkably similar to SP. Scrap++’s approach depends on recursive traversal combinators, a one-layer traversal, and a type extension of the basic computations. However, LEESA’s approach is different in many significant ways. First, unlike LEESA, the objective of Scrap++ is to mimic Haskell’s SYB and therefore does not provide an intuitive axes traversal notation. Second, LEESA presents a software process for generating schema-specific meta-information that is used during compilation

for generating traversals as well as compatibility checking. Third, SYB lacks parental and sibling contexts. Finally, no technique is provided in Scrap++ to produce intuitive error messages.

Lämmel et al. [31] present a way of realizing adaptive programming predicates (*e.g.*, *from*, *to*, *through*, and *bypass*) by composing SP primitive combinators and traversal schemes. Due to a lack of static type information, their simulation of AP in terms of SP lacks important aspects of AP, such as static checking and avoiding unnecessary traversal into substructures. LEESA, on the other hand, uses the externalized meta-information to not only statically check the traversals but also makes them efficient.

Static meta-information has also been exploited by Cunha et al. [32] in the transformation of structure-shy XPath and SP programs for statically optimizing them. Both the approaches eliminate unnecessary traversals into substructures, however, no transformation is necessary in the case of LEESA. Instead, the behaviors of `All` and `One` primitive combinators are customized at compile-time to improve efficiency. Moreover, LEESA’s structure-shy traversals support mutually recursive types, unlike [32].

Lämmel [33] sketches an encoding of XPath-like axes (downward, upward, and sideways) using strategic function combinators in the SYB style. LEESA is similar to this work because both the approaches suggest an improvement of XPath-like set of axes with support for strategic, recursive traversal abstractions and provide a way of performing schema-conformance checking. The key differences are the improved efficiency of the *descendant* axis traversal in case of LEESA, its domain-specific error reporting capability, and its use of an imperative, object-oriented language as opposed to Haskell, which is a pure functional language.

Gray et al. [6] and Ovlinger et al. [5] present an approach in which traversal specifications are written in a specialized language separate from the basic computations. A code generator is used to transform the traversal specifications into imperative code based on the Visitor pattern. This approach is, however, heavyweight compared to the embedded approach because it incurs high cost of the development and maintenance of the language processor.

Language Integrated Query (LINQ) [34] is a Microsoft .NET technology that supports SQL-like queries natively in a program to search, project and filter data in arrays, XML, relational databases, and other third-party data sources. “LINQ to XSD” promises to add much needed typed XML programming support over its predecessor “LINQ to XML.” LINQ, however, does not support strategic combinator style like LEESA. The Object Constraint Language (OCL) [35] is a declarative language for describing well-formedness constraints and traversals over object structures represented using UML class graphs. OCL, however, does not support *side-effects* (*i.e.*, object structure transformations are not possible).

Czarnecki et al. [8] compare staged interpreter techniques in MetaOCaml with the template-based techniques in Template Haskell and C++ to implement embedded DSLs. Two approaches – *type-driven* and *expression-driven* – of implementing an embedded DSL in C++ are presented. Within this context, our

previous work [25] presents LEESA’s expression-driven pure embedding approach. Spirit<sup>6</sup> and Blitz++<sup>7</sup> are two other prominent examples of expression-driven embedded DSLs in C++ for recursive descent parsing and scientific computing, respectively. Although LEESA shares the implementation technique of expression templates with them, strategic and XPath-like axes-oriented traversals cannot be developed using Spirit or Blitz++.

## 7 Conclusion

In this paper we presented a case for *pure embedding* in C++ as an effective way of implementing a DSL particularly in the domain of object structure traversal where mature implementations of iterative data access layer abound. While many of the existing embedded DSLs perform poorly with respect to domain-specific error reporting, our novel approach of fusing together C++ Concepts and compile-time type manipulation using template metaprogramming allows us to report impossible traversals by terminating compilation with short and intuitive error messages. We believe this technique is applicable to other embedded DSLs in C++ irrespective of their domain.

To show the feasibility of our approach, we developed **L**anguage for **E**mbedded **q**u**E**ry and **t**raver**S**Al (LEESA), which is a pure embedded DSL in C++ for object structure traversal. LEESA improves the modularity of the traversal programs by separating the knowledge of the object structure from the type-specific computations. LEESA adopts the strategic combinator style to provide a library of generic reusable traversal schemes. With an eye on “structure shyness”, LEESA supports XPath-like traversal axes to focus only on the relevant types of richly structured data to shield the programs from the effects of schema evolution. Our contribution lies in combining these powerful traversal techniques without sacrificing efficiency and schema-conformance checking.

LEESA is available for download in open-source at <http://www.dre.vanderbilt.edu/cosmic>.

**Acknowledgment.** We are grateful to Endre Magyari for his implementation support to enhance the UDM code generator. We also thank the DSL’09 program committee and our shepherd Dr. Ralf Lämmel for providing detailed reviews.

## References

1. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N., et al.: XML Schema Part 1: Structures. W3C Recommendation (2001)
2. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer 30(4), 110–112 (1997)

---

<sup>6</sup> <http://spirit.sourceforge.net>

<sup>7</sup> <http://www.oonumerics.org/blitz>

3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
4. Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming, p. 18 (October 15, 2002), <http://www.cwi.nl/~ralf>
5. Ovlinger, J., Wand, M.: A Language for Specifying Recursive Traversals of Object Structures. *SIGPLAN Not.* 34(10), 70–81 (1999)
6. Gray, J., Karsai, G.: An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In: 36th Hawaiian International Conference on System Sciences (HICSS), pp. 325–334 (2003)
7. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. *ACM Computing Surveys* 37(4), 316–344 (2005)
8. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. In: *Domain Specific Program Generation 2004*, pp. 51–72 (2004)
9. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell (2003)
10. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) *RTA 2007*. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
11. Visser, J.: Visitor Combination and Traversal Control. In: *OOPSLA 2001: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 270–282 (2001)
12. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) *PADL 2002*. LNCS, vol. 2257, pp. 137–154. Springer, Heidelberg (2002)
13. Lieberherr, K.J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company (1996)
14. Lieberherr, K., Patt-shamir, B.: Traversals of Object Structures: Specification and Efficient Implementation. *ACM Trans. Program. Lang. Syst.*, 370–412 (1997)
15. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading (2000)
16. World Wide Web Consortium (W3C): XML Path Language (XPath), Version 2.0, W3C Recommendation (January 2007), <http://www.w3.org/TR/xpath20>
17. Visser, E., Benaissa, Z., Tolmach, A.: Building Program Optimizers with Rewriting Strategies. In: *Proceedings of the International Conference on Functional Programming (ICFP 1998)*, pp. 13–26. ACM Press, New York (1998)
18. Abrahams, D., Gurtovoy, A.: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, Reading (2004)
19. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 291–310 (2006)
20. Siek, J.G., Lumsdaine, A.: C++ Concept Checking. *Dr. Dobb's J.* 26(6), 64–70 (2001)
21. Portland Pattern Repository WikiWikiWeb: Hierarchical Visitor Pattern (2005), <http://c2.com/cgi/wiki?HierarchicalVisitorPattern>
22. Simeoni, F., Lievens, D., Connor, R., Manghi, P.: Language Bindings to XML. *IEEE Internet Computing*, 19–27 (2003)

23. Magyari, E., Bakay, A., Lang, A., Paka, T., Vizhanyo, A., Agrawal, A., Karsai, G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In: The 3rd OOPSLA Workshop on Domain-Specific Modeling (October 2003)
24. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (1995)
25. Tambe, S., Gokhale, A.: An Embedded Declarative Language for Hierarchical Object Structure Traversal. In: 2nd International Workshop on Domain-Specific Program Development (DSPD) (October 2008)
26. Becker, P.: Standard for Programming Language C++. Working Draft, N2798=08-0308, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (October 2008)
27. Gregor, D.: ConceptGCC: Concept Extensions for C++ (August 2008), <http://www.generic-programming.org/software/ConceptGCC>
28. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer 34(11), 44–51 (2001)
29. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In: RTAS 2005, pp. 190–199 (2005)
30. Munkby, G., Priesnitz, A., Schupp, S., Zalewski, M.: Scrap++: Scrap Your Boilerplate in C++. In: WGP 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming, pp. 66–75 (2006)
31. Lämmel, R., Visser, E., Visser, J.: Strategic programming meets adaptive programming. In: Proc. Aspect-Oriented Software Development (AOSD), pp. 168–177 (2003)
32. Alcino, C., Joost, V.: Transformation of Structure-Shy Programs: Applied to XPath Queries and Strategic Functions. In: PEPM 2007: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pp. 11–20 (2007)
33. Lämmel, R.: Scrap Your Boilerplate with XPath-like Combinators. In: POPL 2007: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 137–142 (2007)
34. Hejlsberg, A., et al.: Language Integrated Query (LINQ), <http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx>
35. Object Management Group: Unified Modeling Language: OCL version 2.0 Final Adopted Specification. OMG Document ptc/03-10-14 edn. (October 2003)

# Unit Testing for Domain-Specific Languages

Hui Wu<sup>1</sup>, Jeff Gray<sup>1</sup>, and Marjan Mernik<sup>2</sup>

<sup>1</sup> Department of Computer and Information Sciences,  
University of Alabama at Birmingham,  
Birmingham, Alabama USA  
`{wuh, gray}@cis.uab.edu`

<sup>2</sup> Faculty of Electrical Engineering and Computer Science,  
University of Maribor, Maribor, Slovenia  
`marjan.mernik@uni-mb.si`

**Abstract.** Domain-specific languages (DSLs) offer several advantages by providing idioms that are similar to the abstractions found in a specific problem domain. However, a challenge is that tool support for DSLs is lacking when compared to the capabilities offered in general-purpose languages (GPLs), such as Java and C++. For example, support for unit testing a DSL program is absent and debuggers for DSLs are rare. This limits the ability of a developer to discover the existence of software errors and to locate them in a DSL program. Currently, software developers using a DSL are generally forced to test and debug their DSL programs using available GPL tools, rather than tools that are informed by the domain abstractions at the DSL level. This reduces the utility of DSL adoption and minimizes the benefits of working with higher abstractions, which can bring into question the suitability of using DSLs in the development process. This paper introduces our initial investigation into a unit testing framework that can be customized for specific DSLs through a reusable mapping of GPL testing tool functionality. We provide examples from two different DSL categories that serve as case studies demonstrating the possibilities of a unit testing engine for DSLs.

**Keywords:** Domain-specific languages, unit testing, tool generation.

## 1 Introduction

Tool support is an important factor toward determining the success and adoption of any software development paradigm. Software development using domain-specific languages (DSLs) is no exception. This paper discusses the current lack of basic tool support for common tasks that a DSL programmer may expect to perform. The paper motivates the need for DSL testing tools and introduces an initial prototype framework that supports unit testing of DSL programs.

The need to test DSL programs is motivated by the recent trend in end-user programming. Scaffidi et al estimate that there are several million end-user programmers [29]. End-user programmers are more likely to introduce software errors than professional programmers because they lack software development training and proper tool

support [16]. As observed in several industry studies, individual examples of software errors have been very costly [18], [30]. For instance, it has been estimated that software failures collectively contribute to over \$60 billion in unreported losses per year [32]. Likewise, without the availability of software development tools, the final products of end-user programming can also be dangerous [16]. The proper programming tools (e.g., editor, compiler, test engine, and debugger) are needed for end-users to improve the integrity of the products they develop. With a large pool of end-user developers, and the rising cost of software failures, it is imperative that end-users be provided with tools that allow them to detect and find software errors at an abstraction level that is familiar to them.

## 1.1 Benefits of DSL Adoption

Despite advances in programming languages and run-time platforms, most software is developed at a low-level of abstraction relative to the concepts and concerns within the problem space of an application domain. DSLs assist end-users in describing solutions in their work domains [41]. A DSL is a programming language targeted toward a particular problem domain rather than providing general solutions for many domains [23], [38]. DSLs have also been shown to assist in software maintenance whereby end-users can directly use the DSLs to make required routine modifications [4], [36]. DSLs can help end-users and professional programmers write a software solution in a more concise, descriptive, and platform-independent way. It is often the case that domain experts are not familiar with general-purpose languages (GPLs) [31] in order to solve their domain problems; however, they may be more comfortable with addressing their domain problems through a DSL that is closer to the abstractions of their own domain knowledge. A language that closely represents key domain abstractions may also permit the construction of a more concise program that is also easier to test for correctness compared to the same intention expressed in a GPL. This paper demonstrates the idea of DSL testing using two small DSLs that could be adopted by end-users with little programming experience.

In addition to the benefits offered to end-user programmers, there are also advantages of DSL usage by professionally trained developers. The benefits of using DSLs within a software development project are increased flexibility, productivity, reliability, and usability, which have been shown through empirical evaluation on numerous case studies [9], [21], [23], [40]. Popular examples of DSLs include the languages used to specify grammars in parser generators like CUP [8] or ANTLR (ANother Tool for Language Recognition) [1]. Other examples include the Swing User-interface Language (SWUL), which is a DSL to construct a Java Swing user interface [6]; Structured Query Language (SQL) is a DSL to access and manipulate databases; HTML is a DSL that serves as a markup language for creating web pages.

## 1.2 Challenges of DSL Tool Implementation

A common practice for DSL implementation is to translate a single DSL construct into several GPL constructs [23], and then reuse the associated GPL tools to provide the infrastructure for interpreting or executing a DSL program. In the research described in this paper, we adopt this popular practice of source-to-source transformation to translate a

DSL to an existing GPL, such as Java or C++, which have well-developed programming tools. Translating a DSL to an existing GPL is a popular implementation approach because the underlying tools of the converted GPL can be reused (e.g., compiler, profiler, testing engine and debugger). Although direct reuse of the existing GPL tools offers several benefits, a GPL tool does not provide the proper abstractions that are needed by DSL programmers. An approach that hides the underlying use of the GPL tools and removes the accidental complexities that cause the abstraction mismatch between the DSL and GPL is needed. The goal of DSL tool implementation is to allow users of a language to work only at the abstraction level of the DSL semantics, while not being concerned about the complexities of the solution as realized in the tools of a supporting GPL. The situation is similar to the desire to have tools for a GPL at the level of the programming language, rather than underlying machine code (e.g., a debugger that is at the machine code level is useless to most GPL programmers). A goal is complete tool support at the proper abstraction level, regardless of how the tool's execution semantics are eventually realized.

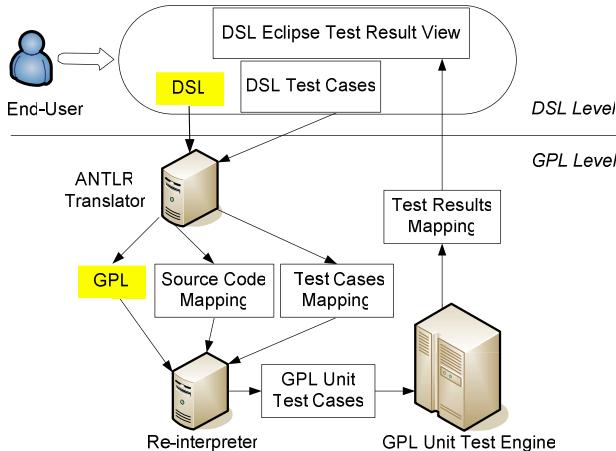
The construction of a DSL compiler or interpreter is only the first piece of the needed toolchain. A DSL programmer also needs tools to discover the existence of software errors and locate them in a DSL program. The paucity of such tools can be one of the major factors that may prevent wider acceptance of DSLs in the software industry. Our previous work [43] focused on the generation of debuggers for DSLs through adaptations to the DSL grammar. This paper follows a similar path that considers the generation of unit testing engines for DSLs.

### 1.3 Tool Support for Testing DSL Programs

The research described in this paper focuses on a framework that assists in customizing a tool that tests a DSL program's behavior. In order to assess the benefits of our approach, we define a unit test script that is customized to consider the domain-specific characteristics for different types of DSLs. To guide developers in testing their DSL programs, we adopt the traditional unit testing concepts such as defining the expected value of a test case, test assertions, and test result reports.

Building DSL testing tools from scratch for each new DSL is time consuming, error prone, and costly. In this paper, we show that such tools can be implemented by a generalization of a testing tools framework implemented within Eclipse (Figure 1) through reuse of existing GPL tool functionality. Figure 1 highlights the architecture of a DSL unit testing framework; we have previously developed a similar framework for DSL debuggers [43]. A key technique of the framework is a mapping process that records the correspondence between the DSL code and the generated GPL code. The translator that defines the mapping between the DSL and GPL levels is written in ANTLR, which is a lexer and parser generator [1]. This translator generates GPL code and source code mapping information that can be used to determine which line of the DSL code is mapped to the corresponding segment of generated GPL code.

The correspondence between DSL testing actions and GPL testing actions are given by pre-defined mapping algorithms that also can be specialized to deal with specific domain features. Through the explicit mapping of source code and test case mappings, the GPL testing tool responds to the testing commands sent from the re-interpreter component (bottom-left of Figure 1). The test result at the GPL level is



**Fig. 1.** Separation of DSL Perspective and GPL Tools

later sent back to the DSL testing perspective by the testing results mapping component, which is a wrapper interface to convert the GPL testing result messages back into a form to be displayed at the DSL level (i.e., rephrasing the GPL test results into the testing or debugging perspective within Eclipse that is tailored for a specific DSL). As a result, the framework enables a DSL programmer to interact directly with the testing perspectives at the DSL level. More details about our framework, including video demonstrations and complete examples, can be found at the project website [11].

The remainder of this paper is organized as follows: Section 2 introduces the necessary background information to provide the reader with a better understanding of other sections of the paper; Section 3 describes an overview of the issues concerning DSL unit testing; Section 4 presents two case studies that illustrate our ideas; Section 5 shares a few lessons learned and describes several existing limitations that point toward future work; Section 6 discusses related work; Section 7 offers a summary of the paper.

## 2 Background

Our approach exploits a technique to build DSL tools from existing GPL tools available for debugging, unit testing, and profiling (e.g., jdb, JUnit, NetBeans Profiler) with interoperability among the tools provided by plug-in mechanisms offered by most IDEs. The Eclipse plug-in development environment [14] was selected due to its ability to serve as a tool integration platform that offers numerous extension points for customization through an extensible architecture. As an example, Eclipse provides a reusable debugging interface (e.g., buttons for common debugging commands and variable watch lists) and integration with JUnit through extension mechanisms.

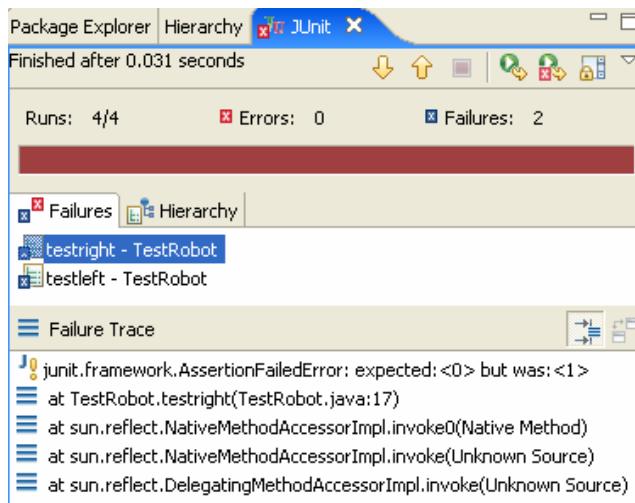
To provide tool support consistent with accepted software engineering practice, a DSL unit test engine should offer developers the ability to discover the existence of

software errors within the DSL program. After identifying the presence of an error through testing, DSL debuggers can further help end-users to locate the errors in the DSL code. This section introduces the necessary background of the basic tools and techniques mentioned throughout the paper, and provides a brief description of the JUnit test platform and DSL Debugging Framework (DDF). This section also introduces two sample DSLs that serve as case studies in later sections of the paper.

## 2.1 Eclipse JUnit Test Platform

A unit test engine is a development tool used to determine the correctness of a set of modules (e.g., classes, methods, or functions) by executing source code against specified test cases. Each unit test case is tested separately in an automated fashion using a test engine. The test results help a programmer identify the errors in their program.

JUnit is a popular unit testing tool for constructing automated Java test cases that are composable, easy to write, and independent [20]. A JUnit plug-in for Eclipse [12] provides a framework for automating functional unit testing [1] on Java programs. JUnit generates a skeleton of unit test code according to the tester's specification. The software developer needs to specify the expected value, the tested variable, the tested module of the source code, and the test method of the test cases. JUnit provides a set of rich testing methods (e.g., **assertEquals**, **assertNotNull**, **assertFalse**, and **assertSame**) and reports the results (shown in Figure 2) as: the total number of passed or failed test cases; the true expected value and current value of the failed test cases; the name and location of the passed and failed test cases; and the total execution time of all the test cases. The test results can be traced back to the source



**Fig. 2.** Screenshot of the JUnit Eclipse Plug-in

code locations of the tested program. The test cases are displayed in a hierarchical tree structure that defines the relationship among test cases. In its current form, JUnit is focused solely on Java and is not applicable to general testing of DSL programs.

Testing frameworks similar to JUnit, such as NUnit [26], also focus at the GPL level and do not provide opportunities for unit testing DSL programs. In Section 3, we describe how our mapping framework enables unit testing of DSL programs using JUnit as the underlying unit test engine.

## 2.2 DSL Debugging Framework (DDF)

DDF represents our prior work [43] that was developed as a set of Eclipse plug-ins providing core support for DSL debugging. In the DDF, a DSL is specified using ANTLR, which can be used to construct recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions. A DSL is usually translated into a GPL that can be compiled and executed [23]. From a DSL grammar, the DDF generates GPL code representing the intention of the DSL program (i.e., the DSL is translated to a GPL and the GPL tools are used to generate an executable program) and the mapping information that integrates with the host GPL debugger (e.g., the stand-alone command line Java debugger – jdb [19]). The generated mapping code and pre-defined debugging method mapping knowledge re-interpret the DSL program and DSL debugging states into a sequence of commands that query the GPL debugger. The debugging result responses from the GPL debugger are mapped back into the DSL debugger perspective. Thus, the end-user performs debugging actions at the level of abstraction specified by the DSL, not at the lower level abstraction provided by the GPL. The DDF was developed using architecture similar to Figure 1, but with mappings to different GPL tools (in the DDF case, a debugger rather than a unit test engine).

## 2.3 Sample DSLs: The Robot and Feature Description Languages

DSLs can be categorized as imperative or declarative [23]. An imperative DSL follows a similar definition used for imperative programming languages, which assumes a control flow that is centered on the importance of state changes of variables. A declarative DSL is a language that declares the relationships among input and output values, with little concern over specific control flow. This paper uses an example from both the imperative and declarative categories to illustrate the concept of unit testing of DSLs. A third category, called embedded DSLs, is discussed in Section 5 as a topic of future work for DSL unit testing.

The top of Figure 5 illustrates a very simple imperative DSL that we have used in various aspects of our work on DSL testing engines. This simple language moves a toy robot in various directions (e.g., up, down, left, right) and provides a mechanism for users to write their own navigation methods (e.g., a **knight** method that moves according to the rules of a knight in chess). An implicit **position** variable keeps track of the position of the robot as the control flow of the program modifies the robot location.

The Feature Description Language (FDL) [37] is a declarative DSL for describing feature models in software product lines. The upper part of Figure 3 (adapted from [37] and previously presented in [43] within the context of DSL debugging) is an example written in FDL to describe car features. The lower part of Figure 3 enumerates all of the possible legal configurations that result from the features defined on the

<b>Car Features in FDL</b>
<b>feature 1:</b> Car: all (carbody, Transmission, Engine, Horsepower, opt(pullsTrailer))
<b>feature 2:</b> Transmission: oneof (automatic, manual)
<b>feature 3:</b> Engine: moreof (electric, gasoline)
<b>feature 4:</b> Horsepower: oneof (lowPower, mediumPower, highPower)
<b>constraint 1:</b> include pullsTrailer
<b>constraint 2:</b> pullsTrailer requires highPower
<b>All Possible Car Configurations</b>
1:(carbody, pullsTrailer, manual, highPower, gasoline, electric) 2:(carbody, pullsTrailer, manual, highPower, electric) 3:(carbody, pullsTrailer, manual, highPower, gasoline) 4:(carbody, pullsTrailer, automatic, highPower, gasoline, electric) 5:(carbody, pullsTrailer, automatic, highPower, electric) 6:(carbody, pullsTrailer, automatic, highPower, gasoline)

**Fig. 3.** Car Features Specified in FDL and List of Possible Car Configurations (adapted from [37])

upper part of the figure. In feature 1 of Figure 3, a **Car** is made of four mandatory parts: **carbody**, **Transmission**, **Engine**, and **Horsepower**. As shown at the end of feature 1, a **Car** has an optional feature called **pullsTrailer**. Features starting with a lowercase letter are primitive features that are atomic and cannot be expanded further (e.g., the **carbody** feature). Features that start with an uppercase character are composite features, which may consist of other composite or primitive features (e.g., the **Transmission** feature consists of two primitive features, **automatic** and **manual**). There are several composition logic operators to help describe more complex situations. In feature 2 of Figure 3, the **oneof** composition logic operator states that **Transmission** can be either **automatic** or **manual**, but not both. In feature 3, the **moreof** composition logic operator specifies that the **Engine** can be either **electric** or **gasoline**, or both. FDL also provides constraint keywords to describe a condition that a legal composition must satisfy. In constraint 1, all cars are required to have a **pullsTrailer**. In constraint 2, only **highPower** cars are associated with the **pullsTrailer** feature. The combination of constraints 1 and 2 imply that all cars in this product line must be **highPower**.

### 3 DSL Unit Testing Framework

As observed from traditional software development, unit testing supports early detection of program errors, and the complementary process of debugging helps to identify

the specific location of the program fault [27] to reduce the cost of software failures. To complement the DDF, the DSL Unit Testing Framework (DUTF) assists in the construction of test cases for DSL programs, much in the sense that JUnit is used in automated unit testing of Java programs. After identifying the existence of an error using DUTF, the DDF (Section 2.2) can then be used to identify the fault location within the DSL program. The DUTF framework invokes the underlying GPL unit test engine (e.g., JUnit) to obtain unit test results that are remapped onto the abstractions representing the DSL. The key mapping activity in DUTF translates the DSL unit test script into GPL unit test cases that are executed at the GPL tool level. In the DUTF, the reports of passed and failed test cases appear at the DSL level instead of the underlying GPL level. A failed test case reported within the DUTF reveals the presence of a potential error in the DSL program.

An illustrative overview of the DUTF is shown in Figure 4. With the mapping generator embedded inside the grammar of the Robot language, the lexer and parser generated by ANTLR (step 1) takes the Robot DSL program as input. ANTLR not only translates the Robot DSL into the corresponding Robot.java representation, but also generates the Mapping.java file (step 2). At the same time, another translator generates the JUnit test case (e.g., TestRobot.java) from the Robot DSL unit test script and another mapping file. The mapping file represents a data structure that records all of the information about which line of a Robot DSL unit test case is mapped to the corresponding JUnit test case in the generated code. A DSL unit test case is interpreted into a JUnit test case against the generated Robot.java code. At the GPL level, the generated JUnit test cases represent the unit testing intention of Robot unit test cases. The mapping component interacts and bridges the differences between the Eclipse DSL unit test perspective and the JUnit test engine (step 3).

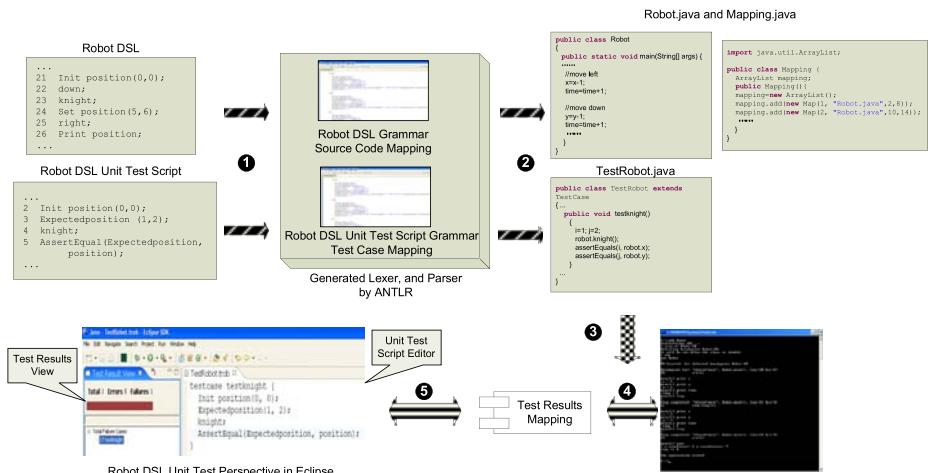


Fig. 4. DSL Unit Testing Framework (DUTF)

There are two round-trip mapping processes involved (step 4 and step 5) between the Robot DSL unit test perspective in Eclipse and JUnit. The results from the mapping components are reinterpreted into the GPL unit test engine as unit test cases that are executed against the translated GPL code. The *source code mapping component* (Section 3.1) uses the generated mapping information to determine which DSL test case is mapped to the corresponding GPL unit test case. The mapping indicates the location of the GPL test case corresponding to a single test case defined in a test script at the DSL level. The *test cases mapping component* (Section 3.2) considers the user's test cases at the DSL level to determine what test cases need to be created and executed by the underlying GPL unit test engine.

The GPL unit test engine (in this example, JUnit) executes the test cases generated from DSL test scripts. Because the messages from the GPL unit test engine are still in the GPL format, the test result at the GPL level is sent back to the Eclipse DSL test result view by the *test results mapping component* (Section 3.3), which is a wrapper interface to remap the test results back into the DSL perspective. The domain experts only see the DSL test result view at the DSL level. The following sub-sections describe the various mappings that are needed to reuse a GPL testing tool at the DSL abstraction level.

### 3.1 Source Code Mapping

Along with the basic functionalities translated from a DSL to its equivalent GPL representation, the syntax-directed translation process also can produce the mapping information augmented with additional semantic actions embedded in the DSL base grammar. ANTLR is used to translate the DSL to a GPL (e.g., Java) and also to generate the mapping hooks that interface with the DUTF infrastructure. The base grammar of the DSL is modified with additional semantic actions that generate the source code mapping needed to create the DSL unit test engine. The mapping consists of the line number of the DSL statement, the translated GPL file name, the line number of the first line of the mapped code segment in the GPL, the line number of the last line of the corresponding code segment in the GPL, the name of the function at the current DSL line number, and the statement type (e.g., **functiondefinition**, **functioncall**, or **none**) at the current DSL line number.

A **functiondefinition** contains **functionhead**, **functionbody**, and **functionend**, where: **functionhead** marks the start of a function (line 3 on the top of Figure 5 is the **functionhead** of **knights**); **functionbody** is the actual definition of a function (lines 4 to 6 on the top of Figure 5 represent the **functionbody** of **knights**); **functionend** indicates the end of a function (line 7 on the top of Figure 5 is the **functionend** of **knights**). A **functioncall** is the marker for a function that is being called from another program location. The statement type for a built-in method or statement of a GPL program is set to **none**. For example, the mapping information at Robot DSL line 13 in the top of Figure 5 is {13, "Robot.java", 20, 21, "main", "none"}. This vector indicates that line 13 of the Robot DSL is translated into lines 20 to 21 in Robot.java, designating the "Set position()" method call inside of the main function. For each line of the Robot DSL code, there is corresponding mapping information defined in the same format. Although the examples presented in this section are tied to Java and the simple Robot DSL, the source code mapping and interaction with the GPL unit

Program Written in Robot DSL	
<pre> ... 3   begin knight: 4     position(+0,+1); 5     position(+0,+1); 6     position(+1,+0); 7   end knight: 8   ... 9   Init position(0,0); 10  left; 11  down; 12  knight; 13  Set position(5,6); 14  up; 15  right; 16  Print position; ... </pre>	<pre> <b>Generated Java Code</b> ... 6  public static void move_knight(){ 7    x=x+0; 8    y=y+1; 9    x=x+0; 10   y=y+1; 11   x=x+1; 12   y=y+0;} 13  public static void main(String[] args) { 14  x=0; 15  y=0; ... 18  move_knight(); ... 20  x = 5; 21  y = 6; ... 26  System.out.println("x coord=" + x + " " + 27  "y coord= " + y); } ... </pre>

**Fig. 5.** Robot DSL Source Code Mapping

test engine and unit test platform can be separated from different DSLs and GPLs. Variable mapping implicitly exists within the DSL compiler specified during the syntax-directed translation in the semantics specification. Figure 6 describes a part of the Robot DSL grammar specification that specifies the semantic actions taken on the implicit **position** variable. This part of the grammar translates line 4 of the Robot DSL in the top of Figure 5 into lines 7 and 8 of the generated Robot.java in the bottom of Figure 5. The Robot DSL variable position is mapped to x and y variables in Robot.java. The translation of the position variable represents a one-to-many variable mapping, where one DSL variable is mapped to two or more GPL variables. These forward (i.e., from DSL to GPL) variable mappings are used implicitly by the DUTF for generating the DSL unit test engines.

```

Functionbody
: (VARS LPAREN opl:OP func_n1 :NUMBER COMMA op2:OP func_n2:NUMBER RPAREN
  { funcall="functionbody";
    dsllinenumber=dsllinenumber+1;
    fileio.print("      x=x"+opl.getText()+func_num1.getText()+";");
    gplbeginline=fileio.getLineNumber();
    fileio.print("      y=y"+op2.getText()+func_num2.getText()+"");
    fileio.print("      time=time+1;");
    gplendline=fileio.getLineNumber();
    filemap.print("mapping.add(new
      Map(\"+dsllinenumber+\", \"Robot.java\""+ gplbeginline+" "+gplendline+" "+\""+
      funcname+"\""+","+"\""+funcall+"\""+"));
  }
);

```

**Fig. 6.** Part of Robot DSL Grammar Specification

The source code mapping shown in Figure 5 is the same mapping that is also used in the DDF for DSL debuggers [43]. In previous work, we showed how the adaptations to the grammar that support DSL tools represent a crosscutting concern. The grammar concerns can be specified using an aspect language that is specific to language grammars [42]; however, this topic is out of scope for the discussion in this paper.

### 3.2 Test Cases Mapping

The abstraction mismatch between DSLs and GPLs also contributes to the mismatch in test cases. When writing a unit test case at the DSL level, one variable in a DSL program may not be equivalent to one variable in the corresponding GPL representation (i.e., a DSL variable may be translated into several variables or objects in the generated GPL). The presentation format of the DSL variable may also differ from the GPL representation. In the case of DUTF, the DSL unit test script is mapped to the corresponding GPL unit test cases by a test case translator written in ANTLR. In the DUTF, the generated GPL test cases are exercised by the underlying GPL unit test engine (e.g., JUnit). The main task of a unit test is to assert an expected variable value against the actual variable value when the program is executed. The variable mapping from a DSL program to the corresponding GPL program is used to construct the mapping of test cases at the GPL source code level. The base grammar of the unit test script is augmented with additional semantic actions that generate the variable mapping and test case line number mapping.

Figure 7 shows the mapping from a Robot DSL unit test case (called **test-knight**) to a corresponding JUnit test case of the same name. In the Robot DSL unit test script, line 2 on the left side is mapped to lines 2 and 3 on the right side; line 3 on the left side is mapped to lines 4 and 5 on the right side. One assertion statement in the Robot DSL unit test script may be translated into multiple separate assertion statements in JUnit due to the mismatch of variables between the DSL and GPL. For example, the variable called **position** in the Robot DSL is translated into two variables (**x** and **y**) in the Java translation; line 5 (left side of Figure 7) is mapped to lines 7 and 8 (right side of Figure 7). This one-to-many test case assertion mapping must be re-mapped back into the DSL view. The Car example, specified in the declarative FDL of Section 2.3, is used as another target DSL case study. Figure 8 shows the mapping from a Car FDL unit test case (called **testFeatures**) to a corresponding JUnit test case of the same name. In the Car FDL unit test script, line 2 on the top of Figure 8

Robot DSL Unit Test Case	JUnit Unit Test Case
<pre> 1 testcase testknight { 2   Init position(0,0); 3   Expectedposition(1,2); 4   knight; 5   AssertEqual(Expectedposition, 6               position); 7 8 }</pre>	<pre> 1 public void testknight() { 2   robot.x = 0; 3   robot.y = 0; 4   int x= 1; 5   int y= 2; 6   robot.move_knight(); 7   assertEquals(x, robot.x); 8   assertEquals(y, robot.y); 9 }</pre>

**Fig. 7.** Test Case Mapping Between Robot Test Case and JUnit Test Case

is mapped to the JUnit test case at the bottom. In this figure, the expected car features are from lines 12 to 14, where three specific features (e.g., **carbody**, **manual**, **highPower**) are desired features. Line 3 invokes a unit of the original Car FDL program that executes all four features defined in the Car FDL program; line 4 invokes a constraint that requires every car feature combination list to include a **pullsTrailer**. The result of lines 3 and 4 is the creation of an explicit variable called **feature**, which keeps track of the list of features that are included in the actual features generated by the given FDL specification.

The parse function used in line 27 of the JUnit test case is a helper function that stores the output of the generated Java program into a unified data structure, and then converts it to the same class type as the current tested car's feature called **testFeatures**. The **compareFeatures** function used in line 27 of the JUnit test case is another helper function that compares the two parameters. The traditional JUnit built-in assertion functions (e.g., **assertEquals**) are not applicable and not capable of handling the particular scenarios in FDL that compare car feature test assertions. This limitation is due to the fact that the order of the car's features written in the FDL test case script is irrelevant. However, the **assertEquals** assertion in JUnit will report an error if two objects are not exactly equal. In other words, the order of the features of the current car and expected car may not be equal. Even if the contents of these two objects are equal the result is still false when compared with **assertEquals** at the JUnit level. However, at the Car FDL abstraction level, they are equal. To address this issue, an external function called **compareFeatures** has been written to handle this situation where only the contents matter and the ordering issue can be ignored.

Line 6 of the Car FDL unit test case is mapped to line 28 of the JUnit test case. It is an assertion to assess the number of possible valid feature combinations. The external **getFeatureListNumber** function retrieves the number of the feature combinations from the parsed data structure. It is not possible to get the size of a feature list because the existing FDL compiler does not provide such a method, so a helper method was needed. The **assertEquals** statement is used to compare the actual feature list size with the expected feature combination number.

In this example, one assertion statement in the Car FDL unit test script is translated into one assertion statement in JUnit. This one-to-one test case assertion mapping is simpler than the one described in the Robot unit test engine case but the comparison

### Car FDL Unit Test Case

```

1 TestCase testFeatures {
2   Expectedfeature:(carbody, manual, highPower);
3   use Car.FDL(All);
4   Constraint C1: include pullsTrailer;
5   AssertTrue(contain(Expectedfeature, feature));
6   AssertEqual(6, numberof feature);
7 }
```

### GPL Unit Test Case (JUnit)

```

11 public void testFeatures () {
12   testFeatures.add("carbody");
13   testFeatures.add("manual");
14   testFeatures.add("highPower");
...
27   assertTrue(compareFeatures(testFeatures,parse(fc,root,cons)));
28   assertEquals(6,getFeatureListNumber(parse(fc,root,cons)));
...
```

**Fig. 8.** FDL Test Cases Mapping

function is more complicated than the Robot example. JUnit does not support the sophisticated assertion functionality that is needed for FDL unit testing. Thus, helper and comparison functions were needed to realize the unit test intention for FDL programs. The provision of such helper functions for the specific use of FDL unit testing represents additions that were needed to customize the DUTF for FDL.

### 3.3 Unit Test Result Mapping

JUnit reports the total number of test cases, total number of failed test cases, and total number of error test cases (i.e., those representing run-time exception errors during test executions). If one test case in a DSL is translated into multiple test cases at the GPL level, the result mapping can become challenging (i.e., rephrasing the results at the JUnit level back into the DSL perspective). One test case's failure result should not affect other test cases. In order to get the final test result of one test case in the DSL, all corresponding GPL test cases have to be tested. The approach adopted in DUTF is to use one-to-one mapping at the test case level (i.e., each test case specified at the DSL level is translated into one test case at the GPL level). Within each test case, one assertion in a DSL test case may be translated into one or many assertions in a GPL test case. The one-to-many mapping that is encapsulated inside the individual test case makes the test result easier to interpret across the abstraction layers.

One failed assertion at the GPL level should result in an entire test case failure. Only those GPL test cases that have passed all assertions should result in a successful test case at the DSL level. Such a simple mapping also helps to determine the location

```

1  protected void handleDoubleClick(DoubleClickEvent dce) {
2      IStructuredSelection selection = (IStructuredSelection) dce.getSelection();
3      Object domain = (TestResultElement) selection.getFirstElement();
4      String casename = ((TestResultElement) domain).getFunctionName();
5      int linenumber = 0;
6      for (int i = 0; i < mapping.size(); i++) {
7          Map map = (Map) mapping.get(i);
8          if (map.getTestcasename().equals(casename)) {
9              linenumber = map.getDslnumber();
10         }
11     }
12     OpenDSLTestEditorAction action = null;
13     action = new OpenDSLTestEditorAction(this, testFileName, linenumber);
14     action.run();
15 }

```

**Fig. 9.** handleDoubleClick function in TestResultView Class

of a failing DSL test case without considering the many-to-one consequence from the line number mapping. The test result from JUnit indicates the location of the failed test case in the JUnit code space, which is not helpful for end-users to locate the position of the specific failed test cases in their DSL unit test script. For simplicity, we keep the test case name the same during the translation process. By matching the test case name through the test case mapping information, we can obtain the corresponding line number of the DSL unit test script from the JUnit test case line number in the test result report. This is illustrated in more detail in the concrete examples presented in Section 4 (e.g., Figures 12 and 13).

The DUTF provides a capability that allows the DSL end-user to double-click on the test cases listed in the **Test Result View**, which will then highlight the specific test case in the editor view. Figure 9 is an example of the plug-in code that was written to interact with JUnit to handle the end-users double-clicking on the failed test case. The method searches through the source code mapping to find the selected test case name (line 8) and then obtains the line number (line 9) of the test case. This information is then used to display the test script editor and highlight the clicked test case in the test script (line 13).

## 4 Example DSL Test Engines

This section illustrates the application of the DUTF on two different types of DSLs (e.g., imperative DSLs and Declarative DSLs). We have implemented unit test engines for various DSLs (e.g., the toy Robot language and the FDL, both described in Section 2.3) using our approach.

### 4.1 Generation of Declarative DSL Test Engine

This sub-section describes the generation of a unit test engine for the imperative Robot DSL. The DUTF adapts the JUnit graphical user interface in Eclipse by adding a new view called the DSL **Test Result View**, which is similar to the underlying

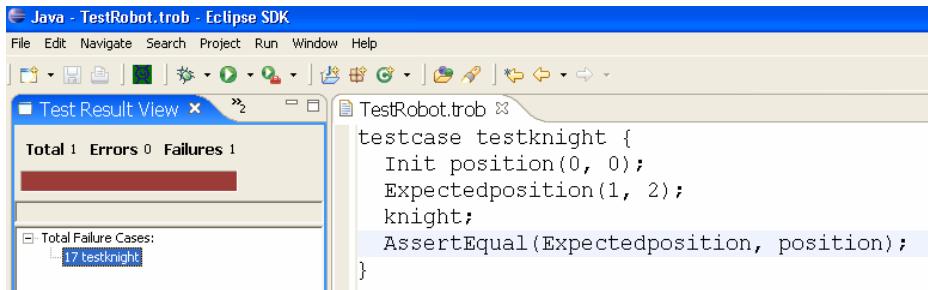


Fig. 10. Screenshot of Unit Testing Session on Robot Language

Correct knight method	Incorrect knight method
<pre> 1 begin knight: 2   position (+0,+1); 3   position (+0,+1); 4   position (+1,+0); 5 end knight: </pre>	<pre> 1 begin knight: 2   position (+0,+1); 3   position (+1,+1); 4   position (+1,+0); 5 end knight: </pre>

Fig. 11. Correct and Incorrect Knight Methods

JUnit **Test Result View**, but mapped to the DSL abstraction. Figure 10 shows a screenshot of a Robot DSL unit test session. A DSL test case is composed of a test case name (e.g., **testknight**) and test body. The test body defines the expected value of a certain variable, the module to be tested, as well as the criteria for asserting a successful pass (e.g., **assertEquals**). The Robot DSL unit test cases are specified in a unit test script, which itself is a DSL. We have implemented the Robot DSL unit test script translator in ANTLR to generate JUnit test cases from the DSL test script. The source code mapping for the Robot DSL unit test script can also be generated by adding additional semantics to the base DSL grammar (in this case, the grammar of the Robot language). The base grammar generates the equivalent Java code for the Robot DSL (see Section 3.1) and the additional semantics generate the Java unit test cases for the Robot DSL unit test script (see Section 3.2).

The right side of Figure 10 is the DSL unit test script editor, which shows an actual Robot DSL unit test script called **TestRobot**. The highlighted test case called **testknight** has an expected value that is set as **position(1, 2)**. The function unit to be tested is the **knight** move and the assertion criteria determine whether there is a distinction between the expected **position** and the actual **position** after **knight** is executed. An incorrect implementation of the knight method is shown in the right side of Figure 11 (i.e., line 3 incorrectly updates the robot to **position (+1, +1)**). When the **testknight** test case is executed on the incorrect knight implementation, the expected **position** value (e.g.,  $<1, 2>$ ) is not equal to the actual **position** (e.g.,  $<2, 2>$ ). In this **testknight** example, the assertion on the **x** coordinate will fail on the incorrect knight implementation, but the assertion

to test **y** will succeed. Consequently, the **testknight** test case is reported as a failure in the **Test Result View** on the left side of Figure 10. The **AssertEqual** assertion in this DSL unit test script tests whether its two parameters are equal. The **Test Result View** also indicates the total number of test cases (in this case 1), the total number of failures (in this case, there was 1 failure), and the number of runtime error test cases (in this case, there were 0 errors causing run-time exceptions). The progress bar that appears in the **Test Result View** indicates there is at least one test case that failed (the bar actually appears red in failure or error cases, and green when all test cases are successful). The list of test cases underneath the progress bar indicates all the names of the test cases that failed to pass the test case (e.g., **testknight**).

## 4.2 Generation of Declarative DSL Test Engine

In addition to generating a unit test engine for an imperative DSL like the Robot language, we also used the DUTF to generate a declarative DSL unit test engine for the FDL. The declarative DSL test engine translates the unit test script into unit test cases in JUnit, which are specified in the script grammar. Because of the domain-specific syntax of the FDL, the DUTF unit test script translator required modification so that it can generate the correct unit test cases for FDL in Java. Also in the declarative DSL case, the variable mapping from the GPL to DSL is different from the imperative DSL case. In the Robot language, a commonly referenced abstraction is the **position** variable and in the FDL an important abstraction is the **feature** variable, which contains the list of possible features up to the current line of execution in the DSL.

A developer or end-user who uses the FDL often tries to narrow down a design space amid a large range of possible configurations. Manually analyzing a large design space can be challenging and even infeasible; an automated testing tool that assists in determining the correct configuration specification can be very helpful to a developer. A screenshot of the unit testing session on a Car FDL program is shown in Figure 12. The right side of Figure 12 is the FDL unit test script editor, which shows an FDL unit test script called **TestCar**. The test case called **testfeatures** has an expected value called **Expectedfeature** that is set as **(carbody, manual, highPower, electric, pullsTrailer)**. The expected value has the legitimate car feature configuration according to the Car FDL program. The target unit to

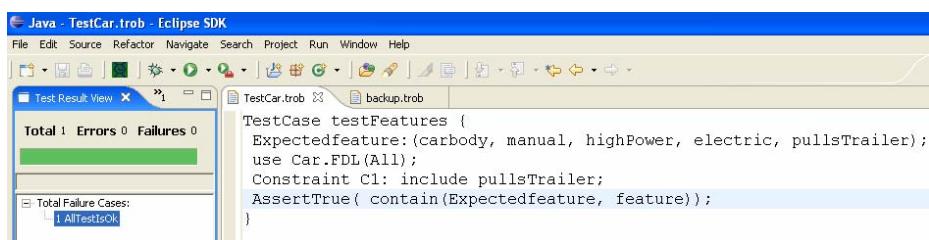


Fig. 12. Screenshot of Unit Testing Session on Car FDL

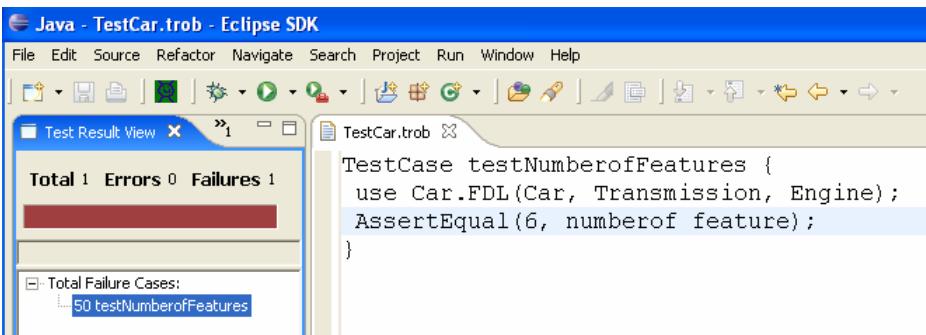


Fig. 13. Screenshot of Unit Testing Session on Car FDL

be tested is all the features (from feature 1 to feature 4 in Figure 3) plus one constraint (constraint 1 in Figure 3). We introduce another assertion called **AssertTrue** to assess whether the tested unit will return true or false. If it returns true, the **AssertTrue** assertion will pass, otherwise it will fail. An assertion is set to test whether the **Expectedfeature** is contained in the set of possible features after executing all the features and one constraint. In the left side of Figure 12, the **Test Result View** indicates this assertion succeeds, so **Expectedfeature** is one of the possible features.

Figure 13 is another test case called **testNumberOfFeatures**, which is highlighted in the **Test Result View**. The expected number of possible features is 6. The targeted testing unit consists of three features (from feature 1 to feature 3 in Figure 3). The **numberof** operator returns the size of a set. An **AssertEqual** assertion tests whether the number of possible features after executing all these three features is 6. In the left side of Figure 13, the test result view indicates this assertion fails. The only features executed are **Car**, **Transmission**, and **Engine**. There are two options for **Transmission** (**automatic** and **manual**), three options for **Engine** (**electric**, **gasoline**, and **electric/gasoline**), and two options for **pullsTrailer** (with or without). The total number of features is actually 12 ( $2*3*2$ ) rather than the expected 6, which causes the test case to fail as indicated in the DSL unit **Test Result View** of Figure 13.

## 5 Lessons Learned, Limitations, and Future Work

One of the most challenging tasks in creating the DUTF was identifying the differences among unit testing between the GPL and DSL levels for each particular mapping. The main task of a unit test is to assert an expected variable value against the actual variable value when the program is executed. The abstraction mismatch between DSLs and GPLs also contributes to the mismatch in test cases. When writing a unit test case at the DSL level, one variable in a DSL program may not be equivalent to one variable in a GPL (i.e., a DSL variable may be translated into several variables or objects in the generated GPL). Moreover, the concept of comparing raw values

may not be relevant at the DSL level. For example, in the case of the FDL, a software developer may be interested in testing the number of all possible configurations, or testing if a particular configuration is present in the specified program. In the latter case, the order of particular features in a configuration is irrelevant. Hence, a unit test engine should not report a failure if only the order of features is different (i.e., the equality assertion at the FDL level is about set equality of feature lists).

By using the DUTF, the development effort to build DSL unit testing tools was significantly reduced compared to developing a testing tool for each DSL from scratch. DUTF consists of 22 reusable software components implemented in over 3k lines of code. The amount of code that was written for each new unit test engine can be used to quantify the level of effort required to adapt a unit test engine. Based on the two example DSL unit test engines described in this paper for the Robot and FDL languages, on average, less than 360 additional lines of code were needed for each new DSL unit test engine (e.g., the Robot DSL required two additional classes representing 239 lines of code, and the FDL DSL required four additional classes representing 482 lines of code). Most of the customized code handles the different scripting languages that were needed for each type of DSL (e.g., imperative or declarative). Of course, more complex and feature-rich DSLs will likely require additional customization, but the infrastructure of DUTF provides a basis for general unit testing concepts that can be reused by different DSL tooling efforts.

There are opportunities for applying the ideas of this paper to other contexts and testing tools:

- Software developers may also be interested in the performance of their DSL applications during the execution of their program (e.g., CPU performance profiling, memory profiling, thread profiling). A DSL profiler would be helpful to determine performance bottlenecks and hotspots during execution. The same approach as depicted in Figure 1 can be applied to generate a DSL profiler (i.e., the framework’s GPL unit test engine can be replaced by a GPL profiler, which monitors the run-time characteristics of the execution environment). The framework can use the NetBeans Profiler as the underlying GPL profile server in the case when a DSL is generated to Java. The NetBeans Profiler provides basic profiling functionalities including CPU, memory and threads profiling, as well as basic JVM monitoring. The GPL profile engine could execute the profiling commands generated from the re-interpreter inside the framework. The domain experts will only see the DSL profiling result view and interact at the DSL level. We are currently in the process of completing the DSL profiler architecture. An alternative to the NetBeans Profiler is JFluid [10], which could also be integrated into the general testing tool framework.
- Even though our framework implementation is integrated within Eclipse using Java-based tools (e.g., jdb and JUnit), we believe the concepts of the framework can be generalized and applied to other IDEs (e.g., Microsoft Visual Studio [33]) such that other testing tools (e.g., Cordbg or NUnit) can be leveraged for reuse when the underlying generated GPL (e.g., C++ or C#) changes.

There remain several open issues that need to be investigated further, such as:

- Embedded DSLs are becoming more popular and represent the case when DSL statements are embedded inline within the code of a GPL. From our previous experience with generating embedded DSL debuggers [43], we believe the same approach described in this paper can apply to unit test engines for embedded DSLs. DSL segments mapping to a GPL represent the same situation discussed in this paper. GPL segments surrounding embedded DSL statements are already in the GPL format and do not require further mapping. The interesting question for unit testing embedded DSLs concerns the interaction between the DSL and GPL boundaries (e.g., when data defined by a DSL is processed by GPL code).
- The two DSLs introduced in this chapter were rather simple to implement in terms of the source-to-source translation from the DSL program to GPL code. These two examples represented a direct translation where text in the DSL had a linear correspondence to the generated GPL. However, some source-to-source translations of a DSL may have a single DSL construct spread across multiple places in the generated GPL code. The challenge is that a more complex mapping is needed to track the line numbers between the two abstractions and how the test results map back into the DSL perspective.

## 6 Related Work in DSL Tool Support

The End-Users Shaping Effective Software (EUSES) Consortium [13] represents collaboration among several dozen researchers who aim to improve the software development capabilities provided to end-users. A contribution of EUSES is an investigation into the idea of “What You See Is What You Test” (WYSIWT) to help isolate faults in spreadsheets created by end-users [7]. More specific to the focus of our research, this section provides an overview of related work in the areas of language definition framework tools (e.g., ASF+SDF, JTS, LISA, and SmartTools). The following related work represents frameworks that can generate many useful language tools, however, none of them addresses unit testing for DSLs.

ASF+SDF is the meta-language of the ASF+SDF Meta-Environment [35], which is an interactive language environment to define and implement DSLs, generate program analysis and transformation tools, and produce software renovation tools. ASF+SDF is a modular specification formalism based on the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF). ASF+SDF has produced many language tools including a debugger.

The Jakarta Tool Suite (JTS) [3] is a set of tools for extending a programming language with domain-specific constructs. JTS consists of Jak (a meta-programming language that extends a Java superset) and Bali (a tool to compose grammars). The focus of JTS is DSL construction using language extensions that realize a product line of DSLs.

The Language Implementation System based on Attribute grammars (LISA) [24], [25] is a grammar-based system to generate a compiler, interpreter, and other language-based tools (e.g., finite state automata, visualization editor). To specify the semantic

definition of a language, LISA uses an attribute grammar, which is a generalization of context-free grammars where each symbol has an associated set of attributes that carry semantic information. With each grammar production, a set of semantic rules is associated with an attribute computation. LISA provides an opportunity to perform incremental language development of an IDE such that users can specify, generate, compile-on-the-fly, and execute programs in a newly specified language [17].

Another language extension environment for Java is the Java Language Extender (JLE) framework [39], which is also based on attribute grammars written in a specification language called Silver. The JLE permits extensions to a host language (e.g., Java) to incorporate domain-specific extensions.

SmartTools is a language environment generator based on Java and XML [2]. Internally, SmartTools uses the AST definition of a language to perform transformation. The principal goal of SmartTools is to produce open and adaptable applications more quickly than existing classical development methods. SmartTools can generate a structured editor, UML model, pretty-printer, and parser specification.

These language definition framework tools help domain experts to develop their own programming languages and also generate useful language tools for the new languages (e.g., language-sensitive editor and compiler). Other researchers are more interested in testing methods and the efficient way to generate the unit test cases such as parameterized unit testing [34], testing grammar-driven functionality [22], generating unit tests using symbolic execution [44], generating test inputs of AspectJ programs [45]. The idea of applying formal methods to determine the correctness of DSL programs was discussed in [5]. However, there does not appear to be any literature or relevant discussion related to unit testing of DSL programs.

## 7 Conclusion

As the cost of software failures rise substantially each year and the number of end-user programmers involved in the software development process increases, there is an urgent need for a full suite of development tools appropriate for the end-user's domain. Software failures pose an increasing economic risk [15] as end-user programmers become more deeply involved in software development without the proper unit test capabilities for their DSL applications. The utility of a new DSL is seriously diminished if supporting tools needed by a software developer are not available.

To initiate discussion of testing DSL programs, this paper introduced DUTF, which is a novel framework for generating unit test engines for DSLs. The technique is centered on a mapping process that associates DSL line numbers with their corresponding representation in the generated GPL line numbers. The Eclipse plug-in architecture and the JUnit test engine provide the infrastructure and unit testing support needed to layer the concept of DSL unit testing on top of pre-existing GPL tools. Although the contribution described in this paper is only a first effort with an associated prototype [11], we believe that the issue of testing DSL programs and the necessary tool support will become increasingly important as a push is made to adopt DSLs in general practice.

## Acknowledgements

This work was supported in part by an NSF CAREER grant (CCF-0643725) and an IBM Eclipse Innovation Grant (EIG).

## References

- [1] ANTLR, ANOther Tool for Language Recognition (2008),  
<http://www.antlr.org/>
- [2] Attali, I., Courbis, C., Degenne, P., Fau, A., Fillon, J., Parigot, D., Pasquier, C., Coen, C.S.: SmartTools: a Development Environment Generator based on XML Technologies. In: ICSE Workshop on XML Technologies and Software Engineering, Toronto, Canada (2001)
- [3] Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Fifth International Conference on Software Reuse, Victoria, Canada, pp. 143–153 (1998)
- [4] Bentley, J.: Little Languages. *Communications of the ACM* 29(8), 711–721 (1986)
- [5] Bodeveix, J.P., Filali, M., Lawall, J., Muller, G.: Formal methods meet Domain Specific Languages. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 187–206. Springer, Heidelberg (2005)
- [6] Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In: Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, pp. 365–383 (2004)
- [7] Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., Wallace, C.: End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In: International Conference on Software Engineering, Portland, OR, pp. 93–105 (2003)
- [8] CUP User Manual (2007),  
<http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>
- [9] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, Reading (2000)
- [10] Dmitriev, M.: Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation. Sun Microsystems Technical Report. Mountain View, CA (2004), <http://research.sun.com>
- [11] Domain-Specific Language Testing Studio (2009),  
<http://www.cis.uab.edu/softcom/DDF>
- [12] Eclipse (2008), <http://www.eclipse.org>
- [13] End-Users Shaping Effective Software Consortium (2007),  
<http://eusesesconsortium.org>
- [14] Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, Reading (2003)
- [15] Gelperin, D., Hetzel, B.: The Growth of Software Testing. *Communications of the ACM* 31(6), 687–695 (1988)
- [16] Harrison, W.: The Dangers of End-User Programming. *IEEE Software* 21(4), 5–7 (2005)
- [17] Henriques, P., Pereira, M., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic Generation of Language-based Tools using LISA. *IEE Proceedings – Software* 152(2), 54–69 (2005)
- [18] Hilzenrath, D.: Finding Errors a Plus, Fannie says: Mortgage Giant Tries to Soften Effect of \$1 Billion in Mistakes, *The Washington Post* (2003)

- [19] JDB, The Java Debugger (2008),  
<http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html>
- [20] JUnit (2006), <http://www.junit.org>
- [21] Kieburtz, B.R., Mckinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I., Walton, L.: A Software Engineering Experiment in Software Component Generation. In: International Conference on Software Engineering, Berlin, Germany, pp. 542–552 (1996)
- [22] Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: IFIP International Conference on Testing Communicating Systems, New York, NY, pp. 19–38 (2006)
- [23] Mernik, M., Heering, J., Sloane, A.: When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4), 316–344 (2005)
- [24] Mernik, M., Lenič, M., Avdićašević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 1–4. Springer, Heidelberg (2002)
- [25] Mernik, M., Žumer, V.: Incremental Programming Language Development. *Computer Languages, Systems and Structures* 31, 1–16 (2005)
- [26] NUnit Project Page (2008), <http://www.nunit.org/>
- [27] Olan, M.: Unit Testing: Test Early, Test Often. *Journal of Computing Sciences in Colleges* 19(2), 319–328 (2003)
- [28] Rebernak, D., Mernik, M., Wu, H., Gray, J.: Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars. In: IET Software (Special Issue on Domain-Specific Aspect Languages) (2009) (in press)
- [29] Scaffidi, C., Shaw, M., Myers, B.: Estimating the Numbers of End Users and End User Programmers. In: Symposium on Visual Languages and Human-Centric Computing, Dallas, TX, pp. 207–214 (2005)
- [30] Schmitt, R.B.: New FBI Software May Be Unusable. *Los Angeles Times* (2005)
- [31] Sebesta, R.W.: Concepts of Programming Languages. Addison-Wesley, Reading (2003)
- [32] Tassey, G.: The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3 (2002), <http://www.nist.gov/director/prog-ofc/report02-3.pdf>
- [33] Thai, T.L., Lam, H.: Net Framework Essentials. O'Reilly, Sebastopol (2002)
- [34] Tillmann, N., Schulte, W.: Parameterized Unit Tests with Unit Meister. In: European Software Engineering Conference (ESEC)/Symposium on the Foundations of Software Engineering, Lisbon, Portugal, pp. 241–244 (2005)
- [35] Van Den Brand, M., Heering, J., Klint, P., Oliver, P.: Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems* 24(4), 334–368 (2002)
- [36] Van Deursen, A., Klint, P.: Little Languages: Little Maintenance? *Journal of Software Maintenance* 10(2), 75–92 (1998)
- [37] Van Deursen, A., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology* 10(1), 1–17 (2002)
- [38] Van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35(6), 26–36 (2000)
- [39] Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute Grammar-based Language Extensions for Java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
- [40] Wile, D.S.: Lessons Learned from Real DSL Experiments. *Science of Computer Programming* 51(3), 265–290 (2004)

- [41] Wile, D.S., Ramming, J.C.: Guest Editorial: Introduction to the Special Section “Domain-Specific Languages (DSLs)”. *IEEE Transactions on Software Engineering* 25(3), 289–290 (1999)
- [42] Wu, H., Gray, J., Roychoudhury, S., Mernik, M.: Weaving a Debugging Aspect into Domain-Specific Language Grammars. In: *Symposium for Applied Computing (SAC) – Programming for Separation of Concerns Track*, Santa Fe, NM, pp. 1370–1374 (2005)
- [43] Wu, H., Gray, J., Mernik, M.: Grammar-Driven Generation of Domain-Specific Language Debuggers. *Software: Practice and Experience* 38(10), 1475–1497 (2008)
- [44] Xie, T., Marinov, D., Schulte, W., Noktin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
- [45] Xie, T., Zhao, J.: A Framework and Tool Support for Generating Test Inputs of AspectJ Programs. In: *International Conference on Aspect-Oriented Software Development*, Bonn, Germany, pp. 190–201 (2006)
- [46] Zhu, H., Hall, P.A.V., May, J.H.R.: Software Unit Test Coverage and Adequacy. *ACM Computing Surveys* 29(4), 366–427 (1997)

# Combining DSLs and Ontologies Using Metamodel Integration

Tobias Walter<sup>1,2</sup> and Jürgen Ebert<sup>1</sup>

<sup>1</sup> Institute for Software Technology, University of Koblenz-Landau  
Universitätsstrasse 1, Koblenz 56070, Germany  
`{ebert,walter}@uni-koblenz.de`

<sup>2</sup> ISWeb — Information Systems and Semantic Web,  
Institute for Computer Science, University of Koblenz-Landau  
Universitätsstrasse 1, Koblenz 56070, Germany

**Abstract.** This paper reports on a case study where the domain specific language BEDSL for the description of network devices for computer networks is combined with the feature description language FODA used for defining the variability structure of product lines. Furthermore, annotations by fragments of the web ontology language OWL can be added.

In essence, the approach is a three-way integration, which regards two documents written in BEDSL and FODA, respectively, and semantic OWL-annotations as three equally important views of the system under discussion. The integration of languages is done on the level of their metamodels. The standard metamodel of OWL 2 is merged with two self-developed metamodels for the respective domain specific languages.

The merge is loss-free, i.e. the resulting merged model still contains all information from its parts. Thus, the BEDSL part can be used to visualize the network model, the FODA part still defines the feature structure of the corresponding product line and the OWL part can be extracted and fed into an OWL tool to assert the semantic conditions.

## 1 Introduction

*Domain specific languages* (DSLs) are used to model and develop systems of particular application domains. Such languages are high-level and provide abstractions and notations for better understanding and easier modeling using concepts and notations that are familiar to domain experts. Often a variety of different domain specific languages is used simultaneously to describe the system from several viewpoints.

In the context of *Model Driven Software Engineering* (MDSE) DSL models may be combined with other models in standardized languages to form a complete and consistent overall model of the system under development. Thus, they have to be viewed as one integrated overall model and simultaneously the view-specific (sub-)models shall not be destroyed.

For reasoning about all kinds of models description logics based *ontology languages* get more and more accepted as a means for describing concept structures

and constraints. Ontology languages like OWL allow a precise description of the underlying concepts and provide a good basis for reasoning over models in the described application domain.

This paper reports on a case study where a domain specific language for the description of network devices (BEDSL, [1]) for computer networks is combined with a feature description language used for defining the variability structure of product lines (FODA, [2]). Those two languages are integrated into one common description and extended by fragments of the web ontology language OWL 2 [3] to annotate semantics and semantical constraints.

This approach is a three-way integration, which regards two documents written in BEDSL and FODA, respectively, and the semantic OWL-annotations as three equally important views of the system under discussion. The integration of languages is done on the level of their metamodels. The standard metamodel of OWL 2 [3] is merged with two self-developed metamodels for the respective domain specific languages.

The merge is loss-free, i.e. the resulting combined model still contains all information from its parts. Thus, the BEDSL part can be used to visualize the network configuration, the FODA part still defines the feature structure of the corresponding product line and the OWL part can be extracted and fed into an OWL tool to assert the semantic conditions.

The structure of this paper is as follows. Section 2 shortly introduces the case study and gives an example of an integrated BEDSL-FODA-OWL description as it might be produced by the domain engineers. Section 3 discusses the three languages separately, gives a short sketch of their use and presents some relevant parts of their respective metamodels. In section 3 the three single metamodels are integrated into one (using some elementary transformation steps) in such a way that on the one hand no information is lost and on the other hand all inter-model relations are respected and expressed explicitly. Section 5 demonstrates the integration using the overall example and shows e.g. the OWL-part of the model can be extracted in its full form from the integrated model. Section 6 discusses related work, and section 7 summarizes the paper and gives an outlook.

## 2 Case Study

In this section we will present a case study which is provided by *Comarch*<sup>1</sup>, one of the industrial partner in the *MOST project*<sup>2</sup>.

Comarch, a Polish software house, is specialized in designing, implementing and integrating IT solutions and services. Specifically, Comarch provides an *Operations Support Systems* (OSS) [4] that generally refers to systems that perform management, inventory, engineering, planning, and repair functions for communications service providers and their networks.

For software development Comarch uses model-driven methods, where different kinds of domain specific languages (DSL) are deployed during the modeling process.

---

<sup>1</sup> <http://www.comarch.com/>

<sup>2</sup> <http://www.most-project.eu/>

In this case study we want to consider two domain specific languages used by Comarch. On the one hand for representing the business entities from its problem domain, namely network devices, Comarch uses its own self-developed language called BEDSL (Business Entity Domain Specific Language). The language is very simple such that modeling is easy and productivity and quality of the models are obtained. On the other hand the feature description language FODA is considered to define the variability of their product lines. Since Comarch complains that these domain specific languages are not expressive enough, the web ontology language OWL comes into play to define additional semantic conditions for both languages, BEDSL and FODA.

To fulfil this need the idea is to integrate ontologies within the two domain specific languages in the following way. A domain modeler should use BEDSL and FODA as much as he can since he is familiar with these languages. If he realizes that the DSL he is using is not expressive enough or if he wants to define additional semantic conditions he should be able to annotate the model elements with very simple *OWL text*. We use the term *OWL text* because the domain modeler should not annotate the model elements with complete ontologies but only with relevant parts. On the one side he might not be familiar with the Web Ontology Language (OWL) and developing ontologies, on the other side redundant information that is already defined by the static structure of the domain models should not be stated again in the ontology to avoid redundancy.

Figure 1 depicts two domain models. On the left side BEDSL is used to define the structure of network devices. Here each network device has some ports and each port has a state which either can be free or reserved. On the right side FODA is used to define features of a product line. In the example the diagram states that if the feature *Network Management Operations* is selected exactly one of the subfeatures *Show Ports* and *Allocate Ports* has to be selected, too.

Besides the static structure of the two domain models the Comarch designer wants to define that every *Networkdevice* entity is available for the feature *Show Ports* but not for the feature *Allocate Port*, because only free ports can be allocated. But a network device which has some ports whose state is *FreePort* is available for the *Allocate Port* feature. Furthermore the domain modeler wants to prescribe that each network device has exactly 8 ports and a port has exactly 1 state.

Because the domain specific language is not expressive, much less than for example UML class diagrams (roles, cardinalities etc. are missing), some model elements are annotated with OWL text and global conditions are stated, using the Manchester Syntax style [5] in this case. In the example we first have the constraint **hasPort exactly 8 Port (1)**. Thus the designer states that *Networkdevice* using the **hasPort**-association is exactly connected with 8 Ports. Furthermore he defines that each port has exactly one state using the following statements: **hasState exactly 1 State (2)**.

The second part of constraints contains the following statements: **AvailableFor value ShowPorts (3)**. Here the designer states that the entity *Networkdevice* is available for the *Show Ports* feature.

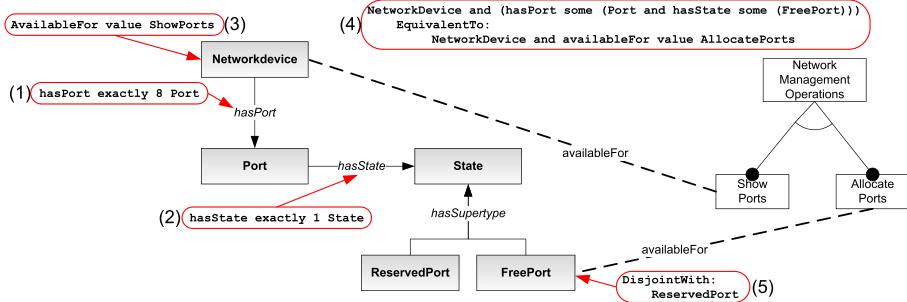


Fig. 1. Domain model with annotated model elements

At last we want to define that a network device which has some port with state **FreePort** is available for the *Allocate Port* feature. Therefore the domain modeler can define the following global constraint: **NetworkDevice and (hasPort some (Port and hasState some (FreePort))) EquivalentTo: NetworkDevice and availableFor value AllocatePorts** (4). Using Manchester syntax style for all annotations he states that an entity of network device is available for the *Allocate Ports* feature, if and only if the network device has a port which has the state **FreePort**.

Using the statements **DisjointWith: ReservedPort** (5) the domain modeler defines that the semantic extension of **ReservedPort** and **FreePort** is disjoint.

Having annotated domain models the domain modeler should be able to check the constraints defined by the OWL text. Here the idea is to project the domain models and its constraints to a complete ontology for querying and reasoning.

### 3 Domain Specific Modeling and Ontologies

The approach of integrating domain specific languages and ontology languages presented here occurs at the metamodel layer and thus bases on different metamodels for DSLs and ontologies.

This work is based on the *four-layer metamodel hierarchy* as a basic requirement for formally defining domain specific languages. Having the idea of metamodeling we present a simple DSL that is used by our partner Comarch to describe their business entities. The idea of this paper is to integrate ontologies into languages itself. Instead of a precise specification of the metamodels of the web ontology language, which is already done by the OMG [6] or by W3C [3], we want to give the idea of how developing ontologies in a model-driven way.

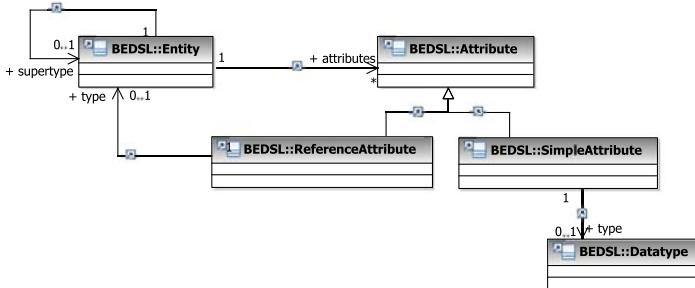
#### 3.1 Metamodeling

Like other modeling languages DSLs must be defined formally to be supported by tools. Otherwise it would not be possible to generate code from them [7]. Here *metamodeling* is used as well known way to describe languages [8].

An important framework for defining languages is the OMG four-layer modeling architecture. In such a modeling architecture the *M0-layer* represents the real world objects. Models are defined at the *M1-layer*, a simplification and abstraction of the M0-layer. Models at the M1-layer are defined using concepts which are described by metamodels at the *M2-layer*. Each metamodel at the M2-layer determines how expressive its models can be. Analogously metamodels are defined by using concepts described as meta-metamodels at the *M3-layer*. In the OMG model driven architecture approach we have MOF at the M3-layer which is defined by itself [9]. So in general each model is defined by its corresponding metamodel as an instantiation of classes or concepts belonging to the corresponding metamodel.

### 3.2 Metamodel of the Business Entity Domain Specific Language

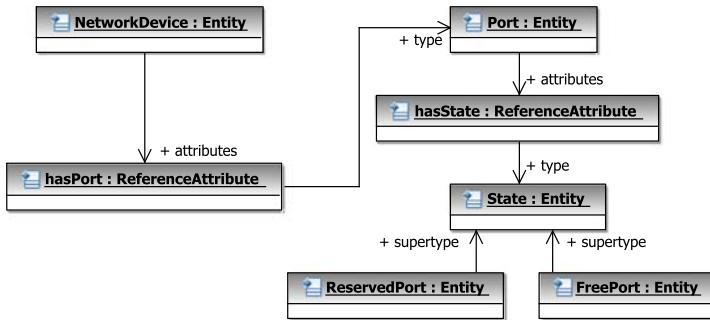
As mentioned above Comarch uses a domain specific language to describe business entities and the relation between them. Figure 1 depicts on the left side an example of using the domain specific language BEDSL in concrete syntax. Here the domain model defines the relation between the entities **Networkdevice**, **Port**, **State**, **FreePort** and **ReservedPort**.



**Fig. 2.** Metamodel of the Business Entity Domain Specific Language (BEDSL)

Figure 2 depicts the metamodel of the *Business Entity Domain Specific Language* (BEDSL). To describe concepts we first have two classes **Entity** and **Attribute**. Each **Entity** can be specialized and can be assigned with a number of attributes. The metamodel provides two types of attributes for entities: **SimpleAttribute** and **ReferenceAttribute**. A **SimpleAttribute** is used for data values which have to correspond to a data type. Via a **ReferenceAttribute** an entity can point to some other entity.

Figure 3 depicts an instance of the metamodel of the domain specific language at the M1-layer which describes the example from figure 1 in abstract syntax. Here a **NetworkDevice** of type **Entity** has a **ReferenceAttribute** a **Port** of type **Entity**. The **Port** also has a **PortState** which has two specializations **ReservedPort** and **FreePort**.



**Fig. 3.** Example of using the Domain Specific Language

### 3.3 Metamodel of the Web Ontology Language

In general ontologies are used to define sets of concepts that are used to describe domain knowledge and allow specifying classes by rich, precise logical definitions [10]. Advantages of ontologies, e.g. represented by languages such as OWL or RDFS, are the support of other (domain specific) languages by enabling validation or automated consistency checking. Furthermore ontology languages provide a better support for reasoning than MOF-based languages [11].

Recent works exist that compare the *ontological technology space* (OTS) and the *metamodeling technology space* (MMTS) based on an UML environment [12] [13]. A short summary of the comparison between UML modeling and ontological modeling is provided by figure 4.

To get a feeling for the OWL 2 metamodel and for some requirements for this paper we present two constructs of the metamodel.

The first part of the metamodel depicted in figure 5 presents the *Object Property Axioms* `ObjectPropertyDomain` and `ObjectPropertyRange`. The `ObjectPropertyDomain` axiom is used to connect one OWL class, described by `ClassExpression`, with the domain of one OWL object

OTS	MMTS
ontology	package
class	class, classifier
individual, value	instance
property	association, attribute
subclass, subproperty	subclass, generalization
data types	data types
enumeration	enumeration
domain, range	navigable, non-navigable
cardinality	multiplicity

**Fig. 4.** Comparison of ontology technology space and metamodeling technology space [13]

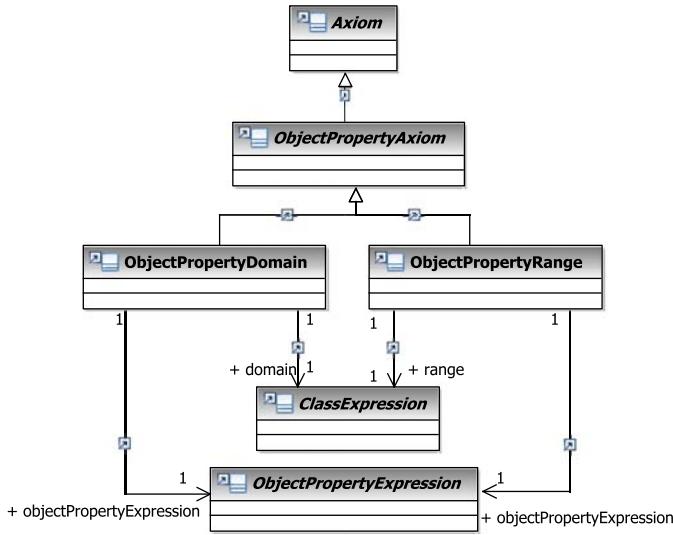


Fig. 5. Object Property Axioms

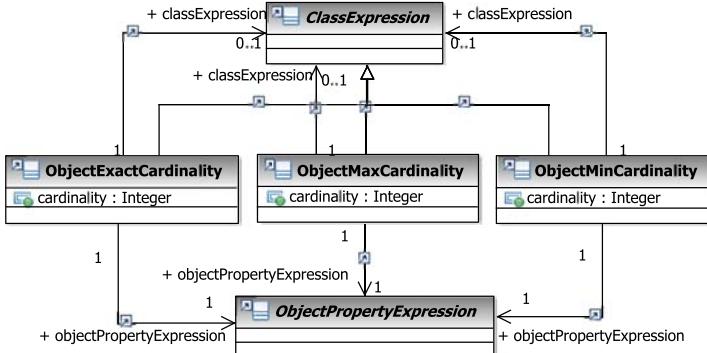


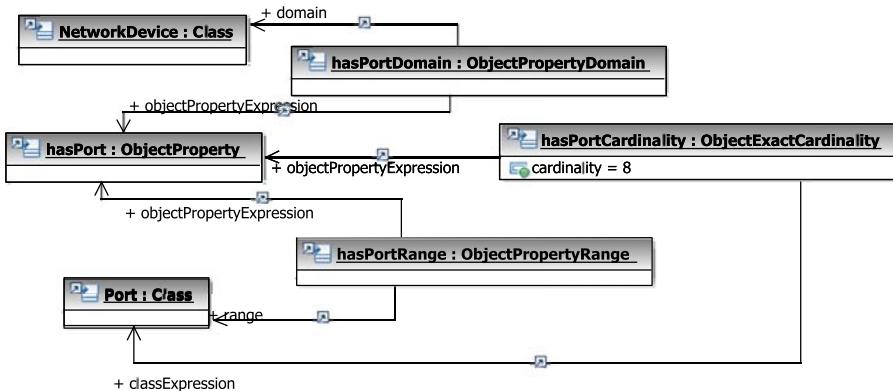
Fig. 6. Object Property Cardinality Restrictions

property, described by the `ObjectPropertyExpression`. Vice versa, the `ObjectPropertyRange` axiom is used to connect one OWL class, described by `ClassExpression`, with the range of one OWL object property, described by the `ObjectPropertyExpression`.

Classes in OWL 2 can be formed by placing restrictions on the cardinality of object property expressions. Figure 6 depicts the *object property cardinality restrictions* provided by OWL 2. The specialization of class expressions `ObjectMinCardinality`, `ObjectMaxCardinality`, and `ObjectExactCardinality` contain those individuals that are connected by an object property expression to at least, at most, or exactly a given number of instances of a specified class expression [3].

A complete and detailed structural specification and the functional-style syntax of the OWL 2 Web Ontology Language is presented in [3].

Figure 7 depicts an example of using the metamodel of OWL 2. Here two instances of `Class` are used to define the OWL classes `Networkdevice` and `Port`. An instance of `ObjectProperty` called `hasPort` in addition with the instances of `ObjectPropertyDomain` and `ObjectPropertyRange` is used to connect the two classes. At last the `ObjectExactCardinality` restriction is adopted on the object property. Figure 8 depicts the ontology in concrete syntax. In fact the ontology defines that a network device has exactly 8 ports.



**Fig. 7.** Example of using the OWL 2 metamodel

---

```

Class: owl:Thing

Class: NetworkDevice
  SubClassOf:
    owl:Thing

Class: Port
  SubClassOf:
    owl:Thing

ObjectProperty: hasPort
  Domain:
    NetworkDevice
  Range:
    hasPort exactly 8 Port
  
```

---

**Fig. 8.** Ontology in concrete syntax

### 3.4 Metamodel of a Feature Oriented Domain Analysis

In the first part of this section we will give a short language specification of another domain specific language, called FODA (*Feature Oriented Domain Analysis*) [2]. In the following we depict the metamodel of the language.

Type	Notation
<i>Mandatory</i> - If feature $C$ is selected, feature $F$ must be selected too	
<i>Optional</i> - If feature $C$ is selected, feature $F$ may or may not be selected	
<i>Alternative</i> - If feature $C$ is selected, exactly one of the child features $F1, F2$ has to be selected	
<i>Or</i> - If feature $C$ is selected, one or more of the child features $F1, F2$ can be selected	
<i>Selection</i> - If feature $C$ is selected, any number between $i$ and $j$ of its child features must be selected too	

**Fig. 9.** Types of feature relationships in a feature diagram

FODA appeals to many product line developers because features are essential abstractions that both customers and developers understand [14].

Thus the main concept in the feature description language FODA is the *feature* itself. Here a feature is a distinguishable characteristic of a concept that is relevant to some stakeholders [15]. To describe features and relationships between them a feature model is used which is represented by a *feature tree*. The nodes of this tree depict the features themselves. Edges of different types are used to specify relationships between these features.

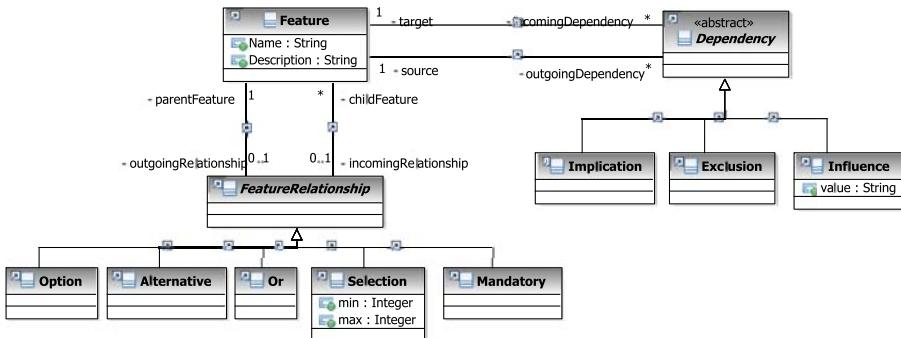
Figure 9 provides an overview of all types of *feature relationships* which connect two different features. Every feature relationship has a unique type and assigns a child feature to its parent feature. For graphical notation we use the one from [2].

Beside feature relationships some features also can depend on some other:

- *Implication* - An implication between two features  $F1$  and  $F2$  defines that if feature  $F1$  is selected  $F2$  also must be selected.
- *Exclusion* - An exclusion between two features  $F1$  and  $F2$  defines that if feature  $F1$  is selected the feature  $F2$  must not be selected. Vice versa, if feature  $F2$  is selected feature  $F1$  must not be selected.
- *Influence* - An influence between two features  $F1$  and  $F2$  defines that if feature  $F1$  is selected then one property of  $F2$  is constrained by one value if  $F2$  is selected too.

Figure 10 depicts a simplified metamodel of FODA with regard to the language description of FODA given here.

Features, represented by the class **Feature** in the metamodel, are on the one hand connected to other features via **FeatureRelationship**-associations, and on the other hand they may be connected via **Dependency**-associations. For every type of feature relationship, there exists a subclass of **FeatureRelationship** in the metamodel. For every kind of dependencies between two features, there also exists one subclass of **Dependency**.



**Fig. 10.** Metamodel of FODA

## 4 Integration of Domain Specific Languages and OWL

In this section we describe the integration of the two domain specific languages BEDSL and FODA with the ontology language OWL 2.

The general idea is to have an integrated modeling language that allows us to define domain models and constraints for model elements simultaneously. For instance we want to apply OWL class axioms on entities of the domain model. Furthermore different constraints might be adopted on the relation between entities, features and attributes.

After integrating the three languages BEDSL, OWL and FODA at metamodel-level (M2) we are able to create models that consist on the one side

of domain models but on the other side also of OWL constructs to constrain the domain model. In concrete syntax (cf. figure 1) we would be able to create our domain models and annotate model elements with OWL text in Manchester Syntax style. In our case OWL text means that only necessary parts of the ontology are defined, because redundant information that is already defined by the domain model should not be stated again in OWL to avoid redundancy.

We describe the integration of the domain specific language BEDSL and the ontology language OWL 2 using eight steps. In each step one transformation is presented which describes where the two metamodels are connected or how the metamodels have to be extended for a seamless integration.

After the integration of BEDSL and OWL we present the integration of FODA with the present integrated metamodel which consists of BEDSL and OWL elements.

Finally we depict some parts of the integrated metamodel. Section 5 gives an example of using it.

#### 4.1 Integration of BEDSL and OWL

In this section we want to present eight steps where transformations are applied to integrate the ontology language OWL and the Comarch domain specific language BEDSL. These integration steps are described semi-formally to focus on the application of the approach and not to overload the paper by too much formal text.

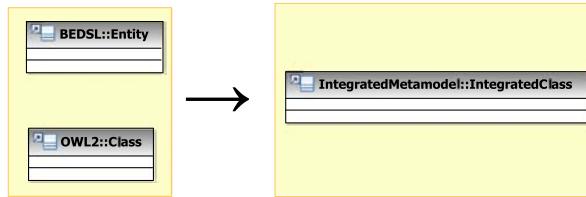
Beside of OWL class axioms that should be adopted on entities of BEDSL we want to restrict the relation between entities and attributes. Here we distinguish two cases: constraints and restrictions that are adopted on reference attributes and those that are adopted on simple attributes.

Constraints and restrictions on reference attributes are those that are adopted in OWL on object properties. So we have to materialize an association between **Entity** and **ReferenceAttribute** by a separate class that accepts OWL object property restrictions.

On the other side we have to materialize an association between **Entity** and **SimpleAttribute** by a separate class to adopt data property restriction on this relation.

**Step 1: Identify OWL Class and BEDSL Entity.** The first step of integration of the metamodels of the BEDSL and OWL occurs using the transformation depicted in figure 11. Here we express the fact that the concepts of an **Entity** class of the BEDSL metamodel and an **OWL Class** of the OWL 2 metamodel are equivalent. Using this transformation the classes are merged to one single class called **IntegratedClass**. Then on the one side we are able to associate attributes of the DSL with the new class **IntegratedClass** and on the other side it is allowed to adopt OWL constructs like class axioms on it.

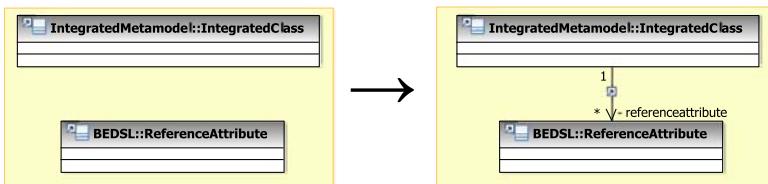
**Step 2: Connect Integrated Class with Reference Attribute.** Next after having the new class **IntegratedClass** we connect it with the class



**Fig. 11.** Step 1: Identify OWL Class and BEDSL Entity

**ReferenceAttribute** creating a new association between them. The reason is to have a relation between the two classes which could be constrained by OWL object property restrictions.

Figure 12 depicts the transformation. Here a *1-to-n* association is constructed between **IntegratedClass** and **ReferenceAttribute**, thus one instance of **IntegratedClass** can be connected with many instances of type **ReferenceAttribute**.

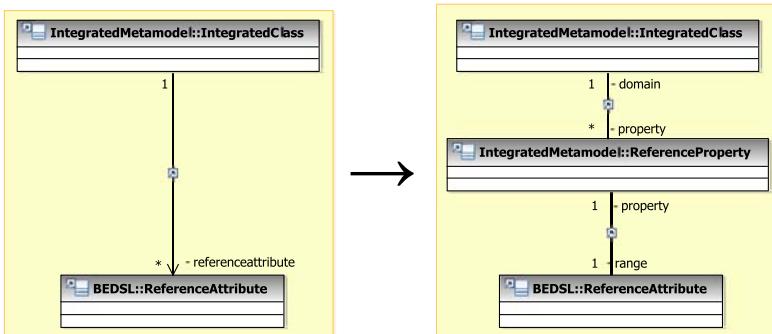


**Fig. 12.** Step 2: Connect Integrated Class with Reference Attribute

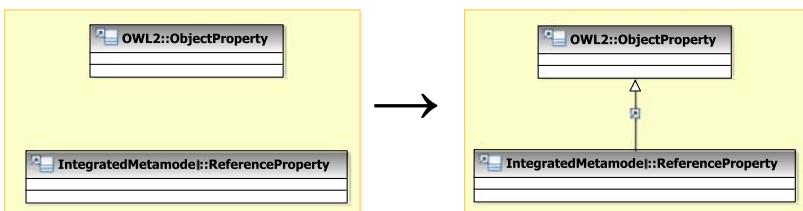
**Step 3: Transform Association to Class ReferenceProperty.** In OWL restrictions can be applied on object properties. For this purpose the association between class **IntegratedClass** and class **ReferenceAttribute** has to be transformed to a separate class leveraging this association to a class called **ReferenceProperty**.

Figure 13 depicts the transformation that supports the creation of the leveraging class **ReferenceProperty**. The source model of this transformation consists of classes **IntegratedClass** and **ReferenceAttribute** which are connected via an association. In place of the association in the source model there is a new class **ReferenceProperty** which is an element of the integrated metamodel. This class is incident with two associations which link to the classes **IntegratedClass** and **ReferenceAttribute**. Because the role names **domain** and **range** are used, the object property can later be reconstructed to get a consistent ontology for querying and reasoning.

**Step 4: Connect OWL Object Property and Reference Property.** Step 3 of the integration extended the association between **IntegratedClass** and **ReferenceAttribute** by a new class **ReferenceProperty**. So far the new class was not yet integrated with the **OWL ObjectProperty** class of the OWL 2



**Fig. 13.** Step 3: Transform Association to Class ReferenceProperty



**Fig. 14.** Step 4: Connect OWL Object Property and Reference Property

metamodel. So in step 4 we create a specialization relationship between the OWL object property class and the newly created **ReferenceProperty** class.

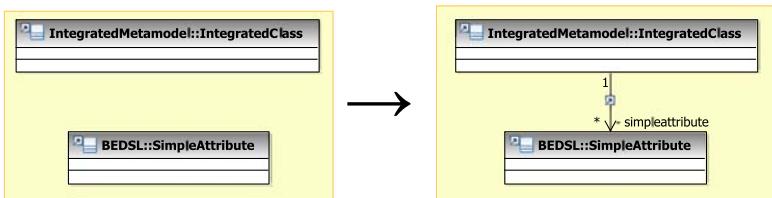
Figure 14 depicts the transformation. As a result now restrictions on object properties can be applied to the relation between **IntegratedClass** and **ReferenceAttribute**.

**Step 5: Connect IntegratedClass with Simple Attribute.** While in step 2 to 4 the integration of OWL object properties was described, we now consider the integration of OWL data properties with regard to simple attributes of the metamodel of the BEDSL. So analogously to step 2 we connect the **IntegratedClass** with the **SimpleAttribute** class by a new association.

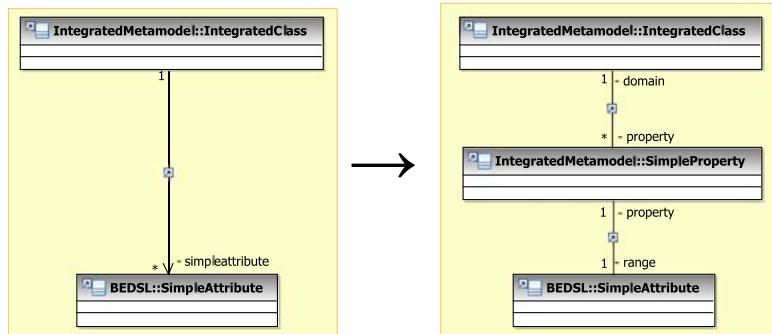
Figure 15 depicts this transformation. After the transformation we have a separate relation between **IntegratedClass** and **SimpleAttribute** which can be constrained by OWL data property restrictions.

**Step 6: Transform Association to Class SimpleProperty.** To adopt restrictions for OWL data properties on the association between **IntegratedClass** and **SimpleAttribute**, the association is materialized to a separate class called **SimpleProperty**.

Figure 16 depicts the transformation. In the target model of the transformation there now exists the new class **SimpleProperty**, which is connected with **IntegratedClass** and **SimpleAttribute**. Later domain and range of the data property could be reconstructed because of the role names at the association ends.



**Fig. 15.** Step 5: Connect IntegratedClass with Simple Attribute

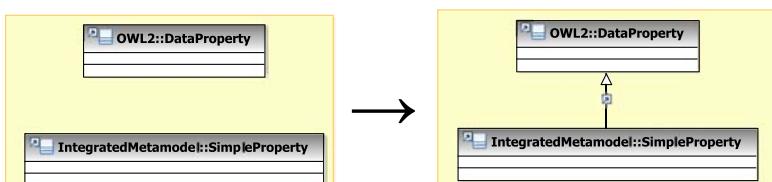


**Fig. 16.** Step 6: Transform Association to Class SimpleProperty

**Step 7: Connect OWL Data Property and Simple Property.** Step 6 of the integration extended the association between `IntegratedClass` and `SimpleAttribute` by a new class `SimpleProperty`. So far the new class was not integrated to the OWL `DataProperty` of the OWL 2 metamodel. So in step 7 we create a specialization relationship between the OWL data property class and the newly created `SimpleProperty` class.

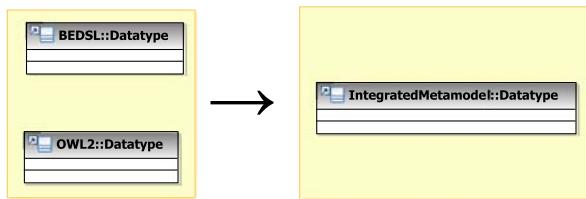
Figure 17 depicts the concrete transformation. As a result restrictions on data properties could be applied to the relation between `IntegratedClass` and `SimpleAttribute`.

**Step 8: Merge DSL Datatype and OWL Datatype.** The last step of integration merges the classes of `Datatype` in the BEDSL metamodel and in the OWL 2 metamodel, because both classes represent the same concept of a datatype.



**Fig. 17.** Step 7: Connect OWL Data Property and Simple Property

Figure 18 depicts the concrete transformation



**Fig. 18.** Step 8: Merge DSL Datatype and OWL Datatype

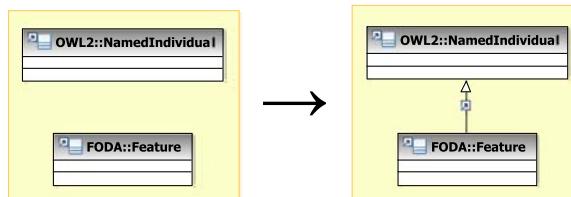
## 4.2 Integration of FODA and the Present Integrated Metamodel

After the integration of the ontology language OWL 2 and the domain specific language BEDSL we now want to show how features and a feature model can be integrated with the present enriched domain model.

On the one side we have the domain model and its corresponding metamodel which prescribe the domain specific language enriched by ontologies. On the other side we have the feature model, which is restricted by its corresponding metamodel (cf. figure 10). Now we want to integrate these two languages at the M2-layer.

To connect the FODA metamodel and the present integrated metamodel we need only one step. A feature model describes how feature instances are related to each other. Hence we can assume that a feature is a named individual in the OWL 2 metamodel.

Figure 19 depicts the transformation that creates the inheritance relation between class **Feature** and class **NamedIndividual** and thus is used to connect the two metamodels.



**Fig. 19.** Connect FODA Feature with OWL Individual

## 4.3 Integrated Metamodel

After having integrated the three languages BEDSL, OWL 2 and FODA we now present parts of the integrated metamodel as the result of all steps.

On the one side the metamodel provides all classes of the OWL 2 metamodel, on the other side all classes of the BEDSL and FODA metamodel can be used.

The intersection of the BEDSL and the OWL 2 metamodel is built by the classes **IntegratedClass**, **ReferenceProperty**, **SimpleProperty** and **Datatype**.

**IntegratedClass** on the one side can be linked to attributes of the domain specific language but on the other side can also adopt different OWL constructs like restrictions or class axioms which are in general allowed for the OWL class **ClassExpression**.

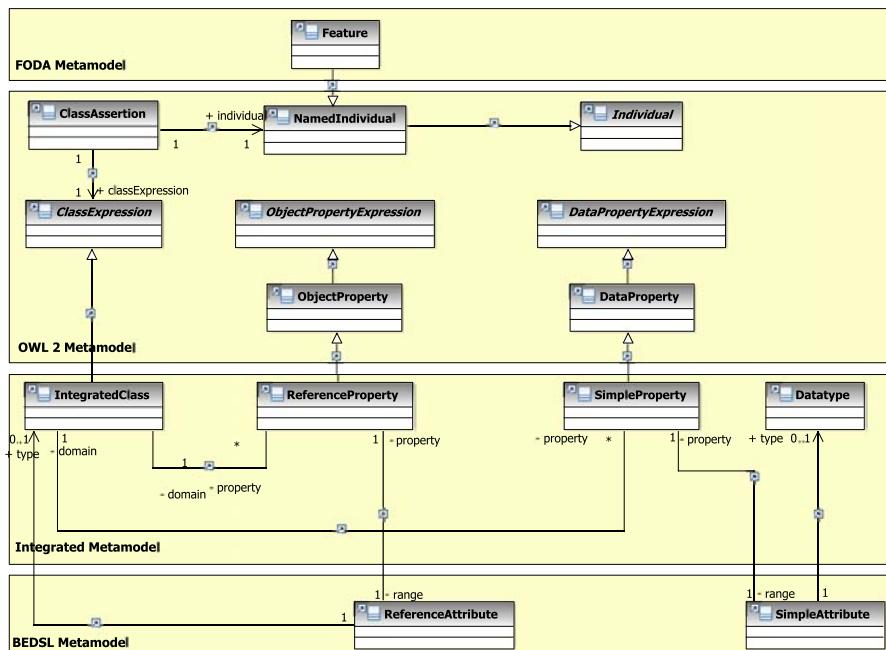
The class **ReferenceProperty** is a specialization of the OWL Class **ObjectProperty** and is associated with the DSL class **ReferenceAttribute**. Thus OWL object property restrictions can be adopted on the relation between **IntegratedClass** and **ReferenceAttribute**.

Besides OWL object property restrictions we consider the integration of OWL data property restrictions. Here we created the class **SimpleProperty** that represents the relation between **IntegratedClass** and **SimpleAttribute**. Using a specialization relationship between the OWL class **DataProperty** and **SimpleProperty** we can adopt OWL data property restrictions on the relation between **IntegratedClass** and **SimpleAttribute**.

The integration of FODA is realized by creating a specialization relationship between the classes **Feature** and **NamedIndividual**.

Figure 20 depicts in detail the relevant parts of the integrated metamodel that connect the OWL 2 and the BEDSL metamodel.

Considering the case study in section 2, we now are able to describe all relevant parts of Comarchs *Operations Support System* (OSS), just using the integrated



**Fig. 20.** Relevant parts of the integrated metamodel

metamodel. The BEDSL parts of the OSS are instances of elements of the BEDSL package in the integrated metamodel. The product line and configurations of the OSS are described by using the FODA package of the integrated metamodel. In addition, we are now able to define any semantics of the OSS and dependencies between the different languages using the OWL2 package of the integrated metamodel.

## 5 Example

In the following we give an example of using the integrated metamodel and show how we can project the enriched domain models to an ontology for querying and reasoning. We exemplify this using a small part of the model in figure 1.

Figure 21(a) depicts a part of the domain model in concrete syntax. Here we have the two concepts `NetworkDevice` and `Port`, which are connected via an association. This association is annotated with some *OWL text*, in our case we define `hasPort exactly 8 Port`. Thus we want to state that each network device has exactly eight ports.

Figure 21(b) depicts the domain model and its constraint in abstract syntax which is correspondent to the integrated metamodel. Here we have two instances of `IntegratedClass` called `NetworkDevice` and `Port`. These instances are connected via a `ReferenceProperty` and a `ReferenceAttribute`. The `ReferenceProperty` is used to adopt the OWL restriction, the `ReferenceAttribute` is used to point to the `Entity` instance.

Since we have the domain model and its constraints in abstract syntax the next step would be to project the domain model to a complete ontology for reasoning and querying and of course later for constraint checking. Figure 22 depicts which parts of the domain model are considered for constructing the ontology.

The two instances of `IntegratedClass` yield two separate OWL `Classes`. Thus in our case we have the OWL classes `NetworkDevice` and `Port`.

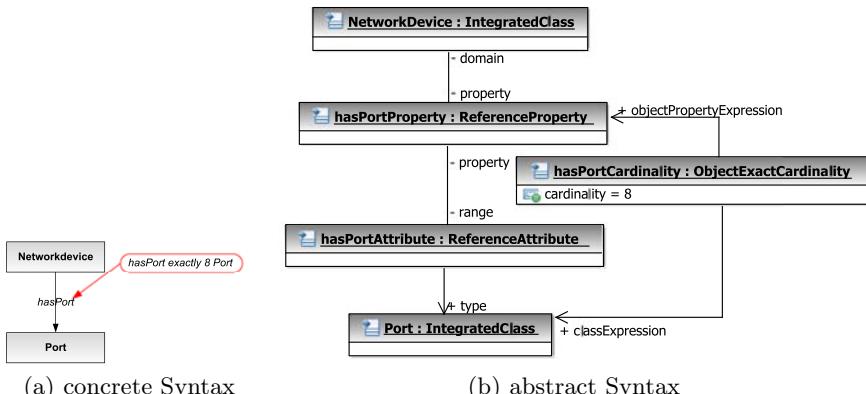
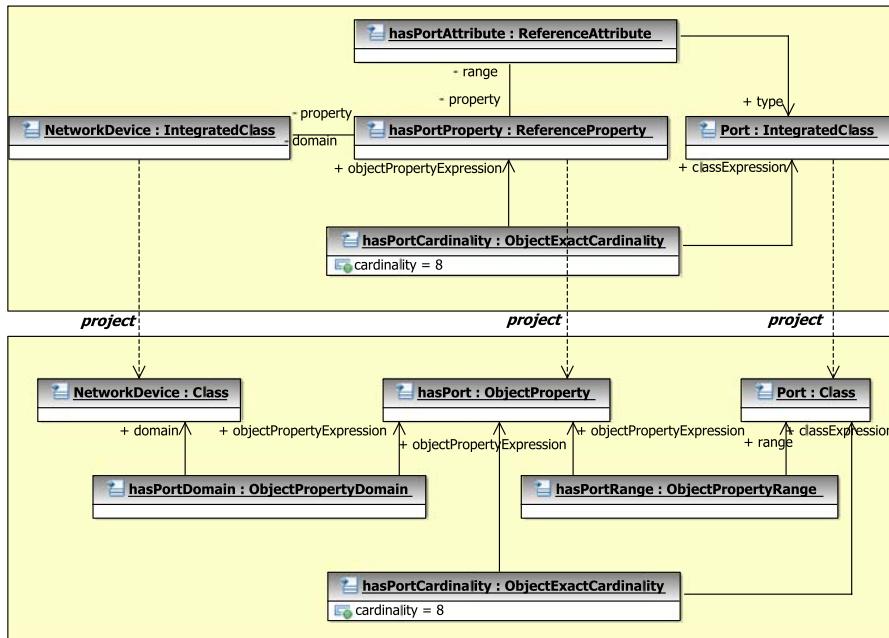


Fig. 21. Domain Model with constraint



**Fig. 22.** Projection of the Domain Model to an Ontology

The object property is constructed with regard to the instance of `ReferenceProperty` because in the integrated metamodel `ReferenceProperty` inherits from `ObjectProperty` of the OWL metamodel. The new instances of `ObjectPropertyDomain` and `ObjectPropertyRange` are derived from the unique role names of the links between the instances `NetworkDevice` and `Port`. At last the *exact cardinality restriction* is added to the ontology.

This procedure can be applied to the whole example in figure 5 analogously. Listing 1.1 depicts the complete extracted ontology in concrete syntax that results from the domain models and constraints (cf. figure 1).

In the top part of the listing we first have the definitions of all relevant classes. Important is that the class `NetworkDevice` inherits from the anonymous class `AvailableFor` value `ShowPorts`. Thus all individuals of `NetworkDevice` are connected via the object property `AvailableFor` with the individual `ShowPorts`. The constraint is extracted from the correspondent annotation of `NetworkDevice` (cf. figure 1).

After the class definition the two feature individuals `ShowPorts` and `AllocatePorts` are declared.

In the following the three object properties `HasState`, `HasPort`, and `AvailableFor` are declared. The constraints `HasPort` exactly 8 `Port` and `HasState` exactly 1 `State` result from the corresponding annotations in the domain model and define the cardinality restriction in the range section of each

object property. The domain part of the `AvailableFor` property is empty, because the following global condition defines it exactly.

**Listing 1.1.** Ontology extracted from the domain models and constraints

---

```

Class: owl:Thing

Class: NetworkDevice
  SubClassOf:
    AvailableFor value ShowPorts

Class: Port
  SubClassOf:
    owl:Thing

Class: State
  SubClassOf:
    owl:Thing

Class: FreePort
  SubClassOf:
    State
  DisjointWith:
    ReservedPort

Class: ReservedPort
  SubClassOf:
    State

*****
Individual: ShowPorts
  DifferentFrom:
    AllocatePort

Individual: AllocatePorts
  DifferentFrom:
    ShowPorts

*****
ObjectProperty: HasState
  Domain:
    Port
  Range:
    HasState exactly 1 State

ObjectProperty: HasPort
  Domain:
    NetworkDevice
  Range:
    HasPort exactly 8 Port

ObjectProperty: AvailableFor

*****
NetworkDevice and (hasPort some (Port and hasState some (FreePort)))
  EquivalentTo:
    NetworkDevice and AvailableFor value AllocatePorts

```

---

The last part of the ontology is extracted from the global constraint of the domain model, stating when a network device entity is available for the feature `AllocatePorts`. In the first line we define an anonymous class that describes

a network device which has some port which has the state free. In the third line of this condition we define again an anonymous class which describes all network devices that are available for the *AllocatePorts* feature. These two class definitions are stated as equivalent.

Since all this information is present in the integrated model, its projection (unparsing) to plane OWL text (here in Manchester Syntax style) is easily possible.

## 6 Related Work

Due to the fact that this paper provides an approach of integrating domain specific languages and ontology languages the following related work concentrates on approaches where ontologies are used with domain specific modeling and different domain specific languages are integrated or connected.

Modeling complex systems usually requires different domain specific languages. Hence the need of integrating them is apparent. Because of semantic overlap of languages, where synergies can be realized by defining bridges, ontologies provide the chance of semantic integration of domain specific languages [16].

A general idea of integrating UML-based models and ontologies is the TwoUse approach (Transforming and Weaving Ontologies and UML in Software Engineering) [17]. TwoUse contains a MOF-based metamodel for UML and OWL modeling and provides a UML profile as a syntactical basis, while our approach mainly deals with domain specific languages and uses their specific syntax for modeling. The querying in TwoUse is based on OCL DL, which extends the OCL basic library with operations for querying ontologies. Our approach is to project the relevant parts of the extended domain models to an ontology and thus using standard reasoners like Pellet or FaCT++ for querying and reasoning.

The core idea in [18] is to establish semantic connectors between different models using an ontology knowledge base. This knowledge base consists of different ontologies. An upper ontology can define concepts that are applicable in most, perhaps all, domains. An application ontology specializes the concepts from the upper ontology with domain-specific variants, where the constructs of this ontology are mapped to the metamodels of different domain specific languages. In fact in this approach ontologies are not integrated within the modeling languages itself. Instead of our approach the languages are not integrated at the metamodel level, they are connected by separate ontologies.

In [19] a domain composition approach is presented which occurs at the metamodel level. Here two types of relationships can be established between concepts of two different metamodels: associations, which are similar to UML associations and correspondences, which relate overlapping concepts. These so called meta-relationships are domain invariants and are shared by all applications. In addition relationships between models at the M1-layer are introduced in to allow designers to connect their models. Our approach tries to provide an integrated modeling at the M1-layer so that the designer can create one model which consists of different languages. The connection of different modeling languages is

forced by the integrated metamodel, thus in our approach the designer cannot connect different modeling languages manually at the M1 level.

An approach of semantic tool integration is presented in [20]. The approach uses an integrated data model and the integration is tool-based. Here the data model of a tool is mapped into other data models. In [20] the authors propose to establish an integrated data model first, and then map the data model of each tool into it. The integrated data model can be defined as a data model that is rich enough to contain data from any of the tools. Our idea is similar in a way that we first want to provide our integrated metamodel that is rich enough to contain all languages separately but also in common.

## 7 Conclusion

Using a non-trivial example, this paper showed that multi-way merge of domain specific languages on the level of metamodels is feasible. In more detail we showed that ontology languages like OWL can be included in this integration leading to a natural usage of OWL as a part of integrated DSL models.

The work described here was done manually. A prototype implementation as a proof of concept is being developed using the TGraph approach [21] that proved to be a good implementation means for software engineering tools.

The integration steps were described semi-formally to focus on the application of the approach. The rules used in section 4 were not formalized using some model transformation language (like e.g. ATL [22]) in order not to overload the paper by too much formal text.

Further work will focus on the formalization and implementation of the work described in this paper.

*Acknowledgement.* Our thanks go to Krzysztof Miksa and his colleagues from Comarch for providing us with the example that inspired this paper. Furthermore this work is supported by EU STReP-216691 MOST.

## References

1. Comarch: Definition of the case study requirements. MOST Project Deliverable (July 2008), <http://www.most-project.eu>
2. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute (1990)
3. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (December 2008), <http://www.w3.org/TR/2008/WD-owl2-syntax-20081202/>
4. IEC, International Engineering Consortium: Operations Support Systems (OSS) (2007), <http://www.iec.org/online/tutorials/oss/>
5. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.: The Manchester OWL Syntax. In: OWLED 2006 Second Workshop on OWL Experiences and Directions, Athens, GA, USA (2006)
6. OMG: Ontology Definition Metamodel. Object Modeling Group (November 2007)

7. Kelly, S., Tolvanen, J.: *Domain-Specific Modeling*. John Wiley & Sons, Chichester (2007)
8. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: *Proceedings of the 16th IEEE international conference on Automated software engineering*, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos (2001)
9. OMG: Meta Object Facility (MOF) Core Specification (January 2006), <http://www.omg.org/docs/formal/06-01-01.pdf>
10. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York (2003)
11. Happel, H.J., Seedorf, S.: Applications of Ontologies in Software Engineering. In: *International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, Athens, USA (November 2006)
12. Gasevic, D., Djuric, D., Devedzic, V.: MDA-based automatic OWL ontology development. *International journal on software tools for technology transfer (Print)* 9(2), 103–117 (2007)
13. Parreiras, F., Staab, S., Winter, A.: On marrying ontological and metamodeling technical spaces. In: *Foundations of Software Engineering*, pp. 439–448. ACM Press, New York (2007)
14. Kang, K., Lee, J., Donohoe, P.: Feature-oriented product line engineering. *IEEE Software* 19(4), 58–65 (2002)
15. Sun, J., Zhang, H., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: *ICECCS 2005: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, Washington, DC, USA, pp. 303–312. IEEE Computer Society, Los Alamitos (2005)
16. Gaševic, D., Djuric, D., Devedzic, V., Damjanovic, V.: Approaching OWL and MDA Through Technological Spaces. In: *The 3rd Workshop in Software Model Engineering (WiSME 2004)*, Lisbon, Portugal (2004)
17. Silva Parreiras, F., Staab, S., Winter, A.: TwoUse: Integrating UML Models and OWL Ontologies. *Arbeitsberichte aus dem Fachbereich Informatik* 16/2007, Universität Koblenz-Landau, Fachbereich Informatik (April 2007)
18. Brauer, M., Lochmann, H.: Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations. In: *Proceedings of 4th International Workshop on (Software) Language Engineering (ATEM 2007) co-located with MoDELS* (2007)
19. Estublier, J., Ionita, A., Vega, G.: A Domain Composition Approach. In: *Proc. of the International Workshop on Applications of UML/MDA to Software Systems (UMSS)*, Las Vegas, USA (June 2005)
20. Karsai, G., Gray, J., Bloor, G., Works, P.: Integration of Design Tools and Semantic Interoperability. In: *Engineering and Technical Management Symposium*, Dallas (2000)
21. Ebert, J., Riediger, V., Winter, A.: Graph Technology in Reverse Engineering, The TGraph Approach. In: Gimnich, R., Kaiser, U., Quante, J., Winter, A. (eds.) *10th Workshop Software Reengineering (WSR 2008)*, Bonn, GI. GI Lecture Notes in Informatics, vol. 126, pp. 67–81 (2008)
22. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) *MoDELS 2005. LNCS*, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)

# A Domain Specific Language for Composable Memory Transactions in Java

André Rauber Du Bois and Marcos Echevarria

PPGInf - Programa de Pós-Graduação em Informática,  
Universidade Católica de Pelotas  
CEP: 96010-000, Pelotas-RS, Brazil  
`{dubois,marcosge}@ucpel.tche.br`

**Abstract.** In this paper we present CMTJava, a domain specific language for *composable memory transactions* [7] in Java. CMTJava provides the abstraction of *transactional objects*. Transactional objects have their fields accessed only by special get and set methods that are automatically generated by the compiler. These methods return *transactional actions* as a result. A transactional action is an action that, when executed, will produce the desired effect. Transactional actions can only be executed by the `atomic` method. Transactional actions are first class values in Java and they are composable: transactions can be combined to generate new transactions. The Java type system guarantees that the fields of transactional objects will never be accessed outside a transaction. CMTJava supports the `retry` and `orElse` constructs from STM Haskell. To validate our design we implemented a simple transactional system following the description of the original Haskell system. CMTJava is implemented as a state passing monad using BBGA closures, a Java extension that supports closures in Java.

## 1 Introduction

Multi-core machines are now available everywhere. The traditional way to program these machines using threads, shared memory and locks for synchronization is difficult and has many pitfalls [16,22,24]. A promising approach to program multi-core machines is to use *software transactional memory* (STM). In this approach, accesses to shared memory are performed inside transactions, and a transaction is guaranteed to be executed atomically with respect to other concurrently executed transactions. Transactions can be implemented using *optimistic concurrency*: all transactions are executed concurrently with no synchronization, all writes and reads to shared memory are performed in a *transaction log*. When a transaction finishes, it validates its log to check if it has seen a consistent view of memory, and if so, its changes are committed to memory. If validation fails the log is discarded and the transaction is executed again with a fresh log. With atomic transactions the programmer still has to worry when critical sections should begin and end. But although having to delimit critical sections, the programmer does not need to worry about a locking protocol, that can induce problems like deadlocks.

In this paper, we present CMTJava, a domain specific language to write *composable memory transactions* in Java. CMTJava brings the idea of composable memory transactions in the functional language Haskell [7] to a object oriented context.

The main characteristics of the CMTJava system are:

- CMTJava provides the abstraction of *Transactional Objects*. Transactional objects have their fields accessed only by special get and set methods that are automatically generated by the compiler. These methods return *transactional actions* as a result. A transactional action is an action that, when executed, will produce the desired effect. Java's type system guarantees that the fields of transactional objects can only be accessed inside transactions. Transactions can only be executed with the `atomic` method. The `atomic` method takes a transaction as an argument and executes it atomically with respect to all other memory transactions. As transactions can not be executed outside a call to `atomic`, properties like *atomicity* (the effects of a transaction must be visible to all threads all at once) and *isolation* (during the execution of a transaction, it can not be affected by other transactions) are always maintained.
- Transactional actions are first class values in Java, they can be passed as arguments to methods and can be returned as the result of a method call. Transactions are composable: they can be combined to generate new transactions. Transactions are composed using STM blocks (Section 2), and we also support the `retry` construct, that allows possibly-blocking transactions to be composed sequentially and the `orElse` construct that allows transactions to be composed as alternatives.
- Transactions are implemented as a state passing monad (Section 3.3). Most of the simplicity of the implementation comes from the fact that we use a Java extension for closures to implement CMTJava. Hence, this paper is also a case support for closures in Java 7. We describe translation rules that translate CMTJava to pure Java + closures. Although the system was implemented in Java, the ideas could also be applied to any object oriented language that supports closures, e.g., C#.

The paper is organized as follows. First, in Section 2, CMTJava is presented through simple examples. Section 3, describes the implementation of the language primitives using a *state passing* monad. A simple prototype implementation of transactions is also described. Section 4 discusses related work and finally Section 5 presents conclusions and directions for future work.

## 2 Examples

In this section we present simple examples with the objective of describing how to program with CMTJava.

## 2.1 The Dinning Philosophers

In this section, an implementation of the *dinning philosophers problem* is given using the CMTJava DSL. The first class to be defined is `Fork`:

```
class Fork implements T0bject{
    private volatile Boolean fork = true;}
```

The `Fork` class has only one field and Java's `volatile` keyword is used to guarantee that threads will automatically see the most up-to-date value in the field. The `T0bject` interface works as a *hint* to the compiler, so it will generate automatically the code needed to access this class in transactions. It also generates two methods to access each field of the class. For the `Fork` class it will generate the following methods:

```
STM<Void> setFork(Boolean fork);
STM<Boolean> getFork();
```

The `setFork` and `getFork` methods are the only way to access the `fork` field. A value of type `STM<A>` represents a *transactional action* that when executed will produce a value of type `A`.

We can now define the `Philosopher` class as in Figure 1. The `STM{...}` block works like the `do` notation in STM Haskell [7]: it is used to compose transactions. The notation `STM{ a1; ...; an }` constructs a STM action by glueing together smaller actions  $a_1; \dots; a_n$  in sequence. Using STM blocks it is

```
class Philosopher implements Runnable{

    Fork right;
    Fork left;
    int id;

    Philosopher(int i, Fork r, Fork l)
    {
        id = i;
        right = r;
        left = l;
    }

    public void run()
    {
        STM<Void> t1 = STM{
            acquireFork(r);
            acquireFork(l)
        }
        atomic(t1);

        System.out.println("Philosopher " + id +
                           " is eating!!!");

        STM<Void> t2 = STM{
            releaseFork(r);
            releaseFork(l)
        }
        atomic(t2);
    }
    (...)
```

Fig. 1. The Philosopher class

possible to implement the `STM<Void> acquireFork(Fork f)` and `STM<Void> releaseFork(Fork f)` methods that are missing on Figure 1:

```
STM<Void> acquireFork(Fork f)
{
    STM<Void> r = STM{
        Boolean available <- f.getFork();
        if (!available)
            retry();
        else
            f.setFork(false)
    }
    return r;
}
```

The `acquireFork` method first checks the current state of the `Fork` using the `getFork()` method. If the fork is not available it blocks the transaction by calling `retry`. If the fork was available, it is set to `false`, i.e., not available. The call to `retry` will *block* the transaction, i.e., the transaction will be aborted and restarted from the beginning. The transaction will not be re-executed until at least one of the fields of the `TObjects` that it has accessed is written by another thread. In the case of the `acquireFork` method, when it calls `retry` the philosopher will be suspended until a neighbor philosopher puts the fork back in the table. The variables created inside a transactional action are *single assignment* variables. These variables are used only to carry the intermediate state of the transaction being constructed and do not need to be logged by the runtime system supporting transactions. To emphasize that these variables are different than the ordinary Java variable a different symbol for assignment is used (`((<-)`).

The `releaseFork` method has a simple implementation:

```
STM<Void> releaseFork(Fork f)
{
    return f.setFork(true);
}
```

It returns a transaction that when executed will set the fork field to `true`. When a philosopher calls `releaseFork` it already holds the fork so there is no need to check its current state.

## 2.2 The `orElse` Method

CMTJava also provides the `orElse` method that allows the composition of transactions as *alternatives*

```
public static <A> STM<A> orElse (STM<A> t1, STM<A> t2)
```

`orElse` takes two transactions as arguments and returns a new transaction. The transaction

```
orElse(t1,t2);
```

will first execute `t1`; if it retries then `t1` is discarded and `t2` is executed. If `t2` also retries then the whole transaction will retry.

In Figure 2 a simple implementation of a class representing a bank account is given. Using `orElse` we could extend the `Account` class with the following method:

```
public static STM<Void> withdraw2Accounts(Account c1, Account c2)
{
    return orElse(c1.withdraw(1000.0), c2.withdraw(1000.0));
}
```

The `withdraw2Accounts` method first tries to withdraw 1000.0 from `c1`. If there is not enough money in that account it tries to withdraw the same amount from `c2`

```
class Account implements TObject{
    private volatile Double Balance;

    public STM<Void> withdraw(Double n)
    {
        STM<Void> t = STM{
            Double b <- getBalance();
            if (b < n)
                retry()
            else
                setBalance(b-n)
        };
        return t;
    }

    public STM<Void> deposit (Double n)
    {
        return STM{
            Double b <- readBalance();
            writeBalance(b + n)
        };
    }

    public static STM<Void> transfer(Account a1, Account a2, Double money)
    {
        return STM{
            a1.withdraw(money);
            a2.deposit(money)
        }
    }
}
```

**Fig. 2.** The `Account` class

## 3 Implementation

### 3.1 Java Closures

To implement CMTJava we used *BGGA Closures*, a Java extension that supports *anonymous functions* and *closures* [2]. This extension is a prototype implementation of a JSR proposal to add closures to Java 7 [3].

In BGGA, an anonymous function can be defined using the following syntax:

*{ formal parameters => statements expression }*

where *formal parameters*, *statements* and *expression* are optional. For example, *{ int x => x + 1 }* is a function that takes an integer and returns its value incremented by one. An anonymous function can be invoked using its `invoke` method:

```
String s = {=> "Hello!"}.invoke();
```

An anonymous function can also be assigned to variables:

```
{int => void} func = {int x => System.out.println(x)};
```

The variable `func` has type *{int => void}*, i.e., a *function type* meaning that it can be assigned to a function from `int` to `void`. Function types can also be used as types of arguments in a method declaration.

A *closure* is a function that captures the bindings of free variables in its lexical context:

```
public static void main(String[] args) {
    int x = 1;
    {int=>int} func = {int y => x+y };
    x++;
    System.out.println(func.invoke(1)); // will print 3
}
```

A closure can use variables of the enclosing scope even if this scope is not active at the time of closure invocation e.g., if a closure is passed as an argument to a method it will still use variables from the enclosing scope where it was created.

### 3.2 Monads

A monad is a way to structure computations in terms of values and sequences of computations using those values [1]. A monad is used to describe computations and how to combine these computations to generate new computations. For this reason monads are frequently used to embed domain specific languages in functional languages for many different purposes, e.g., I/O and concurrency [20], Parsers [13], controlling robots [19], and memory transactions [7]. A monad can be implemented as an abstract data type that represents a container for a computation. These computations can be created and composed using three basic operations: *bind*, *then* and *return*. For any monad `m`, these functions have the following type in Haskell:

```
bind :: m a -> (a -> m b )-> m b
then :: m a -> m b -> m b
return :: a -> m a
```

The type `m a` represents a computation in the monad `m` that when executed will produce a value of type `a`. The `bind` and `then` functions are used to combine computations in a monad. `bind` executes its first argument and passes the result to its second argument (a function) to produce a new computation. `then` takes two computations as arguments and produces a computation that will execute them one after the other. The `return` function creates a new computation from a simple value.

The next section presents the implementation of these three operations for the STM monad in Java.

### 3.3 The STM Monad

The monad for STM actions is implemented as a *state passing monad*. The state passing monad is used for threading a state through computations, where each computation returns an altered copy of this state. In the case of transactions, this state is the meta-data for the transaction, i.e., its log.

The STM class is implemented as follows:

```
class STM<A> {
    { Log => STMResult<A> } stm;

    STM ({ Log => STMResult<A> } stm) {
        this.stm = stm;
    }
}
```

The STM class is used to describe transactions, and it has only one field: a function representing the transaction. A transaction `STM<A>` is a function that takes a `Log` as an argument and returns a `STMResult<A>`. The `Log` represents the current state of the transaction being executed and `STMResult` describes the new state of the transaction after its execution.

The `STMResult<A>` class has three fields:

```
class STMResult<A> {

    A result;
    Log newLog;
    int state;

    (...)

}
```

the first field (**result**) is the result of executing the **STM** action, the second (**newLog**) is the resulting log, and the third is the current **state** of the transaction. Executing a **STM** action can put the transaction in one of two states: it is either **VALID** meaning the the transaction can continue, or the state can be set to **RETRY** meaning that the **retry** construct was called and the transaction must be aborted.

The **bind** method is used to compose transactional actions:

```
public static <A,B> STM<B> bind ( STM<A> t, {A => STM<B>} f ) {
    return new STM<B> ( {Log l1 =>
        STMResult<A> r = t.stm.invoke(l1);
        STMResult<B> re;
        if (r.state == STMRTS.VALID) {
            STM<B> nst = f.invoke(r.result);
            re = nst.stm.invoke(r.newLog);
        } else {
            re = new STMResult(null, r.newLog, STMRTS.RETRY);
        }
        re
    } );
}
```

The **bind** method takes as arguments an **STM<A>** action **t** and a function **f** of type **{A => STM<B>}** and returns as a result a new **STM<B>** action. The objective of **bind** is to combine **STM** actions generating new actions. The resulting **STM** action takes a log (**l1**) as an argument and invokes the **t** action by passing the log to it (**t.stm.invoke(l1)**). If the invocation of **t** did not retry (i.e., its state is **VALID**) then **f** is called generating the resulting **STM<B>**. Otherwise the execution flow is abandoned and **bind** returns a **STMResult** with the state set to **RETRY**.

The **then** method is implemented in a very similar away

```
public static <A,B> STM<B> then ( STM<A> a, STM<B> b ) {
    return new STM<B> ( {Log l1 =>
        STMResult<A> r = a.stm.invoke(l1);
        STMResult<B> re;
        if (r.state == STMRTS.VALID) {
            re = b.stm.invoke(r.newLog);
        } else {
            re = new STMResult(null, r.newLog, STMRTS.RETRY);
        }
        re
    } );
}
```

The **then** method is a sequencing combinator: it receives as arguments two **STM** actions and returns an action that will execute them one after the order.

Finally, the `stmreturn` method is used to *insert* any object `A` into the STM monad:

```
public static <A> STM<A> stmreturn (A a) {
    return new STM<A>({ Log 1 => new STMResult(a,1,STMRTS.VALID) });
}
```

The `stmreturn` method is like Java's `return` for STM blocks. It takes an object as an argument and creates a simple transaction that returns this object as a result. It can also be used to create new objects inside a transaction. For example, the `insert` method returns a transaction that inserts a new element at the end of a linked list:

```
class Node implements TObject{
    private volatile Integer value;
    private volatile Node next;
    (...)

    public STM<Void> insert(Integer v)
    {
        return STM {
            Node cnext <- getNext();
            if( cnext == null) {
                STM{
                    Node r <- stmreturn (new Node(v,null));
                    setNext(r)
                }
            }else
                cnext.insert(v)
        }
    }
}
```

### 3.4 STM Blocks

STM blocks are translated into calls to `bind` and `then` using the translation rules given in Figure 3. Translation rules for STM blocks are very similar to the translation rules for the `do` notation described in the Haskell report [21].

```
STM{ type var <- e; s} = STMRTS.bind( e, { type var => STM { s }})

STM{ e ; s }           = STMRTS.then( e, STM{ s })

STM{ e }                = e
```

**Fig. 3.** Basic translation schemes for STM blocks

For example, the following implementation of a `deposit` method:

```
public STM<Void> deposit (Account a, Double n)
{
    return STM{
        Double balance <- a.getBalance();
        a.setBalance(balance + n)
    };
}
```

is translated to

```
public STM<Void> deposit (Account a, Double n)
{
    return STMRTS.bind(a.getBalance(), { Double balance =>
        a.setBalance(balance + n)} );
}
```

### 3.5 Compiling Objects

A class like `Fork` given in section 2.1 is compiled to the class in Figure 4, using the translation rules given in Appendix A.

```
class Fork implements TObject {

    private volatile Boolean fork = true;
    private volatile PBox<Boolean> forkBox =
        new PBox<Boolean> ( {Boolean b => fork = b; } , { => fork} );

    public STM<Void> setFork (Boolean b) {
        return new STM<Void>({Log l =>
            LogEntry<Boolean> le = l.contains(forkBox);
            if(le!=null) {
                le.setNewValue(b);
            } else {
                l.addEntry(new LogEntry<Boolean>(forkBox,fork,b));
            }
            new STMResult(new Void(), l,STMRTS.VALID)} );
    }

    public STM<Boolean> getFork() {
        return new STM<Boolean> ({Log l =>
            Boolean result;
            LogEntry<Boolean> le = l.contains(forkBox);
            if(le!=null) {
                result = le.getNewValue();
            } else {
                result = fork;
                l.addEntry(new LogEntry<Boolean>(forkBox,fork,fork));
            }
            new STMResult(result, l,STMRTS.VALID)} );
    }
}
```

**Fig. 4.** The Fork class after compilation

Every `TObject` has a `PBox` for each of its fields. A `PBox` works like a pointer to the field. The constructor of the `PBox` class takes two arguments: one is a function that updates the current value of a field and the other is a function that returns the current value of a field. These functions are used by `atomic` to commit a transaction (Section 3.8).

The `setFork` and `getFork` methods are used to access the `fork` field in a transaction. The `setFork` method first verifies if the `PBox` for the field is already present in the transaction log using the `contains` method of the `Log`. A log is a collection of log entries:

```
class LogEntry<A> {
    PBox<A> box;
    A oldValue;
    A newValue;
    (...)
```

Each `LogEntry` represents the current state of a field during a transaction. It contains a reference to the field's `PBox`, the original content of the field (`oldValue`) and the current value of the field (`newValue`). If the log already contains an entry for `fork` then this entry is updated so its `newValue` field contains a reference to the argument of the `set` method. If the log contains no entry for `fork` a new entry is added where the `oldValue` contains the current content of `fork` and `newValue` is set to the argument of `setFork`.

`getFork` will first check if there is an entry for `fork` in the log. If so, it returns the current value of the `newValue` field of the entry, otherwise it adds a new entry to the log where both `oldValue` and `newValue` contain the current value in the `fork` field.

### 3.6 The `retry` Method

The `retry` method has a simple implementation:

```
public static STM<Void> retry() {
    return new STM<Void>({ Log l =>
        new STMResult(new Void(), l, STMRTS.RETRY)});
```

It simply stops the current transaction by changing the state of the transaction to `STMRTS.RETRY`. When a transaction returns `STMRTS.RETRY` the transaction is aborted (see the implementations for `bind` and `then` in Section 3.3).

The `atomic` method is responsible for restarting aborted transactions (Section 3.8).

### 3.7 The `orElse` Method

The `orElse` method is used to compose transactions as alternatives. To implement `orElse`, proper nested transactions are needed to isolate the execution of

the two alternatives. If the first alternative calls `retry`, all the updates that it did must be invisible while executing the second alternative.

The `orElse` method is implemented in a similar way to `orElse` in STM Haskell. Each alternative is executed using a new log called *nested log*. All writes to fields are recorded in the nested log while reads must consult the nested log and the log for the enclosing transaction. If any of the alternatives finishes without retrying, its nested log is merged with the transaction log and a `VALID` `STMResult` is returned containing the merged log. If the first alternative calls `retry`, then the second is executed using a new nested log. If the second alternative also calls `retry`, all three logs are merged and an `STMResult` containing the merged log and state set to `RETRY` is returned. All logs must be merged so that the aborted transaction will be executed again when one of the fields accessed in the transaction is modified by other transaction (Section 3.8). The reader should notice that, if both alternatives that retried modified the same field of a `TObject`, the merged log will contain only one entry for that field. That does not matter as the merged log will be never used for committing, since the transaction was aborted. The final log is used to decide when the transaction should be restarted (Section 3.8).

### 3.8 The atomic Method

An STM action can be executed by calling `atomic`:

```
public static <A> A atomic (STM<A> t)
```

The `atomic` method takes as an argument a transaction and executes it atomically with respect to other concurrent calls to `atomic` in other threads.

In order to execute a transactional action, the `atomic` method generates a fresh Log and then invokes the transaction:

```
Log l = STMRTS.generateNewLog();
STMResult<A> r = t.stm.invoke(l);
```

As described before, an invoked transaction will return a `STMResult` containing the result of executing the transaction, a log and the state of the transaction that can be either `VALID` or `RETRY`. A Log contains, besides log entries, a global lock that is used by `atomic` to update shared data structures when committing.

If the state of the executed transaction is `RETRY`, `atomic` acquires the global lock and validates the log of the transaction. To validate a log, each log entry is checked to see if its `oldValue` contains the same object that is currently in the field of the `TObject`. The current content of a field is accessed using the `PBox` in the log entry. If the log is invalid, the transaction is automatically restarted with a fresh log. If the log is valid, the transaction is only restarted once at least one of the fields that it accessed is modified by another thread. To do that, `atomic` first creates a `SyncObject` for the transaction. A `SyncObject` has two methods:

```
public void block();
public void unblock();
```

When a thread calls the `block` method of a `SyncObject`, the thread will block until some other thread calls the `unblock` method of the same object. If a thread calls `unblock`, and the blocked thread was already unblocked, the method simply returns with no effect. If a thread calls `unblock` before `block` was called, the thread waits for the call to `block` before continuing. Every PBox also contains a list of `SyncObjects` for the threads waiting for that field to be updated. After creating the `SyncObject`, it is added to the list of blocked threads in every PBox present in the transaction log. Then the transaction releases the global lock and calls the `block` method of the `SyncObject`, suspending the transaction. When the transaction is awoken, meaning that at least one of the `TObjects` in its log was updated, it will be executed again with a fresh log.

If the state of the transaction executed by `atomic` is `VALID`, `atomic` acquires the global lock and validates the transaction log. If valid, the transaction commits by modifying all `TObjects` with the content of its log. It also awakes all threads waiting for the commit. This is done by calling the `unblock` method of the `SyncObjects` contained in the PBoxes in the log. If the transaction log is invalid, the log is abandoned and the transaction is executed again with a fresh log.

## 4 Related Work

Transactional memory was initially studied as a hardware architecture [10,23,5]. Software transactional memory [25] is the idea of implementing all transactional semantics in software. Most works on STM in Java provide low level libraries to implement transactions [9,8,17]. Harris and Fraser [6] provide the first language with constructs for transactions. Their Java extension gives an efficient implementation of Hoare's conditional critical regions [11] through transactions, but transactions could not be easily composed. The Atomos language [4] is a Java extension that supports transactions through atomic blocks and also the `retry` construct to block transactions. Transactions are supported by the Transactional Coherence and Consistency hardware transactional memory model (TCC) [18], and programs are run on a simulator that implements the (TCC) architecture. No support composing transactions as alternatives is given.

CMTJava builds on work done in the Haskell language, by bringing the idea of composable memory transactions into an object oriented context. STM Haskell [7], is a new concurrency model for Haskell based on STM. Programmers define *transactional variables* (`TVars`) that can be read and written using two primitives:

```
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM a
```

The `readTVar` primitive takes a `TVar` as an argument and returns an STM action that, when executed, returns the current value of the `TVar`. The `writeTVar` primitive is used to write a new value into a `TVar`. In CMTJava, each field of a transactional object works as a `TVar` and each field has its get/set method that work just like `readTVar` and `writeTVar`. In STM Haskell, STM actions can be composed together using the same `do` notation used to compose IO actions in Haskell:

```

addTVar :: TVar Int -> Int -> STM ()
addTVar tvar i = do
    v <- readTvar tvar
    writeTVar tvar (v+i)

```

STM blocks in CMTJava are just an implementation of the `do` notation available in Haskell. The STM system described here is very similar to the one described in the STM Haskell paper, the main difference being that STM Haskell is implemented to run on uniprocessors. This simplifies the implementation as all calls to the C functions that support transactions in the runtime system are run without interruption. The simple implementation of CMTJava given in this paper can be executed on multi-core machines, as we use a global lock to avoid races during commits, at the cost of a huge bottleneck imposed by the lock.

In [12], an implementation of composable memory transactions using pure Haskell is given. CMTJava uses an implementation of the STM monad that is very similar to the one described in this paper, although CMTJava's implementation of the low level mechanisms to support transactions is very different, as it is based on the STM Haskell runtime system.

## 5 Conclusions and Future Work

In this paper we have presented CMTJava, a domain specific language for composable memory transactions in Java, in the style of STM Haskell. CMTJava guarantees that fields of transactional objects will only be accessed inside transactions. CMTJava is translated to pure Java + closures, that are supported by the BBGA Java extension [2]. Although the system was implemented in Java, the ideas could also be applied to any object oriented language that supports closures.

There are many directions for future work. A semantics for the language could be provided by extending Feather Weight Java [14] with the semantics for composable memory transactions in Haskell. Exceptions inside an `atomic` call could be handled in the same way as in STM Haskell.

The prototype implementation of the STM system presented in this paper is very naive and is given only to illustrate the concepts presented and as a proof of concept. Every time a field of a `TObject` is accessed the system must search in an array of `LogEntries` for the right entry before using the content of the field. The single lock used for committing transactions is also a bottleneck in the system. We plan to re-implement CMTJava in the future using a low level library for transactions in Java (e.g., [8]) or using other optimized technique for the implementation of transactional memory (a survey on the subject is given in [15]).

The source code for CMTJava can be obtained by contacting the authors.

**Acknowledgment.** The authors would like to thank CNPq/Brazil for the financial support provided.

## References

1. All About Monads. WWW page (December 2008), [http://www.haskell.org/all\\_about\\_monads/html/index.html](http://www.haskell.org/all_about_monads/html/index.html)
2. Java Closures. WWW page (December 2008), <http://www.javac.info/>
3. JSR Proposal: Closures for Java. WWW page (December 2008), <http://www.javac.info/consensus-closures-jsr.html>
4. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., Olukotun, K.: The ATOMOS transactional programming language. ACM SIGPLAN Notices 41(6), 1–13 (2006)
5. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency, New York, NY, USA, vol. 32, p. 102. ACM, New York (2004)
6. Harris, T., Fraser, K.: Language support for lightweight transactions. ACM SIGPLAN Notices 38(11), 388–402 (2003)
7. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005. ACM Press, New York (2005)
8. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SPNOTICES: ACM SIGPLAN Notices 41 (2006)
9. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC: 22th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (2003)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture (1993)
11. Hoare, C.A.R.: Towards a theory of parallel programming. Operating System Techniques, 61–71 (1972)
12. Huch, F., Kupke, F.: A high-level implementation of composable memory transactions in concurrent haskell. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 124–141. Springer, Heidelberg (2006)
13. Hutton, G., Meijer, E.: Monadic Parsing in Haskell. Journal of Functional Programming 8(4), 437–444 (1998)
14. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS 23(3), 396–459 (2001)
15. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool (2006)
16. Lee, E.A.: The problem with threads. IEEE Computer 39(5), 33–42 (2006)
17. Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W.N., Scott, M.L.: Lowering the overhead of nonblocking software transactional memory. revised, University of Rochester, Computer Science Department (May 2006)
18. McDonald, A., Chung, J., Chafi, H., Minh, C.C., Carlstrom, B.D., Hammond, L., Kozyrakis, C., Olukotun, K.: Characterization of TCC on chip-multiprocessors. In: Proc. 14th International Conference on Parallel Architecture and Compilation Techniques (14th PACT 2005), pp. 63–74. IEEE Computer Society, Los Alamitos (2005)
19. Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 91–105. Springer, Heidelberg (1999)

20. Peyton Jones, S.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Engineering theories of software construction, pp. 47–96. IOS Press, Amsterdam (2001)
21. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Journal of Functional Programming 1(13) (2003)
22. Peyton Jones, S.: Beautiful Concurrency. O'Reilly, Sebastopol (2007)
23. Rajwar, R., Herlihy, M., Lai, K.K.: Virtualizing transactional memory. In: ISCA, pp. 494–505. IEEE Computer Society, Los Alamitos (2005)
24. Rajwar, R., James, G.: Transactional execution: Toward reliable, high-performance multithreading. IEEE Micro. 23(6), 117–125 (2003)
25. Shavit, N., Touitou, D.: Software transactional memory. DISTCOMP: Distributed Computing, 10 (1997)

## Appendix A: Translation Rules for TObjects

Translation rules for `TObjects` are given as Haskell functions. These functions take as arguments Strings (i.e., names of variables and their types) and return as result a String representing the code to be generated. Strings in Haskell are represented using double quotes and the `(++)` operation is used to concatenate two strings.

`PBox` objects are generated using the `genPBox` function:

```
type Type = String
type VarName = String

genPBox :: Type -> VarName -> VarName -> String
genPBox vtype var parName =
  "private volatile PBox<" ++ vtype ++ "> " ++
  var ++ "PBox = new PBox<" ++ vtype ++ "> (" ++
  vtype ++ " " ++ parName ++ " => " ++
  var ++ " = " ++ parName ++ " ; } , { => "++var++"});"
```

The parameters for `genPBox` are the type of the field of the transactional object (`vtype`), the name of the field `var`, and a random name for the parameters of the closures being defined `parName`.

Set methods for `TObjects` are generated using `genSetMethod`:

```
genSetMethod :: Type -> VarName -> VarName -> VarName -> String
genSetMethod vtype var parName boxName =
  "public STM<Void> set"++var++"("++vtype++" "++parName++")\n" ++
  "{\n" ++
  "  return new STM<Void>({Log l => \n" ++
  "  LogEntry<"++vtype++"> le = l.contains("++boxName++");\n" ++
  "  if(le!=null) {\n" ++
  "    le.setNewValue("++parName++");\n" ++
  "  } else {\n" ++
  "    l.addEntry(new LogEntry<"++vtype++">("++boxName++", "++
```

```

    var++,"++parName++));\n"++
" }\n" ++
" new STMResult(new Void(), 1,STMRTS.VALID)} );\n" ++
"}\n"

```

Get methods for TObjects are generated using genGetMethod:

```

genGetMethod :: Type -> VarName -> VarName -> VarName -> String
genGetMethod vtype var parName boxName =
"public STM<++vtype ++> get"++var++"() {\n"++
" return new STM<++vtype++> ({Log l =>\n"++
"     ++vtype++ " result;\n"++
"     LogEntry<++vtype++> le = l.contains("++boxName++");\n"++
"     if(le!=null) {\n"++
"         result = le.getNewValue();\n"++
"     } else {\n"++
"         result ="++var++";\n"++
"         l.addEntry(new LogEntry<++vtype++>("++boxName++"
"         ,"+var++","+var++")); \n"++
"     }\n"++
"     new STMResult(result,1,STMRTS.VALID)} );\n"++
"}\n"

```

# CLOPS: A DSL for Command Line Options

Mikoláš Janota, Fintan Fairmichael, Viliam Holub,  
Radu Grigore, Julien Charles, Dermot Cochran, and Joseph R. Kiniry

School of Computer Science and Informatics,  
Lero — The Irish Software Engineering Research Centre, and  
The Complex and Adaptive Systems Laboratory (CASL),  
University College Dublin, Ireland

**Abstract.** Programmers often write custom parsers for the command line input of their programs. They do so, in part, because they believe that both their program’s parameterization and their option formats are simple. But as the program evolves, so does the parameterization and the available options. Gradually, option parsing, data structure complexity, and maintenance of related program documentation becomes unwieldy. This article introduces a novel DSL called CLOPS that lets a programmer specify command line options and their complex inter-dependencies in a declarative fashion. The DSL is supported by a tool that generates the following features to support command line option processing: (1) data structures to represent option values, (2) a command line parser that performs validity checks, and (3) command line documentation. We have exercised CLOPS by specifying the options of a small set of programs like `ls`, `gzip`, and `svn` which have complex command line interfaces. These examples are provided with the Open Source release of the CLOPS system.

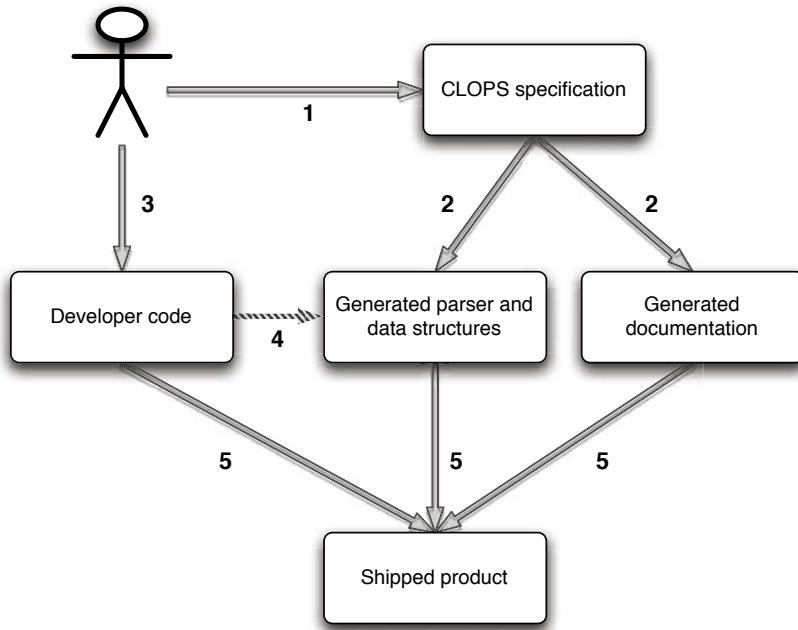
## 1 Introduction

A boiling frog is a well known phenomenon in software development. Just as a frog meets its fate in water that is slowly heated, so do software projects become unmanageable as their complexity gradually increases.

Processing of command line options is not an exception. The first few versions of a program have small number of options that are easy to process and do not have many dependencies between them.

No wonder then, that many programmers, with famous last words “this is easy” on their lips, opt for custom code for option processing. As the number of the program’s options and their inter-dependencies grows, option processing code gets bloated and diverges from its documentation.

This article suggests an alternative by providing a DSL that enables programmers to explicitly, declaratively capture options and their inter-dependencies. Furthermore, a tool is provided that produces documentation, code for command line processing, and data structures. The data structures produced represent the values of the options and are automatically populated during parsing.



**Fig. 1.** Developer usage scenario

The observations described above shaped the language proposed in this article. On the one hand, simple option sets must be easy to write when the program is in its early stages. But on the other hand, a set of options must be easy to evolve and even complicated option sets must be supported.

There are two types of users involved in using CLOPS — the developer who specifies the command line interface and integrates the command line parser with the main program body, and the end-user who interacts with the final product by executing it and providing options on the command line. Fig. 1 gives a diagrammatic representation of a developer’s usage scenario for creating a command line program using CLOPS.

The *developer* describes the command line of the tool in the CLOPS DSL(1). This description is used by the CLOPS tool to produce code and documentation(2). The code is used as a library(4) by the developer to parse the command line(3). The result of parsing is a set of data structures (options) that can be easily queried. The documentation is typically presented to the end-user(5).

The *end-user* is typically not aware that the developer’s tool uses CLOPS. This is despite the fact that CLOPS parses the command line, provides many of the error messages, and generates the documentation that the user consults (such as a `man` page and the usage message printed by the program itself).

This article contains several core contributions:

- We identify the main concepts seen in command line processing and describe their interdependencies (Section 4).
- We propose a novel DSL called CLOPS for describing command line option interfaces. Its syntax and semantics are described in Section 5.
- We provide a tool (its implementation is discussed in Section 6) that reads in a description in this language and generates multiple artifacts:
  - a Java implementation of the command line parser,
  - documentation, and
  - data structures that are used to store the result of parsing.
- (Section 7) summarizes results of applying CLOPS to well-known programs like `ls`; interesting cases are discussed and the CLOPS description is compared to existing implementations.
- Section 8 is a summary of experience of one of the authors that have applied CLOPS in projects he has been working on.

Section 9 elaborates how our research relates to other parts of Computer Science, namely Software Product Lines. Section 10 is an overview of other approaches tackling command line processing and Section 11 provides pointers for further research and improvements. The upcoming section sets up the ground for further discussion and is followed by a light-weight exposure to CLOPS DSL in Section 3.

## 2 Background

*Command line utilities* are widespread in the software community and hence there is no need to go into great detail about them. Nevertheless, several properties of command line options that will be important for later discussion must be reviewed.

A *program* is run from a command line with a list of *options* that affect its behavior. For instance, the program `ls` lists files in a multi-column format if run with the option `-C` and lists files one-per-line if run with the `-1` option.

This means that the invoked program has to process the options provided on the command line and store the result of that analysis into its state in some fashion.

Obviously, the behaviors of these particular options, `-C` and `-1`, cannot be put into effect at the same time. We say that there is a *dependency* between these two options.

As a consequence, `ls -C -1` results in the same behavior as if only `-1` was provided and vice versa, according to the principle *last one wins*. For such dependency we say that the options *override* one another. There are several kinds of dependencies with varying resolution strategies.

Another common pattern seen in command line arguments is option *arguments*. To give an example, the program `tail` prints the last  $x$  lines of a given input and is typically invoked with `tail -n x filename`. This means that option processing needs to make sure that  $x$  is indeed a number, and produce an error otherwise, and convert its textual representation into a numeric one. Often,

```

NAME:: encrypt
ARGS::
  input:{-i}:{file}:[mustexist, canbedir="false"]
  output:{-o}:{file}:[canbedir="false"]

```

**FORMAT**:: *input output*;

---

**Listing 1.** A simple CLOPS description of an encrypting program

an argument has a *default value*. If **tail** is invoked without the **-n** option, for instance, *x* is assumed to have value 10.

The set of options supported by a certain program represent its (textual) user interface. Hence, it is important for a program's options to be well-documented as a program's users cannot be expected to read its source code. In UNIX-like systems documentation is maintained in the database of *man(ual) pages*. As stated before, one of the objectives of this work is to keep a program's documentation consistent with its behavior.

### 3 CLOPS Exposure

Let us imagine we are writing a program which encrypts a given file. As we want to avoid problems with weirdly-named files (such as starting with the hyphen), we introduce one-argument options **-i filename** and **-o filename** to enable the user to specify the input and the output file, respectively.

Listing 1 is how we write it in CLOPS. The list under **ARGS** provides the possible command line options identified as *input* and *output*. For both of the options it specifies that they accept file-names as values. Additionally, the file given as *input* must exist and none of the files can be a directory.

The **FORMAT** is a regular expression that tells us that both *input* and *output* must be specified and *input* comes first.

The parser generated according to this CLOPS description will accept command lines such as **-i foo -o bar**, provided that the file **foo** exists.

After a while we get bored with this restrictive form and we decide to enable the user not to specify the output file as the program will derive the name from the name of the input file. Additionally, our colleague who has been using our tool is annoyed by all the debugging messages that are meaningless to him and we introduce the quiet mode. Since we have our first end-user, we add textual descriptions so we can generate simple documentation. The result can be seen in Listing 2. Note that for *quiet* we have not specified any type, which is the same as writing *quiet*:{-q}:{*boolean*}.

### 4 Concepts

The previous sections highlighted several common patterns that are seen in command line interfaces. Namely, we have different *types* of options, options *depend* on one another, and the *order* of options on the command line is important.

**NAME**:: *encrypt1***ARGS**::

```

:[ mustexist, canbedir="false"]:"The input file."
:[ canbedir="false"]:"The output file."
:"Turn the quiet mode on."

```

**FORMAT**:: *quiet? input output?*;**Listing 2.** Extended version of the encrypting program

What follows is a concept analysis of this domain. Each of these concepts is later reflected in the CLOPS DSL.

*Options.* The command line interface of a particular program focuses on a set of options. Each option has a *type* such as string, number, or file. For parsing purposes, each option is associated with a set of strings (possibly infinite) that determines how the option is specified on the command line. For instance, for the command `head`, any string of the form `-n NUMBER` specifies the number-of-lines option. For a given option, we will call these strings *match strings*. Last but not least, an option has a *documentation string* for end-user documentation purposes.

*Values.* Parsing a command line results in an internal representation of that particular command line. We will refer to this result as *option values*. As the program executes, these values control its behavior. Each option can be *set* to a value that must agree with the option's type. For example, the value of the number-of-lines seen above, is an integer.

*Format.* The ordering of options provided on the command line is often important. Different orderings may have different meanings to the program, and sometimes a different ordering of the same options is not valid. The *format* specifies all valid sequences of options.

We should note that the format not only determines whether a sequence of options is valid, but plays an important role in parsing. For instance, in the case of the version control system `svn`, one can write `svn add add`, where the first occurrence of the string `add` means “add a file to the repository” and the second occurrence means that the file to be added is called `add`. The parser is able to make this distinction as `svn`'s format specifies that a command must precede a filename and only one command use is permitted.

*Validity.* Only certain combinations of option values are *valid*. A function that determines whether a given combination of the values is valid or not is called a *validity function*. Validity functions are used to enforce constraints on options, for instance ensuring that two mutually exclusive options are not both enabled. Whenever an end-user specifies option values that fail validity testing, the command line parser produces an appropriate error message.

*Rules.* A change in the value of one option may affect values of other options. Such relationships are represented using *rules* that are triggered during parsing to change the value of other options. More specifically, a rule consists of: a *trigger*, telling us at which point of the format it may fire; a *guard*, a condition for firing; and an *effect*, the action executed when the guard holds. Rules are used to realize, among other features, overriding. For instance, in `1s`, the option `-1` triggers a rule that sets `-C` to *false*, and vice versa.

*Command line string.* An option is associated with a set of strings that state how that particular option is specified on the command line. However, each command line specifies *multiple* options.

In practice, a command line input is represented as a list of strings. A special character (denoted here by `_`) is appended to each of these strings and then the strings are concatenated to produce the *command line string*. This string will be used as the input to the parser.

Parsing partitions the given command line string into substrings, each corresponding to a different option. For instance, `1s` run with the command line string `-S_---format=long_` sorts files by size (`-S_`) and prints them in the long format (`--format=long_`).

Usually options come on the command line separately, therefore the match strings typically contain the delimiter character. For instance, the `1s` command's `--format` option *matches* on `--format=FORMATSPECIFIER_`, where the `FORMATSPECIFIER` is one of `across`, `commas`, `horizontal`, `long`, `single-column`, `verbose`, and `vertical`.

In some programs, the individual match strings do not have to be delimited. For instance, `ps xa` is equivalent to `ps x a`. This means that the option `x` matches on the character `x` *optionally* followed by `_`. These patterns motivated us to represent a command line input as a single string as we can treat such options uniformly.

## 5 DSL

We define the CLOPS domain specific language by giving its syntax and its semantics.

### 5.1 Syntax

Fig. 2 gives an overview of the CLOPS syntax<sup>1</sup>. The meta-symbols used in the right hand side of productions (`*`, `+`, `?`, `|`) have the usual meaning (zero or more times, one or more times, optional, alternative). Parentheses are also meta-symbols. (We omit the literal round parentheses from the grammar for clarity.) Terminals are typeset in **bold**. Nonterminals are typeset in roman if their productions are present, and are typeset in *italics* if their productions are missing.

---

<sup>1</sup> See also <http://clops.sourceforge.net/clops/grammar.html>

```

description → options format fly? override? validity?
options → OPTIONS:: opt+
format → FORMAT:: formatRegex
fly → FLY:: flyRule+
override → OVERRIDE:: overrideRule+
validity → VALIDITY:: validityRule+
opt → optName optAliases optType? optProps? optDoc?
formatRegex → optName | formatRegex (★ | + | ?) | formatRegex OR formatRegex
flyRule → optName (: guard)? → assignment+
overrideRule → guard → assignment+
validityRule → guard (: errorMsg)?
optName → id
optAliases → :{ regex (, regex)? }
optType → :{ id }
optProps → :[ propName = string (, propName = string)* ]
optDoc → string
guard → expr
assignment → optName := expr
errorMsg → expr
propName → id

```

**Fig. 2.** Syntax of the CLOPS DSL

There are four nonterminals not described in the grammar: *id*, *string*, *regex*, and *expr*. An *id* is a Java identifier<sup>2</sup>. A *string* is a Java string literal. A *regex* is a Java string literal *s* that does not cause an exception when used in an expression *Pattern.compile(s)*. An *expr* is a side-effect-free Java expression whose type is constrained, as discussed later.

A command line parser *description* in CLOPS consists of one or more *option* declarations, a *format* declaration, and, optionally, *rule* declarations. The informal meaning of each of these syntax sections is summarized through a series of examples below. A formal semantics is given in the next section.

**Example 1: A Prototypical Option Declaration** Here is an example of an option declaration:

```

tabsize :{ "-T", "--tabsize" }
: { int }
:[ minvalue="0" default="8" ]
:"assume given tab stops"
```

The *name* of the option is *tabsize*, it has *aliases* `"-T"` and `--tabsize`, its *type* is *int*, and its *documentation string* is `"assume given tab stops"`. Values are given to the option *properties* *minvalue* and *default*. The type *int* is **not** a Java type, but is an *option type*. The built-in types include *string*, *file*, and *boolean*, which is the default option type. The complete set of built-in option types is given later in Table 1, and Section 6 discusses how the type set is extended.

<sup>2</sup> The option type is allowed to contain `-`, unlike Java identifiers.

The *match strings* of the option `tabsize` are the words of the regular language specified by the expression `((-T) | (--tabsize)) [=] ([^=]*)+`. The prefix `((-T) | (--tabsize))` is built from the aliases, while the suffix `[=] ([^=]*)+` is automatically defined by the option type. The suffix is overridden for any option type using the property `suffixregexp`.

For another example, consider again the command `head`. This command permits the user to write `-n NUMBER` but also a shorter version `-NUMBER`. To specify this shorthand syntax, the suffix is `(\\d)+` (non-zero number of digits followed by the delimiter), and the prefix `-` (a hyphen).

**Example 2: Format Strings using Options.** A convention many UNIX tools follow is to consider everything on a command line that does begin with a hyphen character `(-)` as *not* being a filename. All file names that do start with a hyphen must appear after the special option `--`. This behavior is captured in the format string:

*(Flag OR File)\* (HyphenHyphen HyphenFile)\*?*

Here `Flag`, `File`, `HyphenHyphen`, and `HyphenFile` are option names. Section 6 introduces *option groups*, which make formats easier to write.

**Example 3: Fly Rules.** *Fly rules* are used, among other purposes, to obtain the ‘last wins’ behavior discussed earlier with regards to the `ls` command. Two fly rules that describe this interface are:

*HyphenOne*  $\rightarrow$  *HyphenC* := *false*  
*HyphenC*  $\rightarrow$  *HyphenOne* := *false*

Validity rules, such as the following, are used if an error is preferable:

*HyphenOne?*  $\&\&$  *HyphenC?*  $\rightarrow$  "Both -1 and -C were given."

For fly rules, a trigger is specified via an option name on the left of the arrow. For override rules, the trigger is described similarly. All validity functions are triggered when parsing finishes.

The fly rule examples above do not specify a guard, so the guard defaults to `true`. The validity rule example contains an explicit guard and illustrates some convenient syntactic sugar. The API generated for each option consists of a method to query its value and a method to query if it was set at all. The option name used in an expression is desugared into a call to get the option’s value, while the option name followed by a question mark is desugared into a call to check if the option was set. As expected, the (Java) type of the guard must be `boolean`. In the future we plan to allow JML [12] expressions to be used as guards.

The actions of fly rules are assignments. The left hand side of an assignment names an option, while the right hand side contains an expression. The expression must have a type convertible (according to Java rules) to the type of the option’s value. Override rules behave the same fashion, but validity rules are different. As seen in the validity rule example, the action of the rule is not written as an assignment, but is instead implicit. In this case, the assigned option has the type `string-list`, and is used to collect error messages.

The validity function is specified through validity rules. These rules are triggered after parsing has finished and, if their guard evaluates to **true**, the string on the righthand side is produced as an error.

## 5.2 Semantics

As previously mentioned, a CLOPS description is used to generate several artifacts, such as documentation and source code. The command line parser for the described options is the crux of these artifacts.

The mapping of the CLOPS description to a generated parser defines the DSL's semantics. The goal of this section is to provide an operational semantics of the command line parser for a given CLOPS description.

To be able to discuss a CLOPS description formally, we will assume that we have several mathematical objects that correspond to it.

**Preliminaries.** Assume the existence of a finite state automaton  $\mathcal{A}$ . Each transition of  $\mathcal{A}$  is labeled with an option  $o$  and a set of rules  $R$ . The notation  $s \xrightarrow{o, R} s'$  denotes that there is a transition from state  $s$  to state  $s'$  in  $\mathcal{A}$ , labeled with the option  $o$  and the set of rules  $R$ .

Legal sequences of options, as given by the CLOPS description, correspond to paths in the automaton that start in the starting state and end in an accepting state. The automaton is built in the usual way from the regular expression specified via the CLOPS format section.

A *rule* is a triple  $(g, o, e)$  containing a guard, an option, and a side-effect-free value expression. The intuition behind the rules is that all the rules labeling a transition are triggered whenever that transition is taken. When a rule  $(g, o, e)$  is triggered, its guard  $g$  is evaluated. If the guard evaluates to *true*, then the expression  $e$  is evaluated and the result is assigned to the value of the option  $o$ . We say that such a rule has *fired*.

The function *parse* computes a value for a given option from an input string. Hence, its type is  $\text{Option} \times \text{String} \rightarrow \text{Value}$ . The manner in which a String is converted to a Value is defined by the Option in question.

The function *match* tries to find a matching string of a given option among the prefixes of the command line string. Using the functional languages convention, its type is  $\text{Options} \times \text{String} \rightarrow \text{Maybe String}()$ . If the match is successful it returns the prefix on which the option matches, otherwise it returns **null**.

The *option store* is a partial function  $V$  from options to their values. The function is partial as when an option has not been set, it has no value.

**Details.** Let us now proceed with the definition of the command line parser based on these mathematical objects (the automaton  $\mathcal{A}$  and the functions *parse* and *match*).

The parsing of the command line string is a traversal of the automaton. At each stage of the parse the available transitions from the current automaton state represent the options that are valid for use at this point in the command line string. Each available option is checked to see if it matches at the current

position of the command line string. If no option matches, we produce an error. If an option matches we take the transition for that option, fire the rules that label the same transition if their guards evaluate to true, and move forward in the command line string by the match length.

The rest of the section defines this traversal formally.

Some new notation is convenient. The notation  $V[x \mapsto v]$  stands for a function that returns  $V(y)$  if  $x \neq y$  and  $v$  otherwise. For the side-effect-free expression  $e$  defined on the option values  $V$ , the notation  $e(V)$  represents the value to which  $e$  evaluates after substituting values for options.

The operational semantics of the command line parser is given as a transition function on the parser's states. A *state* of the parser is either an *error state* or a triple  $(c, V, s)$ , a command line string (see Section 4), a function from options to their values, and a state of the automaton  $\mathcal{A}$ , respectively.

For a state  $(c, V, s)$  we define the following two auxiliary sets:

1. Let *Legal* be the set of options labeling a transition from  $s$  in  $\mathcal{A}$

$$\text{Legal}(s) \equiv \{o \mid (\exists s')(s \xrightarrow{o, R} s')\}$$

2. Let *Matched* be a subset of *Legal* comprising options that match on  $c$ .

$$\text{Matched}(s, c) \equiv \{o \in \text{Legal}(s) \mid \text{match}(o, c) \neq \text{null}\}$$

The parser transition goes from the state  $(c, V, s)$  to the state  $(c', V', s')$  if *Matched* contains *exactly one* option  $o$  and  $s \xrightarrow{o, R} s'$ . The command line  $c'$  is obtained from  $c$  by stripping the prefix  $\text{Match}(o)$ .

The values of the options are updated by computing the function *Parse* for  $o$  and by triggering, in parallel, the rules labeling the transition being taken. This is formally written as follows.

Let  $V'' = V[o \mapsto \text{parse}(\text{match}(o, c))]$ . Then  $V'$  is defined as

$$V'(p) = \begin{cases} e(V) & \text{if there is a single rule } (g, p, e) \in R \text{ and } g(V'') \text{ holds, or} \\ V''(p) & \text{there is no such a rule.} \end{cases}$$

This covers the case when  $|\text{Matched}| = 1$  and there are no conflicting rules. However, if  $|\text{Matched}| = 1$ , and there are at least two rules whose guards evaluate to **true** that assign different values to the same option, then  $(c, V, s)$  goes to the error state.

If the set *Matched* is empty, there is no transition of  $\mathcal{A}$  to be taken. If it contains *more* than one option, the parsing is ambiguous. Thus, if  $|\text{Matched}| \neq 1$  we also go to the error state.

The computation of the command line parser starts in a state with the command line string to be processed, an option value function not defined anywhere (not assigning any values), and the start state of the automaton.

*Note:* In Future Work we suggest how one could check *statically* that the size of the set *Match* is at most one.

## 6 Implementation

Source code and builds of the implementation are freely available for download on the tool’s website<sup>3</sup>. There is also a user-oriented tutorial that details the first steps in running the tool and creating a CLOPS description. The tool has been implemented in Java and requires a Java Runtime Environment of at least version 5.0. Although we chose to write our implementation in Java, there is nothing to stop our approach from being implemented in other programming languages, as we do not rely on any language features that are unique to Java.

The overall functionality of the tool is as one might expect: it parses a specification provided in the CLOPS DSL and produces a command line parser with consistent documentation.

### 6.1 Option Types

Recall that when a user declares an option in the DSL, they must specify the type of the option. An option type determines the Java type used to represent the option’s value and, consequently, the parsing mechanism used for the corresponding option. Table 1 lists the built-in option types.

In order to extend the CLOPS DSL, new option types may be implemented and registered for use. An *option factory* defines the known set of options. Thus, to extend the CLOPS system and provide additional types, a developer extends the default option factory to make the system aware of new developer-provided types. The implementation of new option types can extend the built-in types to reuse and refine their functionality, or one may be written from the ground up, so long as the *Option* interface is satisfied.

While parsing each option declaration, the DSL parser queries the option factory for the corresponding option type. The option factory provides the necessary details of an option type: the class representing the option at runtime, as well as the Java type that is used for the value of the option.

The information provided by the option factory about each option’s type is used in the code generation phase. An interface is generated that provides access to the value for each option. For instance, for a string option, a *getter* method with return type *String* is created. For each option there is also a method with return type *boolean* that determines whether an option has been set at all.

Each option type has a mechanism for indicating the properties that it accepts. When parsing a given input specification, we can check that the option will actually accept all provided properties. Assuming there are no problems, code is generated that sets the option properties to their specified values during the initialization phase.

---

<sup>3</sup> <http://clops.sourceforge.net/>

**Table 1.** Built-in option types and properties. Abstract types are typeset in *italics*.  $T$  is the type of the concrete option’s value.

Name	Inherits from	Java type	Properties
<i>basic</i>	None	$T$	default[ $T$ ], suffixregexp[string]
boolean	basic	<i>boolean</i>	allowarg[boolean]
counted-boolean	basic	<i>int</i>	countstart[int], countmax[int], warnonexceedingmax[boolean], erroronexceedingmax[boolean]
string	basic	<i>String</i>	
string-regexp	string	<i>String</i>	regexp[string]
string-enum	string	<i>String</i>	choices[string], casesensitive[boolean]
int	basic	<i>int</i>	minvalue[int], maxvalue[int]
float	basic	<i>float</i>	minvalue[float], maxvalue[float]
file	basic	<i>File</i>	canexist[boolean], mustexist[boolean], canbedir[boolean], mustbedir[boolean], allowdash[boolean]
<i>list</i>	basic	<i>List</i> $\langle T \rangle$	allowmultiple[boolean], splitter[string]
string-list	list	<i>List</i> $\langle String \rangle$	
file-list	list	<i>List</i> $\langle File \rangle$	canexist[boolean], mustexist[boolean], canbedir[boolean], mustbedir[boolean], allowdash[boolean]

## 6.2 Option Groups

A tool’s format string sometimes grows quite long and is consequently difficult to understand and maintain due to its sheer number of options, independent of their dependencies. For this reason, we allow the options to be grouped.

An option group is defined by specifying a unique identifier paired with a list of options and/or other groups that are contained in the group. Then, when the identifier for a group is used in a format string, its definition is recursively expanded as a set of alternatives. Of course, these are hierarchical groups, and thus must be acyclic. We have found that appropriate use of option groupings make format strings much more concise, understandable, and more easily maintained.

There are many differing styles of documentation used to provide information to the end-user on the command line options available for a given tool. For this reason we have leveraged a powerful and mature open-source templating library, the Apache Velocity Project’s templating engine [15], to produce documentation from the information contained within a CLOPS description. We provide several built-in templates for documentation generation, and if a developer requires a different style for their documentation, they can modify an existing template, or create their own.

## 7 Experiments

In order to challenge CLOPS and to measure its effectiveness, several popular programs that have interesting option sets were described using the CLOPS system. Programs that are difficult or impossible to accurately describe using existing command line parsing tools without resorting to a large amount of custom parsing code were explicitly chosen.

Sources of information as to the semantics of a given tool's command line interface include manual pages, source code, command line help (e.g., *tool --help*), as well as trial-and-error with interesting option combinations.

We found describing these tools a useful exercise that led to many adjustments to our design and implementation. The fact that we were successfully able to describe their interfaces emphasizes the power and flexibility of our approach.

Additionally, even though all these programs are used by a large number of people, several inconsistencies were discovered in their documentation, and between their documentation and implementation.

Obviously, using the generated parser guarantees consistency between documentation and implementation. Moreover, the examples discussed in previous sections show that a systematic approach to recording options often highlights inconsistencies in their design, as well as their documentation.

### 7.1 ls

The program **ls** lists files found in the file-system, is frequently used and many options influence which information is displayed and in what format. The GNU implementation of **ls** comes with 56 options.

The options we have found the most interesting are options determining the format of the output. There is an option **format**, with the enumeration domain, where each of the enumeration values corresponds to a boolean option. The following snippet from the manual page shows how.

```
--format=WORD  across -x, commas -m, horizontal -x, long -l,
               single-column -1, verbose -l, vertical -C
```

This means that invoking **ls -C** should be the same as **ls --format=vertical**; **ls --format=horizontal** as **ls -x** etc.

This design is clearly a hotbed of inconsistencies and we have found one. The option **format** can have only one value at a time, i.e., if defined on the command line multiple times, the last one wins. According to the POSIX standard, however, the option **-1** should be assumed whenever **-1** is given<sup>4</sup>. As the GNU implementation follows the standard, the call **ls -1 -1** results in a different output than **ls --format=long --format=single-column**. The former corresponds to **ls -1** while the latter to **ls -1**. This means that the value **single-column** does not have the same effect as the option **-1** and the documentation is inaccurate.

---

<sup>4</sup> <http://www.opengroup.org/onlinepubs/000095399/utilities/ls.html>

Interestingly enough, the BSD implementation of `ls` does not follow the POSIX standard as `ls -1 -1` results in the equivalent of `ls -1`, i.e., the options `-1` and `-1` are regarded as distinct options which override one another.

We believe that situations similar to the `format` option arise quite often: as the program evolves, new options are added (like `-1`) and at some point there is too many that deal with a similar issue that an enumeration option is introduced to unify them. If that is the case, maintaining backward compatibility is difficult.

We should note that the enumeration `format` does appear only in the GNU implementation.

How does CLOPS help? Rules let us specify the equality between options. The following CLOPS description snippet relates `format`'s value "`commas`" and the boolean option `commas`.

```
commas {commas} -> format := {"commas"};
format {true} -> commas := {$(format).equals("commas")};
%$
```

The first identifier in the rules is the triggering option followed by a condition and an action. The first rule specifies that if `commas` is set to `true`, the `format`'s value is set to the value "`commas`". The second rule says that whenever the value of `format` changes, `commas` is set to `true` iff `format`'s value is "`commas`".

Note that this specification is not quite symmetric, as it does not specify what should happen when `commas` is set to `false`. In the current solution we disallow the user to set the option explicitly to `false` as the following would result in an inconsistent state (`-m` is a identifier of the option `commas`).

```
--format=commas -m=false
```

Alternatively we could unset the option `format` if the option representing the current value is set to false. Not enabling boolean options to be set to `false` is seen in the GNU implementation.

Writing up the rules in this fashion clarifies which options correspond to one another.

The source code processing the options in the GNU implementation has 327 non-blank lines of code. The CLOPS description *including* the documentation has 246 non-blank lines.

## 7.2 gzip

The program `gzip` compresses files using Lempel-Ziv coding. The most interesting part of the command line interface for this tool is the compression level. The compression level has a value in the range 1 to 9 and defaults to 6. The compression level is specified on the command line by prepending the number with hyphen (`-4` etc.) or by the predefined human-friendly aliases `--fast` for `-1` and `--best` for `-9`. In CLOPS, the compression level would be defined as an integer parameter:

```
compressionlevel :{ "-" }:{ int }:[ suffixregexp = "([1-9])\\00",
  maxvalue="9",minvalue="1",default="6" ]
```

In other words, we set the option name to `-` with a suffix of one digit followed by an option separator. Compression level aliases are defined as regular options with fly rules setting the compression level:

```
ARGS: compressionlevelbest:{ "--best" }
        compressionlevelfast :{ "--fast" }
FLY: compressionlevelbest -> compressionlevel:={9};
        compressionlevelfast -> compressionlevel:={1};
```

Although such a construction is simple and sufficient, there is a partial inconsistency when we use multiple options setting the compression level: for example, specifying `--best --fast` would correctly set the compression level to 1, but leave the `compressionlevelbest` parameter set. Similarly, we would expect option `compressionlevelbest` to be set to true when `-9` is provided. We can fix this with additional set/reset fly rules:

```
FLY:
compressionlevelbest -> compressionlevel:={9}, compressionlevelfast :={ false };
compressionlevelfast -> compressionlevel:={1}, compressionlevelbest :={ false };
compressionlevel ${( compressionlevel ).equals("9" )} ->
        compressionlevelfast :={ false }, compressionlevelbest :={ true };
compressionlevel ${( compressionlevel ).equals("1" )} ->
        compressionlevelfast :={ true }, compressionlevelbest :={ false };
```

The level of verbosity is increased with the verbose option `-v`. Using `-v` multiple times increases the verbosity level up to a maximum of three. In CLOPS we have a `counted-boolean` option that has type integer, that increases its count each time it is used. Gzip also has a quiet option `-q` that turns off some output. Using the quiet option resets the verbosity level to zero, and using the verbose option turns off quiet. To achieve this in CLOPS we can define verbose and quiet as the following:

```
ARGS:
verbose:{ "-v", "--verbose" }: { counted-boolean }:[ countmax="3", warnonexceedingmax ]
quiet:{ "-q", "--quiet" }
FLY:
quiet -> verbose:={0};
verbose -> quiet:={false};
```

## 8 Testimony

One of the authors (Julien Charles), who joined the project later than the other authors, applied CLOPS to three projects he has been working on. The following text summarizes the experience he gained. As each of the projects was in a different stage of maturity it demonstrates three different approaches to the use of CLOPS as well as a guide to get a better understanding of the tool.

All the tools mentioned in this section are part of the Mobius tool suite<sup>5</sup> and are written in Java. The task was to (1) replace an existing complete front-end

---

<sup>5</sup> Their source code can be obtained at <http://mobius.ucd.ie>.

**ARGS::**

...

*Dir*: {}:{*file*}:[*between*="", *mustExist*="true", *mustBeDir*="true"]  
 : "Specify directory in which Bico acts (there must be  
 only one directory, and this argument is mandatory)."

*Clazz*: {}:{*string-list*}:[*between*="", *argumentshape*="[a-zA-Z.]+"]  
 : "Generate also the file for the specified classes, bico  
 must be able to find them in its class path."

...

**FORMAT::**

*Help* | (*Dir* (*Misc* | *Type* | *Clazz*)\*);

---

**Listing 3.** Solution to the first problem

with CLOPS, (2) apply CLOPS to a tool with an unclearly defined list of options, and (3) apply the CLOPS methodology to a tool with no front-end yet.

### 8.1 Replacing an Already Complete Option Handler

The first encounter with CLOPS was to change the front-end of an already complete tool, Bico, a prelude generator for the proof assistant Coq. Its original architecture for handling options was the following:

- a Main class file containing methods to parse the options and set their values in a structure (174 lines), and
- an Option enum type containing the options' descriptions and their exact syntax (87 lines).

**Option conversion.** Since the options were well defined, most of them were easily translated to a CLOPS description.

The first difficulty we came across was with ambiguous arguments. Bico was taking as arguments a directory and an optional list of class file names. Both these arguments had no specific order on the command line and could interleave.

One method of dealing with this would have been to treat both of them as Strings and distinguish between them later in the Java code. This was not an ideal choice because CLOPS has mechanisms for recognizing files, as well as checking their existence and type, and we wanted to use these features.

The solution (Listing 3) we arrived at was to make the directory a mandatory first argument. The CLOPS specification mandated that the directory must exist. The class files were specified as a list of Strings, where the Strings have to be of a specific form (specified by the *argumentshape* property).

The second difficulty encountered involved the syntax of CLOPS and the authors of the pertaining code had to be consulted. In Bico there are two options that cannot be specified together. In CLOPS it is specified using a validity

rule, which produces an error if its condition evaluates to **true**. The solution to this requirement using a validity rule is given in Listing 4. Such rules are expressive but their syntax is obscure for an outside user and the documentation was insufficient at the time. Still, in the Bico case, it is fairly easy to understand: `$(Map) && $(List)` checks if the value of the options *Map* and *List* are **true** at the same time; `$(o)` expands to the value of *o* and `&&` is the Java and operator.

Note that the options *Map* and *List* default to **false**, hence they always do have a value.

#### ARGS::

```
...
Map: {"-m", "-map", "--map"}:[default="false"]
  : "Triggers the generation of the map implementation."
List: {"-l", "-list", "--list"}:[default="false"]
  : "Triggers the generation of the list implementation."
...
```

#### VALIDITY::

```
 {$(Map) && $(List)} : "The option '-m' and '-l' cannot be
                           specified at the same time."
```

**Listing 4.** Solution to the second problem

**Documentation.** The documentation templating was really useful. It facilitates the generation of correct and clear documentation of the options. Some standard templates are provided, but they can be easily modified slightly to fit the requirements of a particular user. For instance, the default templates show for each validity rule two things: the code generated for the test and the explanation. From the end-user point of view, the only item of relevance is the explanation. It was easy to edit the template and remove the code for the test, as the template uses the Velocity language. For many programs it is also necessary to add a copyright statement to all documentation. Again, this can be easily added by some simple template editing.

**Statistics.** The Main file that uses CLOPS is 134 lines long. The CLOPS description is 47 lines long, for a total of 181 lines of CLOPS-related code to write. Since the original argument-specific code for the program was 261 lines, there was therefore a reduction of 80 lines. In summary the use of CLOPS led to a code reduction of roughly 25% and adds information that was not originally present, such as proper descriptions of the different arguments.

## 8.2 Other CLOPS Usage Examples

**Clarifying arguments handling.** The second tool that we used CLOPS on was Mobius' DirectVCGen. It is a generator of verification conditions from annotated Java source files. It uses ESC/Java2 as a front-end to handle the annotations and the Java source. This case is interesting because it has options inherited from ESC/Java2 as well as several options of its own. ESC/Java2 has

over a hundred options and DirectVCGen actually only uses a few of them, but was not displaying an error message when an unsupported option was provided.

The CLOPS specification has allowed the identification of which arguments were used by DirectVCGen, and what their syntax was. Five arguments were identified, two specific to the DirectVCGen and three inherited from ESC/Java2. Here the aspect of CLOPS that was emphasized was its ability to help provide clear descriptions of the options as well as proper documentation.

The code to handle arguments in the original version was of 118 lines long, and in the new version with CLOPS is 132 lines long (101 lines of Java and 31 lines of CLOPS specification). The main gain here is having a proper description of the options which was not the case in the first version and thus making the program more robust.

**Developing with CLOPS.** The third tool we have used CLOPS on is Logic-Sync, a prelude consistency checker for ESC/Java2. The interest in using CLOPS right from the start of development was to be able to properly test the tool even at the early stages and also to be able to easily keep documentation up to date as the argument syntax evolves. Thanks to the experience acquired with the two other examples there were no specific difficulties in writing the CLOPS description. It was completed, together with the integration to the Java program, in less than one hour.

## 9 Related Concepts

A CLOPS-generated parser performs two prominent operations: 1) Processes the given sequence of strings (a command line) and returns a corresponding set of option values. 2) Decides whether the combination of option values is valid or not.

There is a variety of techniques aimed at describing sets of combinations of certain entities. A grammar describes a set of sequences of tokens; such sets are known as languages. A logic formula corresponds to a set of interpretations that satisfy the formula.

In CLOPS we can find reflections of both: the format string defines a regular language of options imposing restrictions on how options are sequenced on the command line; the validity function imposes restrictions on option values.

We should note, however, that grammars are typically used for quite a different purpose than the command line as they typically correspond to complex nested structures that drive compilers, interpreters, etc.

On the other hand, the elements on the command line live quite independently of one another and each corresponds to a wish of the user running the program. Some elements of the command line provide necessary information for the program to carry out its job, such as `cp fileSrc fileDest`—the copy program hardly can do anything if it does not know which files to copy or where. Some elements of the command line trigger a certain behavior of the program. For instance, the program `ls` will happily run with or without the argument

`-C` (column formatting) or the argument `-1` (one-per-line formatting). Each of these arguments corresponds to a *feature* of the program and the command line enables the user to express which features he requires.

In the program `ls`, the feature `-l` (long listing) is even so popular that users typically alias the invocation `ls -l` to `ll`. This provides us with an alternative view on the program: the original `ls` represents a family of programs and `ll` is a member of that family.

This brings us to another relevant domain *Software Product Lines (SPL)*, which studies families of programs [5]. In SPL an individual program is combination of certain features, and a family of programs is a set of feature combinations. The individual features and dependencies between them are captured in *feature model* [11].

In SPL terminology, the command line options represent variability of the program in question. In general, variability can be *bound* at different times, e.g., at *compile-time* or at *run-time* [13]. In our case, it is always bound at *run-time*. From the user perspective, however, there is no difference between an alias `ll` and a program called `ll` (the efficiency issue here is negligible).

Command line options are commonly regarded as something of small importance but we believe that is an underestimation. Deciding which options a particular program supports determines the scope of the variability in the program. The POSIX standard for common command line utilities is an evidence that it is not a trivial task.

In SPL, examining the scope of variability is known as *feature oriented domain analysis* (FODA) [11]. The variability in FODA is captured in the *feature model*, which is typically captured as a *feature diagram*. A feature model corresponds to a CLOPS description as both are variability models.

This discussion brings us back to grammars, in SPL the relation between grammars and feature models is not new [2]. Grammars are particularly useful in approaches where the order of features is important [3].

## 10 Related Work

There is a relatively large number of libraries for processing command line options. We identified three main groups of command line libraries.

The first group consists of libraries that follow the philosophy of the original Unix `getopt` function (Commons CLI [6] and JSAP [10]). These libraries usually provide only limited capabilities on top of basic `getopt`. Option values are queried via a *Map*-like interface and are usually treated as *Strings* or a particular type explicitly predefined by a getter method name. Libraries from the first group are ignorant of dependencies between options—dependencies must be explicitly handled by the tool developer.

Libraries in the second group recognize options via annotations of variables or methods in the source code (JewelCLI [9] and Args4j [1]). This approach has several advantages to the `getopt`-like option specification. Types are extracted directly from the source code and therefore the parser automatically checks

parameter syntax. Unfortunately, libraries in the second group also ignore option interdependencies.

The third group contains libraries that process options externally specified (JCommando [8]). An external specification is compiled into a library, much like in the CLOPS approach.

Table 2 summarizes and compares some of the best known of these libraries with CLOPS in several ways. Libraries are compared based upon their approach, features, and flexibility.

*Help extraction* is the ability to generate help documentation from the command line specification. *Incremental options* are options where multiple uses change the state of the option in some way, for example using `-v` more than once often increases the level of verbosity. *Relations among options* are mechanisms to specify dependencies between options (for instance, two options that cannot be used together, an option that can only be used in conjunction with another, etc.).

Some of the libraries allow *call-backs* during specific stages of a parse (e.g. after an option has been set). Thus, for some of the abilities detailed it is possible for the developer to write extra code to achieve the desired effect. In these cases the libraries do not facilitate the functionality, but do not prevent it either.

Most of the libraries support short and long variants of option aliases. Supporting a list of variants, rather than only a short/long pair, is rather exceptional (JewelCLI). Strict adherence to a finite enumeration of option-alias strings is a limitation in the flexibility of specifying option names. For example, the `tail` command prints a number of lines from the end of the input, where the exact number is specified as an option of the form `-NUMBER`. Even if a list of variants is allowed, specifying the aliases for this type of option would be too verbose to be an acceptable approach.

The leading hyphen is used almost exclusively as an option prefix. However, alternative prefixes are often attractive in some situations. For example, the `chmod` command uses both `-` and `+` as prefixes for options and effectively distinguishes when a particular feature is to be enabled or disabled. Another example is the tradition of some Windows command line tools to use slash (`/`) instead of hyphen (`-`).

Although it is often useful, the format of a command line is another feature omitted by most libraries. Many tools (including `svn`, `zip`, and `chown`) require a command name early in a command line in order to properly process later options. If there is no support for format within a library, as is nearly uniformly the case, such functionality must be hard-coded.

Dependencies among options are almost always ignored in current command line utilities and, therefore, manual testing the validity of option values is delegated to the user of the library. As mentioned in the Introduction, this leads to overly complicated and potentially buggy code.

As an exception, we have found the JCommando [8] library. JCommando introduces the notion of a *command*, which is similar to an option except that only one command can be set by a command line. The set method for the

**Table 2.** Features provided by the most popular CLO libraries

	CLOPS	JewelCLI	Commons CLI	Args4j	JCommando	JSAP	getopt
Version	1.0	0.54	1.1	2.0.1	1.01	2.1	
Specification	Separate DSL	Source Annotation	Source Builder p.	Source Annotation	Separate DSL/XML	Source Builder p.	Source Table
License	MIT	Apache 2.0	Apache 2.0	MIT	zlib/libpng	GPL	GNU
Option name variations	Regexp	List	Short/Long	No	Short/Long	Short/Long	Short/Long
Alternatives to leading hyphen	Yes	No	No	Yes	Command only	No	No
Option-argument format (e.g. <code>-X</code> , <code>-u=X</code> , <code>-uX</code> )	Yes	No	No	No	No	No	No
Dynamic option names (e.g. <code>gzip -1,-2,...,-9</code> )	Yes	No	No	No	No	No	No
Optional option-arguments	Yes	No	No	No	No	No	Yes
Default values for options	Yes	Yes	No	Yes	No	Yes	No
Types of option-arguments	Basic, special	Java prim., class, list	String	Basic, special	Basic	Java prim., List	char*
Option-argument validation	Type, attr, regexp	Type, regexp	None	Type	Type, range	Type	None
Command line format	Yes	No	No	No	No	No	No
Multiple use of one option	Yes	Last one	First one	Last one	Last one	First one	Call-backs
Help extraction	Yes	No	Yes	Yes	Yes	Yes	No
Fly rules	Rules	No	No	Call-backs	Call-backs	No	Call-backs
Relations among options	Yes	No	No	No	Limited	No	No
Option and arg. separator	Arbitrary	No	Yes, --	No	No	No	Yes, --
Option concatenation (old UNIX option style, e.g. <code>tar zxf</code> )	Yes	No	No	No	No	No	(hyphen required)
Incremental options	Yes	No	No	Call-backs	Call-backs	No	Call-backs

command is called last, after all option set methods. The fallback command (“commandless”) is used if no others are set by the user.

A command can specify what options can/should have been set for a valid parse. This is done using a boolean expression over option identifiers. Each identifier will evaluate to **true** or **false** at runtime according to whether that option was set or not. The boolean expression for the provided command is evaluated when that command is set, and if it evaluates to false an error is produced. This is a much simpler form of validity check than we allow. In particular, there is no way to relate options to each other outside the context of a command, nor is there a way to use option values in the test.

Surprisingly, many libraries do not provide a mechanism to enable a developer to specify, for example, filenames that start with a hyphen as any string starting with a hyphen is assumed to be an option. UNIX `getopt` solves this problem using the special string `--`, which is modeled in a CLOPS specification by the simple aforementioned format expression (see Section 5.1).

*Feature modeling* [11] has a goal similar to that of our DSL. A *feature model* explicitly captures the variability and commonality of a program [5]. In fact, one can imagine a program as a family of programs whose members correspond to the possible configurations of the feature model, as expressed via command line options. Whereas feature models target various types of variabilities at design- and compile-time, command line options represent variability resolved at the runtime.

## 11 Conclusions and Future Work

Many command line tools solve the command line parsing problem using custom code, sometimes relying on a little help from specialized libraries. “A DSL is viewed as the final and most mature phase of the evolution of object-oriented application frameworks,” according to Deursen et al [14]. CLOPS aims to succeed and supersede existing libraries in this regard. Combining an analysis of the set of existing solutions with the identification and refinement of the domain-specific concepts of command line options, a minimized conceptual framework is defined. Consequently, the syntax of the DSL is concise and self-documenting, reflecting exactly this conceptual design, which lends itself to a gradual learning curve. Additionally, while the DSL is simple enough to formalize and is easy to use, it is also powerful enough to cover all command line conventions of which we are aware.

The implementation is done classically, by writing a compiler that generates Java source code. From our experience of reimplementing the interface of standard UNIX tools and implementing the interface of some other tools, CLOPS increases a programmer’s productivity, though the quantification of such is the subject of further work.

Some aspects are painfully missing from the current implementation. One compelling aspect of many DSLs is that consistency checks are accomplished at a higher level of abstraction. One such consistency check for CLOPS is the ability to statically analyze a CLOPS description to guarantee that the generated command line parser compiles and does not fail in certain ways. For example, the

CLOPS system might perform type-checking instead of deferring such to the Java compiler. Creators of other DSLs found that early type-checking is sometimes more precise typechecking [14]. Another potential static check is ensuring regular expressions used to match options that can legally appear in the same parsing state are disjoint. This would ensure that the *Matched* set contains at most one element. (Of course, the parser would still fail at runtime when *Matched* is empty.) Finally, we require, but do not check, that the expressions used in rules do not have side-effects. Existing research on containing side-effects in languages like Java will prove useful [7,4].

Other parts are missing, but are less troublesome. A hassle that accompanies code generation is a more complicated build process. This annoyance is sometimes avoided by generating the parser at runtime. Of course, such an architecture change may hurt performance, but, for many command line parsing tools, we expect performance to be less important than ease of use.

In various flavors of UNIX there are many implementations of common command line POSIX utilities such as `chmod` and `ls`. After describing a single command line interface for a given tool across a suite of implementations one might generate command line strings automatically for system-level testing. Such work has been done in a limited form already here at UCD for the OpenCVS project.

Finally, some of this work is applicable in the context of tools with graphical, rather than textual, interfaces. GUI programs often store their settings in a preferences store that is read upon start-up, and many such programs provide a ‘wizard’ (e.g., a preferences pane) to populate the settings file. While it requires is no stretch of the imagination to envisage the CLOPS framework generating the wizards and the parser for such GUI programs, important details need to be clarified before this step is made.

## Acknowledgments

This work is funded, in part, by Science Foundation Ireland under grant number 03/CE2/I303-1 to “Lero: The Irish Software Engineering Research Centre”, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project, and by an EMBARK Scholarship from the Irish Research Council in Science, Engineering and Technology. Viliam Holub is supported by the IRCSET Embark Postdoctoral Fellowship Scheme. The article contains only the authors’ views and the Community is not liable for any use that may be made of the information therein.

## References

1. Args4j, <http://args4j.dev.java.net/>
2. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

3. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology* (1992)
4. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. *SIGPLAN Not.* 33(10), 48–64 (1998)
5. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley Publishing Company, Reading (2002)
6. The Apache Commons CLI library, <http://commons.apache.org/cli/>
7. Darvas, A., Muller, P.: Reasoning about method calls in JML specifications. In: *Formal Techniques for Java-like Programs* (2005)
8. JCommando: Java command-line parser, <http://jcommando.sourceforge.net/>
9. JewelCLI, <http://jewelcli.sourceforge.net/>
10. JSAP: The Java Simple Argument Parser, <http://www.martiansoftware.com/jsap/>
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.: Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University (November 1990)
12. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In: *Behavioral Specifications of Business and Systems*, pp. 175–188. Kluwer Academic Publishing, Dordrecht (1999)
13. Svahnberg, M., Van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. *Software-Practice and Experience* 35(8), 705–754 (2005)
14. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *SIGPLAN Not.* 35(6), 26–36 (2000)
15. The Apache Velocity Project, <http://velocity.apache.org/>

# Nettle: A Language for Configuring Routing Networks

Andreas Voellmy and Paul Hudak

Yale University,  
Department of Computer Science  
`andreas.voellmy@yale.edu, paul.hudak@yale.edu`

**Abstract.** *Interdomain routing* is the task of establishing connectivity among the independently administered networks (called *autonomous systems*) that constitute the Internet. The protocol used for this task is the *Border Gateway Protocol* (BGP) [1], which allows autonomous systems to independently define their own route preferences and route advertisement policies. By careful design of these BGP policies, autonomous systems can achieve a variety of objectives.

Currently available configuration and policy languages are low-level and provide only a few basic constructs for abstraction, thus preventing network operators from expressing their intentions naturally.

To alleviate this problem, we have designed *Nettle*, a domain-specific embedded language (DSEL) for configuring BGP networks, using Haskell [3] as the host language. The embedding in Haskell gives users comprehensive abstraction and calculation constructs, allowing them to clearly describe the ideas generating their BGP policies and router configurations. Furthermore, unlike previous router configuration and policy languages, Nettle allows users to both specify BGP policies at an abstract, network-wide level, and specify vendor-specific router details in a single uniform language.

We have built a compiler that translates Nettle programs into configuration scripts for XORP [4] routers and a simulator that allows operators to test their network configurations before deployment.

## 1 Introduction

Given the importance of the Internet, one would expect that it is well designed, that the principles upon which it operates are well understood, and that failures are due primarily to hardware crashes, network cable interruptions, and other mechanical or electrical failures. Unfortunately, and perhaps surprisingly, this is not the case.

The protocols that control the Internet were not designed for the way in which it is used today, leading to non-standard usage and ad hoc extensions. Some of the most fundamental principles upon which the Internet operates are poorly understood, and recent attempts to understand them better have revealed that in fact the protocols themselves permit network instabilities and outages

[5]. Furthermore, most of these problems are manifested not through hardware failures, but through software bugs, logical errors, or malicious intrusions.

The Internet has grown dramatically in size since its inception. The lack of centralized control is what has enabled this growth, and as a result the Internet is essentially owned and operated by thousands of different organizations. Most of these organizations (such as the many ISPs) share common goals – for example they all want messages to reach their destinations, and they do not want message content compromised. Because of this alignment of interests, the Internet, for the most part, works.

But this lack of centralized control is also a source of problems. For starters, the goals of the various entities that control the Internet are not always completely aligned. Different business models, economic incentives, engineering decisions, desired traffic patterns, and so on, can lead to conflicting interests in the control of the Internet. These conflicts in turn can lead to oscillations, deadlocks, and other kinds of instabilities in the forwarding of messages [5].

Furthermore, the standard protocols have become increasingly complex (mostly to accommodate the ever-increasing range of interests) and the effect of altering various parameters is not completely understood. Even detecting errors that lead to anomalies in Internet traffic is difficult, as they are often intermittent and difficult to localize.

To make matters worse, the scripting languages used to specify the behaviors of all major routers are all different, and are all astonishingly low level. Even if one knows exactly what behavior is desired, errors in configuring the network are quite common. The scripting languages have few useful constraints (such as a type system), no abstraction mechanisms, poorly understood semantics, and mediocre tools for development and testing. Indeed, it has been estimated that almost half of all network outages are a result of network misconfigurations [6].

## 1.1 A Language-Centric Solution

The goal of our research is to use modern programming language ideas to help ameliorate many of the aforementioned problems. We call our language and accompanying tools *Nettle*, and we have implemented it as a domain-specific embedded language [7] [8], using Haskell as a host. Nettle users express their router configurations in a high-level, declarative way, and the Nettle compiler generates the low-level scripting code for specific back-end routers (currently XORP). This allows network operators to specify goals without becoming mired in implementation details.

Specifically, Nettle offers the following advantages:

1. Nettle works with existing protocols and routing infrastructure; no changes to the routers themselves are needed.
2. Nettle takes a local-network-wide view, allowing operators to specify the entire local network routing configuration as one cohesive policy.
3. Nettle separates implementation- and hardware-specific concerns from logical routing concerns, resulting in a platform-neutral approach in which a single policy specification can be given for a heterogenous collection of routers.

4. Nettle permits multiple routing policies for the same network, and provides a flexible and safe way to merge their behaviors.
5. The embedding of Nettle in Haskell allows users to safely define their own configuration abstractions and invent new configuration methods.
6. By defining high-level policies in Nettle, it is possible to prevent global anomalies (such as route oscillations) if enough nodes on the Internet adopt the same strategy.

We have implemented a compiler for Nettle that generates configuration scripts for the XORP [4] open-source router platform. We have also implemented a BGP network simulator that allows Nettle users to explore the behavior of their BGP configurations under a variety of user-specified scenarios.

In the remainder of this paper we first give a brief introduction to network routing and BGP. We then introduce Nettle's salient features through a series of small examples and describe a complete network configuration. We then show how Nettle can be used to safely capture complex policy patterns as policy-generating functions. Finally, we discuss implementation issues and briefly compare Nettle with XORP and RPSL.

## 2 Introduction to Networks, Routing and BGP

A communication network is a distributed system that supports point-to-point messaging among its nodes. The behavior of such a network is typically decomposed into *forwarding* and *routing* processes. By *forwarding process (routing process)*, we mean the collective forwarding (routing) behavior of all the nodes, rather than the behavior of a single process. Forwarding consists of sending messages between nodes, whereas routing consists of establishing paths between nodes. The relationship between forwarding and routing is that the forwarding process uses the paths established by the routing process. The routing process is dynamic – the paths between nodes may change over time. This is essential in networks in which communication links and nodes can fail. In such situations, the routing process typically responds to events by recomputing paths to avoid failed links or nodes.

In networks that are administered by a single organization, a routing process is typically used to compute least-cost paths among nodes, where cost is often a communication-link metric incorporating information such as bandwidth, reliability, distance or other qualities. On the other hand, networks such as the Internet, which consist of many independently administered sub-networks, also need to support point-to-point messaging. Such networks do not use a single routing process for a variety of reasons. One reason is that different local concerns may result in the use of different communication cost metrics. Another reason is that while networks need to establish some global connectivity, for security reasons they often wish to restrict visibility of their local internal structure.

Instead, such networks typically deploy their own *intra*-network routing process for their own nodes, and a single *inter*-network routing process to communicate with the rest of the network. The inter-network routing process is often

based on a high-level *policy* that is not necessarily based on a least-cost algorithm – it might also factor in economic incentives, traffic engineering, security concerns, and so on. Another key aspect of the inter-domain routing process is that it typically involves an exchange of messages that announce the availability of routes, and these announcements carry certain attributes that allow networks to make policy-based decisions about which routes to use.

The Internet community has established several standard protocols to implement routing processes. In Internet parlance, an intra-network routing process is called an *Interior Gateway Protocol (IGP)* and two commonly used IGP protocols are *Open Shortest Path First (OSPF)* and *Routing Information Protocol (RIP)*. The independently administered networks on the Internet are called *Autonomous Systems (ASes)* and *domains* interchangeably. The interdomain routing protocol in use in the Internet is *Border Gateway Protocol (BGP)* [1]. Autonomous systems are assigned globally unique 32-bit *Autonomous System Numbers (ASNs)* by the *Internet Assigned Numbers Authority (IANA)*. An *Internet Service Provider (ISP)* is an AS whose primary purpose is to provide Internet access to other ASes in exchange for payment.

## 2.1 BGP

BGP [1] is our main focus in this paper. BGP is essentially a routing process that is parameterized by a policy, just as a higher-order function is parameterized by a functional argument. In order to understand BGP policy, we need to understand some basic networking concepts.

An IP address is a 32 bit value, commonly written as four bytes (or *octets*, in Internet parlance), as in *a.b.c.d*. A subset of addresses, called an *address prefix*, is denoted by an address followed by a number between 0 and 32, as in *a.b.c.d/e*. A prefix *a.b.c.d/e* corresponds to all addresses whose leftmost *e* bits match the leftmost *e* bits of *a.b.c.d*. More formally, we can write a prefix *a.b.c.d/e* and address *w.x.y.z* as sequences of bits  $a_1a_2\dots a_e$  and  $w_1w_2\dots w_{32}$ ; the address  $w_1w_2\dots w_{32}$  is contained in prefix  $a_1a_2\dots a_e$  if  $a_i = w_i$  for all  $1 \leq i \leq e$ .

Nodes running BGP communicate by announcing routes to each other. These route announcements carry, among other data, the following information:

1. An address prefix, representing all addresses reachable using this route;
2. A sequence of AS numbers, called the *AS path*, which represents the ASes that messages will traverse when forwarded along this route;
3. *Community attributes*, 32-bit values that act as data used for ad-hoc communication between different ASes.

BGP allows networks to assign numeric *local preference* attributes to routes. These local preference values are then used as the primary criterion in the *BGP decision process*, which is used to choose the best route among several to the same prefix. The BGP decision process is a prefix-wise lexicographic ordering of routes based on their attributes, in the following order:

1. local preference
2. shortest AS path length

3. lowest origin type
4. lowest MED
5. eBGP-learned over iBGP-learned
6. lowest IGP cost to border router
7. lowest router ID (used as a tie-breaker)

The BGP decision process roughly implements shortest AS-path routing, while allowing networks to override this behavior by assigning non-default local preferences to routes. This allows networks to implement a variety of policies. For example, an operator may decide not to choose any paths that traverse a particular set of autonomous systems. Or they may prefer routes through peer A, except when peer B offers a route that also goes through peer C. Or they may want routes to some subset of destinations, such as universities, to go through one peer, while commercial traffic goes through another peer.

BGP routers usually also support *filter policies* and *export modifiers*. Filters are used to remove some received routes from consideration by the decision process or to prevent some routes from being announced to peers. Export modifiers are used to alter the attributes of routes that are announced to peers. For example, a network may add its AS number several times to the AS path of a route so that it appears longer.

We can organize these policies into two sorts, *usage policy* and *advertising policy*. Usage policy governs which routes are considered and how they are chosen for use by the network, while advertising policy governs which routes are offered to neighboring networks and with what attributes those routes are advertised. The usage policy determines how traffic flows out of an autonomous system, while advertising policy influences (but does not determine) how traffic will flow into the autonomous system. We will call a filter used to eliminate routes from consideration in the decision process *use filters* and filters used to prevent route announcements *ad filters*. If we represent the set of neighboring BGP routers as  $P$ , we can write the types of the policy functions as follows:

- $useFilter :: P \times Route \rightarrow Bool$
- $preference :: Route \rightarrow \mathbb{N}$
- $adFilter :: P \times Route \rightarrow Bool$
- $adModifier :: P \times Route \rightarrow Route$

**Community Attributes.** Community attributes are a crucial tool allowing networks to communicate about routes. Community attributes are 32-bit numeric values, commonly written as  $a : b$ , where  $a, b$  are 16-bit values. While there are a few community values that have an established meaning, such as the NO\_EXPORT community value, most community values are available for ad hoc use between BGP neighbors. For example, an ISP may allow its customers to dynamically alter the preference levels of routes announced by the customers – these preferences are encoded as community attributes.

**Internal BGP.** The BGP protocol has two sub-protocols: IBGP (Internal BGP) and EBGP (External BGP). As the names suggest, EBGP governs the

interaction with external peers, while IBGP governs the interaction with internal BGP-speaking peers. IBGP's primary purpose is to ensure that all the BGP routers within an AS know the same routes and that they make decisions consistently.

## 2.2 Protocol Interaction

As mentioned earlier, an AS runs an IGP to establish routes to internal nodes and BGP to establish routes to the wider network. Routers running BGP typically run the IGP protocol also. At these routers the IGP and BGP protocols may interact, and this interaction is called *route redistribution*. Typically, a BGP router will inject some routes into the IGP protocol so that these routes propagate to the non-BGP speaking routers in the network. To avoid overwhelming and potentially destabilizing the IGP, BGP routers are usually configured to advertise only a few aggregated routes, such as the default route 0.0.0.0/0.

## 3 Nettle Tutorial

In this section we introduce the salient features of the Nettle language, illustrating their use through small examples.

**Preliminaries.** Since Nettle is embedded in Haskell, it inherits the syntax of Haskell, and gives the user access to all of the power of Haskell. We assume that the reader is familiar with basic Haskell syntax.

Throughout this paper we have typeset some Nettle operators with more pleasing symbols. In particular,  $\wedge$  is written as  $\wedge$  in Haskell,  $\vee$  as  $\vee$ ,  $\triangleright$  as  $\gg$ ,  $\parallel$  as  $\parallel$ , and  $\parallel\parallel$  as  $\parallel\parallel$ .

An *IP address* such as 172.160.27.15 is written in Nettle as *address* 172 160 27 15. An address prefix such as 172.160.27.15/16 is written in Nettle as *address* 172 160 27 15  $\parallel$  16, i.e. the  $\parallel$  operator takes an address and a length and returns a prefix.

Network operators usually write community attributes as  $x : y$ , where  $x$  is the higher-order 16 bits and  $y$  is the lower-order 16 bits of the data field. By convention,  $x$  is used to encode the local AS number. In Nettle we represent a community attribute as  $x :: y$ .

### 3.1 Routing Networks

A routing network consists of a collection of routers participating in one or more routing protocols. For example, typical networks run BGP on border routers; some internal routing protocol, such as OSPF or RIP, on all routers; and additionally configure some routes statically, that is, not learned through a dynamic routing protocol. Currently Nettle supports only two routing protocols, *BGP* and *Static*.

At the highest level, a Nettle program describing a BGP policy has the form:

*nettleProg* = *routingNetwork* *bgpNet* *staticNet* *redistPolicy*

where

1. *bgpNet* is a description of the BGP network.
2. *staticNet* is a description of the static protocol.
3. *redistPolicy* describes how the BGP and static protocols interact.

*bgpNet* is typically defined as follows:

*bgpNet* = *bgpNetwork* *asNum* *bgpConns* *prefs* *usefilter* *adfilter* *admodifier*

where:

1. *asNum* is the AS number of the AS whose routing is being defined.
2. *bgpConns* is a list of connections.
3. *prefs* is an assignment of preference levels, i.e. integers, to routes and is used to determine which routes to use.
4. Given a set of routes received by a router, *usefilter* discards ones it doesn't want.
5. Given a set of routes known by a router, *adfilter* discards those that it does not wish to advertise to neighbors.
6. *admodifier* is a function which may change attributes of a route as it is exported; this used to influence neighbors routing decisions and to ultimately influence incoming traffic.

In the following subsections we will see how each of these constituent pieces is generated.

**Routers.** Several aspects of routing policy, such as static route announcements and route redistribution, are specified in terms of particular routers in the network. Additionally, the Nettle compiler will need router-specific details, such as hardware configurations, in order to compile a Nettle program. Nettle programs therefore need to know the set of routers to be configured.

In order to maintain modularity, Nettle policies are written in terms of an abstract router type that router-specific data types implement. For example, we can declare an abstract router *r1* , implemented by a XORP router, with the following code:

```
r1 = router r1xorp
where r1xorp      = xorpRouter xorpBgpId xorpInterfaces
      xorpInterfaces = [ifaceEth0]
      ifaceEth0     = xorpInterface "eth0" "data" [vifEth0Eth0]
      vifEth0Eth0   = let block   = address 200 200 200 2 // 30
                      bcastAddr = address 200 200 200 3
                      in vif "eth0" (vifAddrs (vifAddr block bcastAddr))
```

Here the *router* function hides the specific type of router. Policy is then written in terms of the abstract router type, as will be shown in the following sections. This design separates router-specific details from abstract policy and allows configurations to be written modularly. In particular, it allows users to change the router platform of a router without changing any routing policy. Furthermore, this design allows Nettle to support configurations of a heterogeneous collection of router types within a single language.

In the following examples, we assume that we have defined routers *r1*, *r2*, and *r3*.

**Static Routing.** Static routes are specified simply by describing the address prefix, the next-hop router, and the router that knows of the route. For example, the following describes a static routing configuration with three routes, known at two routers:

```
staticConfig [
  staticRoute r1 (address 172 160 0 0 // 16) (address 63 50 128 1),
  staticRoute r1 (address 218 115 0 0 // 16) (address 63 50 128 2),
  staticRoute r2 (address 172 160 0 0 // 16) (address 63 50 128 1)]
```

We note that a standard Haskell **let** expression can be used to rewrite this as:

```
let ip1 = address 172 160 0 0 // 16
  ip2 = address 218 115 0 0 // 16
  ip n = address 63 50 128 n
in staticConfig [staticRoute r1 ip1 (ip 1),
  staticRoute r1 ip2 (ip 2),
  staticRoute r2 ip1 (ip 1)]
```

which is arguably easier to read. Even this simple kind of abstraction is not available in most (if any) router scripting languages.

**BGP Connections.** A connection specification consists of a set of *BGPConnection* values. For example, the following two connections describe external and internal connections, respectively:

```
conn1 = externalConn r1 (address 100 100 1 0) (address 100 100 1 1) 3400
conn2 = internalConn r1 (address 130 0 1 4) r3 (address 130 0 1 6)
```

The first line declares that router *r1* has a BGP connection with an external router from AS 3400 and that the address for *r1* on this connection is 100.100.1.0 and the peer's address is 100.100.1.1. The second line declares that *r1* and *r3* are connected via IBGP using addresses 130.0.1.4 for *r1* and 130.0.1.6 *r3*.

**Subsets of Routes.** At the core of Nettle lies a small language of route predicates for specifying sets of routes. This language is used to apply different policies to different sets of routes. The language, which is designed with an eye towards implementation on available router platforms, consists of a set of atomic

predicates; a conjunction operator,  $\wedge$ , denoting the intersection of two subsets of routes; and a disjunction operator,  $\vee$ , denoting the union of two subsets of routes. For example,

$$\text{nextHopEq} (\text{address } 128\ 32\ 60\ 1) \vee \text{taggedWith} (5000 :: 120)$$

denotes the set of routes whose next hop routers have address 128.32.60.1 or those which are tagged with community attribute 5000:120.

Another example is:

$$\begin{aligned} \text{destInSet} [\text{address } 128\ 32\ 60\ 0 // 24, \text{address } 63\ 100\ 0\ 0 // 16] \\ \wedge \text{taggedWithAnyOf} [5000 :: 120, 7500 :: 101] \end{aligned}$$

which denotes the set of routes for destination prefixes 128.32.60.0/24 or 63.100.0.0/16 and which are marked with one or more of community attributes 5000:120 or 7500:101.

The *asSeqIn* predicate allows users to specify complex conditions on BGP routes, as in the following example which requires that routes originate at AS 7000, pass through any number of networks, and finally pass through either AS 3370 or 4010 once followed by AS 6500:<sup>1</sup>

$$\begin{aligned} \text{asSeqIn} (\text{repeat} (i\ 7000) \triangleright \text{repeat any} \triangleright \\ (i\ 3370 \parallel i\ 4010) \triangleright \text{repeat} (i\ 6500)) \end{aligned}$$

The full collection of predicates is shown in Figure 1.

Using Haskell's standard abstraction mechanisms, this predicate language allows users to write new predicates. For example, we can define a predicate that matches a path exactly, while allowing for prepending, as follows:

$$\begin{aligned} \text{pathIs} &:: [\text{ASNumber}] \rightarrow \text{RoutePredicate BGPT} \\ \text{pathIs} \ xs &= \text{asSeqIn} \$ \text{foldr} (\lambda a\ r \rightarrow \text{repeat} (i\ a) \triangleright r) \ \text{empty} \ xs \end{aligned}$$

It is impossible to express this kind of predicate in any router scripting language that we are aware of.

**Usage and Advertising Filters.** Filters play a role in both usage and advertising policy; a usage filter prevents some routes from being considered for use, while an advertising filter prevents some routes from being advertised to peers. Nettle allows users to declare filters based on predicates using the *reject* function. For example,

$$\text{reject} (\text{destEq} (\text{address } 128\ 32\ 0\ 0 // 16))$$

is a filter that rejects routes to destinations 128.32.0.0/16.

Nettle also allows users to specify connection-dependent filters, i.e. maps associating BGP connections with usage (or advertising) filters. For example, the

---

<sup>1</sup> In this paper *repeat* is the kleene star operation on regular expressions, not the *repeat* function in Haskell's *Prelude* module.

<i>destEq</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>AddressPrefix</i>	$\rightarrow$ <i>RoutePred p</i>
<i>destNotEq</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>AddressPrefix</i>	$\rightarrow$ <i>RoutePred p</i>
<i>destInRange</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>AddressPrefix</i>	$\rightarrow$ <i>RoutePred p</i>
<i>destInSet</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>[AddressPrefix]</i>	$\rightarrow$ <i>RoutePred p</i>
<i>nextHopEq</i>	:: <i>Address</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>nextHopInRange</i>	:: <i>Address</i> $\rightarrow$ <i>Address</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>nextHopInSet</i>	:: <i>[Address]</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>asSeqIn</i>	:: <i>RegExp ASNumber</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>taggedWith</i>	:: <i>Community</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>taggedWithAny Of</i>	:: <i>[Community]</i>	$\rightarrow$ <i>RoutePred BGPT</i>
<i>all</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>RoutePred p</i>	
<i>none</i>	:: <i>Protocol p</i> $\Rightarrow$ <i>RoutePred p</i>	
$(\wedge)$	:: <i>Protocol p</i> $\Rightarrow$ <i>RoutePred p</i> $\rightarrow$ <i>RoutePred p</i> $\rightarrow$ <i>RoutePred p</i>	
$(\vee)$	:: <i>Protocol p</i> $\Rightarrow$ <i>RoutePred p</i> $\rightarrow$ <i>RoutePred p</i> $\rightarrow$ <i>RoutePred p</i>	

**Fig. 1.** The language of route predicates

following usage filter rejects routes learned over connection *c1* which either (1) are for block 128.32.0.0/16 and are tagged with community 5000:120, or (2) are tagged with community 12345:100, while for other connections it rejects only routes that pass through network 7000:

```
usefilter c =
  if c ≡ c1
  then reject ((destEq (address 128 32 0 0 // 16)
                ∧ taggedWith (5000 :: 120)) ∨ taggedWith (12345 :: 100))
  else reject (asSeqIn (repeat any > repeat (i 7000) > repeat any))
```

**Route Preferences.** The central part of BGP usage policy consists of route preferences. In Nettle, users specify route preferences by giving a numerical rank, with higher being more preferred, to sets of routes, which are specified using route predicates. To do this we can use the *route conditional* expressions *cond* and *always*. For example, the expression<sup>2</sup>

```
cond (taggedWith (5000 :: 120)) 120
  $ cond (taggedWith (5000 :: 100)) 100
  $ cond (taggedWith (5000 :: 80)) 80
  $ always 100
```

ranks routes with community 5000:120 at rank 120, routes not tagged with 5000:120 but tagged with 5000:100 at rank 100, and so on, until it finally matches all remaining routes with rank 100.

<sup>2</sup> Note that  $f\$x = f x$ . Using this function allows us to avoid writing some parentheses; for example  $g \$ f \$ x = g (f x)$ .

**Route Modifiers and Guarded Route Modifiers.** The central part of BGP advertising policy consists in modifying route attributes as routes are advertised to peers, so as to communicate intentions about routes, or influence how others prefer routes. Intentions about routes can be communicated through the use of community attributes, while AS path prepending can influence peers' decisions by increasing the apparent path length of the route. Nettle provides a small language for expressing route modifiers. For example, the modifier:

*tag* (5000 ::: 130)

denotes a function mapping a BGP route to the same BGP route tagged with community 5000:130, while the modifier:

*prepend* 65000

represents a function mapping a BGP route to the same BGP route with the AS number 65000 prepended to the AS path attribute. Two route modifiers *f* and *g* can be combined as  $f \triangleright g$ , which denotes the reverse composition of the modifiers denoted by *f* and *g*. For example,

*tag* (5000 ::: 130)  $\triangleright$  *prepend* 65000

represents the function which associates a BGP route with the same route tagged with community 5000:130 and with AS path prepended with AS number 65000. Finally, route modifiers *ident* and *unTag c* leave a route unchanged and remove a community value *c* from a route, respectively.

As with route predicates, route modifiers allow us to define new modifiers and use them wherever route modifiers are required. For example, we can define a modifier which prepends an AS number a specified number of times:

*prepends n asn* = *foldr* (  $\triangleright$  ) *ident* \$ *replicate n (prepend asn)*

Route modifiers can be combined with conditional statements to describe *guarded route modifiers*. For example,

*cond* (taggedWith (5000 ::: 120)) (*prepend* 65000) (*always* (tag 1000 ::: 99))

is a guarded route modifier that prepends the AS number 65000 to a route only if that route is tagged with community value 5000:120, and otherwise tags the route with community 1000:99.

Nettle allows route modifiers to be specified per connection and represents these as Haskell functions. For example,

<i>adMod c</i>	$ $	<i>c</i> $\equiv$ <i>c1</i>	$=$	<i>always (prepend 65000)</i>
	$ $	<i>c</i> $\equiv$ <i>c2</i>	$=$	<i>always (prepends 2 65000 <math>\triangleright</math> tag (65000 ::: 120))</i>
	$ $	<i>otherwise</i>	$=$	<i>always ident</i>

**Connecting Protocols.** Nettle currently provides a simple mechanism for connecting protocols, a process also known as *route redistribution*, in which a subset of routes from one protocol are introduced into another protocol at a particular router. For example,

```
redistAt r1 (destEq (address 80 10 32 0 // 24)) StaticProtocol BGPProtocol
```

indicates that routes with destination 80.10.32.0/24 from the static protocol will be redistributed into the BGP protocol at router *r1*. Multiple redistribution statements can be combined with *redists*:

```
let p1 = redistAt r1 (destEq (address 80 10 32 0 // 24))
  StaticProtocol BGPProtocol
p2 = redistAt r2 (destEq (address 100 10 20 0 // 24))
  StaticProtocol BGPProtocol
in redists [p1, p2]
```

### 3.2 Compiling to Routers

By combining routing policy with information about the specific routers in the network, we can compile appropriate configuration files for each router. We can compile a configuration for a router in our network with the command *compile*, which returns a *String* value, as in the following example, which gives the configuration file for *r1*:

```
compile rNet r1
```

Although Nettle currently only provides a compiler for XORP routers, we ultimately intend the Nettle compiler to support all major router platforms.

### 3.3 Simulation

The Nettle library also provides a simple BGP simulator, whose implementation is based on the formal BGP model of Griffin et al [9]. The simulator provides a coarse approximation of BGP. In particular, it ignores timing details, and rather calculates routing tables for each BGP node in a network in rounds. In each round BGP nodes apply export policy to their best routes, exporting to neighboring nodes, and then apply usage policy to their newly received and currently known routes.

A significant limitation of the simulator is that it does not model IBGP and that it only models one connection per pair of peers. Therefore in order to simulate our Nettle BGP networks with this simulator, we need to approximate our network with a single BGP node, preserving the connections and policies of the original Nettle network as much as possible. Due to these limitations, the simulator is not yet useful for simulating the behavior of most large networks. However, for small networks which have only one connection per BGP router

and external peer, the simulator provides a useful tool for exploring the effects of policies, as the example in Section 4 shows.

We briefly describe how to use the simulator here. To create a simulation

*bgpSimulation connGraph policyMap asNumberMap*

where *connGraph*, of type *Graph* from the *Data.Graph* module, is the connectivity graph of the nodes in the network, where our network is a single node and nodes are identified with integers, and *policyMap* :: *Int* → *SimPolicy* and *asNumberMap* :: *Int* → *ASNumber* are maps giving for each node identifier the node’s policy and its AS number respectively. The library provides a function *networkToSimPolicy* which calculates the simulator policy of the simulator node representing the home network, given a routing network and a map associating to each external node the BGP connection (part of the Nettle specification) connecting it to the home network. A simulation can then be run and viewed, using the *printSimulation* function, whose use is shown in Section 4.

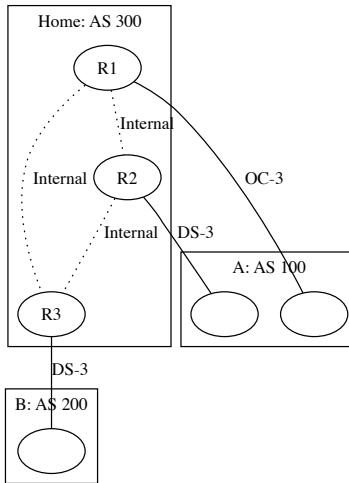
## 4 Extended Nettle Example

We can now demonstrate a complete configuration in Nettle. The example is taken from Zhang et al [10], a Cisco text book on BGP configuration. The example is for the configuration of a network with AS number 300, which is “multi-homed”, (i.e. it connects to the Internet via multiple providers), to providers networks AS 100 and AS 200. AS 300 has 3 BGP-speaking routers, *R1*, *R2*, and *R3* all of which have IBGP sessions among them. Router *R1* is connected to AS 100 via a high-bandwidth OC-3 (Optical Carrier, 155 Mbit/s) connection, while *R2* and *R3* are links to AS 100 and AS 200 respectively over lower bandwidth DS-3 (Digital signal, 45 Mbit/s) links. The topology is shown in Figure 2. For the remainder of this section, we describe the policy from the point of view of the multi-homed network.

Our intention is that most traffic should flow over the OC-3 link to AS 100, while traffic to destinations that are customers of AS 200 should flow over the DS-3 link to AS 200. This policy will achieve some load balancing while also favoring short routes to customers of AS 200. We also plan for some failure scenarios: if the OC-3 link should fail, we would like to roughly balance traffic over the two DS-3 links; if the DS-3 to AS 200 fails, all traffic should flow over the OC-3, whereas if the DS-3 link to AS 100 fails, traffic should be unaffected.

In order to achieve these outbound traffic goals, we first request default and *partial* routes from both providers, meaning that our providers will advertise a default route to the Internet as well as more specific routes to their customers.<sup>3</sup> We can then implement our preference policy with two preference levels, which we will call *high* and *low*. We assign routes traversing the OC-3 link the *high* preference, and all other routes the *low* preference.

<sup>3</sup> This request is not part of BGP; this request is made by informal communication among the networks’ administrators.



**Fig. 2.** The BGP connections for example 1

With this policy, we will never use a route traversing the DS-3 to AS 100 when the OC-3 is working, because 1) we assume that any route advertised by AS 100 will be advertised on both of its links to us and 2) we give a higher preference to routes over the OC-3 link. We will also never use a route traversing the DS-3 link to AS 200 when a route to the same prefix is available over AS 100. We will only use a route traversing the DS-3 to AS 200 when the prefix of that route is not announced by AS 100; this will only be the case when the prefix is for a customer of AS 200 that is also not a customer of AS 100.

We are more limited in our ability to influence inbound traffic patterns. To do this we must manipulate how our routes are advertised in order to influence the decisions that AS 100 and AS 200 make. To influence AS 100's traffic to the OC-3 link, we can prepend our AS number onto routes we advertise across the DS-3 link to AS 100. Assuming that AS 100 does not have explicit preferences set for routes, this will cause AS 100 to prefer the apparently shorter path to us over the OC-3. In order to discourage peers of AS 200 from choosing the route to us through AS 200, we prepend our AS number twice on the link to AS 200. However, in the situation Zhang et al [10] suppose, this amount of prepending causes AS 200 itself (as opposed to its neighbors) to choose the route to us through AS 100 and so the local traffic from AS 200 does not flow over the link with AS 200. To remedy this situation, we take advantage of policies which are known<sup>4</sup> be in place at AS 200. AS 200 has configured policies that assigns routes having community values 200:80, 200:100, and 200:120 the preferences 80, 100, and 120 respectively. By advertising routes to AS 200 with community attribute 200:120, we can cause AS 200 to prefer the direct route to us, while

<sup>4</sup> Knowledge of such policies is communicated in an ad-hoc fashion by neighboring network administrators.

```

import Nettle.Network

home = 300; asA = 100; asB = 200

conn12      = internalConn r1 (address 130 0 1 4) r2 (address 130 0 2 5)
conn13      = internalConn r1 (address 130 0 1 4) r3 (address 130 0 3 4)
conn23      = internalConn r2 (address 130 0 2 6) r3 (address 130 0 3 5)
conn_A_OC3 = externalConn r1 (address 100 100 100 0)
                    (address 100 100 100 1) asA
conn_A_DS3 = externalConn r2 (address 100 100 150 0)
                    (address 100 100 150 1) asA
conn_B_DS3 = externalConn r3 (address 200 200 200 0)
                    (address 200 200 200 1) asB

conns = [conn12, conn13, conn23, conn_A_OC3, conn_A_DS3, conn_B_DS3]
staticNet = staticConfig [route r1, route r2, route r3]
where route r = staticRoute r (address 172 160 0 0 // 16)
                    (localAddr conn_B_DS3)

redistPolicy = redist [nofilter r | r ← [r1, r2, r3]]
wherenofilter r = redistAt r all StaticProtocol BGPProtocol

martians = [address 0      0  0 0 // 8, address 10  0  0 0 // 8,
            address 172  0  0 0 // 12, address 192 168 0 0 // 16,
            address 127  0  0 0 // 8, address 169 254 0 0 // 16,
            address 192  0  2 0 // 24, address 224 0  0 0 // 3,
            address 172  160 0 0 // 16]

adMod c | c ≡ conn_A_DS3 = cond all (prepend home) (always ident)
| c ≡ conn_B_DS3 = cond all
                    (tag (200 :: 120) ▷ prepends 2 home)
                    (always ident)
| otherwise        = always ident

adFilter = const $ reject $ destNotEq homeBlock
where homeBlock = address 172 160 0 0 // 16

net = routingNetwork bgpNet staticNet redistPolicy
where bgpNet = bgpNetwork home conns prefs usefilter adFilter adMod
      prefs = cond (nextHopEq (peerAddr conn_A_OC3)) high
                (always low)
      usefilter = const $ reject $ destInSet martians
      high    = 120
      low     = 100

```

**Fig. 3.** The Nettle configuration for the Zhang et al [10] example; the definitions of the router details  $r1, r2, r3$  are omitted

still prepending our AS number twice, thereby making the route appear long for peers of AS 200.

Finally, we declare use and ad filters. The use filter is a defensive measure which prevents us from accepting routes which are known to be incorrect. The ad filter ensures that we only advertise our own address block and do not inadvertently enable our providers to transit traffic through our network.

The Nettle code for this example, omitting the router-specific details, is shown in Figure 3.

We can use the simulator to check our understanding of the policy. We set up a simulation in which AS 100 announces only the default route, and AS 200 announces the default route and two more specific routes: one for destination 20.20.20.0/24 and another to destination 10.0.0.0/7. We expect that the 10.0.0.0/7 route will be filtered by our martian filter. In addition, we expect the default route over the OC-3 link to be preferred. We also check that we don't inadvertently provide transit traffic to our providers by exporting one provider's routes to the other. The simulation output, shown in Figure 4 confirms our understanding.

```

printSimulation sim
(Node = 1, AS = 300):
( 0.0.0.0 // 0, 100.100.100.1, [100],      120, [])
( 20.20.20.0 // 24, 200.200.200.1, [200,7763],100, [])
(172.160.0.0 // 16, 200.200.200.0, [],          0, [])

(Node = 2, AS = 100):
(172.160.0.0 // 16, 100.100.100.0, [300],0, [])
( 0.0.0.0 // 0, 100.150.150.88, [], 0, [])

(Node = 3, AS = 100):
(172.160.0.0 // 16, 100.100.150.0 ,[300,300],0, [])
( 0.0.0.0 // 0, 100.100.150.23, [], 0, [])

(Node = 4, AS = 200):
(172.160.0.0 // 16, 200.200.200.0 ,[300,300,300],0, [Community 200 120])
( 0.0.0.0 // 0, 10.23.140.223, [],          0, [])
( 20.20.20.0 // 24, 10.23.140.223,[7763],      0, [])
( 10.0.0.0 // 7, 10.23.140.223,[7763],      0, [])

```

**Fig. 4.** A sample simulation for the policy of Section 4

## 5 Nettle as a Basis for User-Defined Policy Languages

The most compelling advantage of Nettle, we believe, is its ability to serve as the basis for the development of higher-level and more specific configuration schemes or patterns. In this section we demonstrate two such schemes that can be precisely expressed in Nettle. These examples illustrate how Nettle enables policy to be written abstractly and at a high-level. Though it is possible to write policy-generating scripts in other languages, the Nettle language offers safety guarantees: if the policy-generating functions type check, then we are sure that the policies they generate will be free from many simple errors. This simple, but substantial verification may allow authors of such functions to write policy-generators more quickly and be more confident in the correctness of the resulting policy.

## 5.1 Hierarchical BGP in Nettle

Gao and Rexford [11] found that commercial relationships between networks on the Internet are roughly of two types: customer-provider relationships and peer relationships. In a customer-provider relationship, one network (the customer) pays the other (the provider) to connect them to the rest of the Internet. In a peer relationship, two networks agree to allow traffic to and from customers of the networks to pass directly between the networks, without payment. These relationships strongly constrain BGP policy in two ways. On the one hand, providers are obliged to support traffic to and from customers. On the other hand networks have an incentive to use customer-learned routes, since using such routes incurs no cost and since customers may pay by amount of data transmitted. These constraints, as described in Griffin et al. [9] can be summarized as follows:

1. Customer routes must be preferred to peer routes, and customer and peer routes must be preferred over provider routes.
  2. Customer routes are be advertised to all neighbors, whereas peer and provider routes can only be shared with customers.

Hierarchical BGP (HBGP) is an idealized model of BGP in which networks relate in the two ways described above and follow the guidelines described. Griffin et al. [9] have shown that these guidelines, if followed by all networks on the Internet, would guarantee the absence of many routing anomalies, including global routing oscillations.

We can express both of these guidelines in Haskell functions that generate Nettle policy given the classification of peerings into HBGP relationship types, and indeed, we have implemented such functions in the *Nettle.HBGP* module. We will need to generate both preferences and advertising filters to implement these guidelines, and we do so with the following two functions:

where we use the following enumerated type to stand for the HBGP relationship types:

```
data PeerType = Customer | Peer | Provider
```

The HBGP preference guidelines do not constrain preferences among neighbors of a given relationship type and we take advantage of this by having the *hbgpPreference* function also take a *PartialOrder ASNumber* argument, which represents route preferences among the neighbors. The function will attempt to satisfy both the HBGP preference constraint and the given partial order, and will report an error if these are incompatible. We omit the implementation of these functions here, but we show how these can be used to easily create a Nettle configuration that follows HBGP guidelines.

In this example, we show only the preference and advertising filter components of the policy for a network with several customers and peers, and one provider. First, we define the set of external network numbers and a map to classify external neighbors into HBGP peer types:

```
home = 100; cust1 = 200; cust2 = 300; cust3 = 400
cust4 = 500; peer1 = 600; peer2 = 700; prov = 800
pTypes = [(cust1, Customer), (cust2, Customer), (cust3, Customer),
           (cust4, Customer), (peer1, Peer), (peer2, Peer), (prov, Provider)]
```

We can then easily define our preferences to be HBGP compatible preferences, where we also give our preference of networks:

```
prefs = hbgpPrefs pTypes basicPrefs
  where basicPrefs = [(cust1, cust2), (cust1, cust4), (cust3, cust4),
                      (peer2, peer1), (prov, prov)]
```

We also add the HBGP advertising filters to our policy:

```
adFilter = hbgpAdFilter peerTyping
```

Compiling our example for a single Xorp router gives a configuration file roughly 400 lines long, much of it consisting of preference setting commands, such as:

```
term impterm13 {
  from {
    as-path: "^\d{2}00|(\d{2}00 \d{1,2} \d{1,2}*\d{1,2}*\d{1,2})*$"
  }
  then {
    localpref: 105
  }
}
term impterm14 {
  from {
    as-path: "^\d{3}00|(\d{3}00 \d{1,2} \d{1,2}*\d{1,2}*\d{1,2})*$"
  }
  then {
    localpref: 104
  }
}
...
...
```

as well as export filters implementing the HBGP scope preferences, such as:

```
term expterm7 {
  to {
    neighbor: 130.6.1.1
    as-path: "^\d{4}00|(\d{4}00 \d{1,2} \d{1,2}*\d{1,2}*\d{1,2})*$"
  }
}
```

```

then {
  reject
}
}
term expterm8 {
  to {
    neighbor: 130.6.1.1
    as-path: "^800|(800 [0-9] [0-9]*([0-9] [0-9]*))*$"
  }
  then {
    reject
  }
}

```

This example illustrates the utility of Nettle in capturing configuration patterns and thus enabling network configurations to be specified more succinctly and at a higher level of abstraction.

## 5.2 Dynamically Adjustable Customer Policy

Providers often receive requests from customers to modify their policy in order to achieve some customer objective. Furthermore, the customer may alter their objective at a later date, and request another change in provider policy. In order to avoid having to repeatedly update their configurations, and thereby increase the risks of introducing accidental errors, providers often provide a way for customers to dynamically alter provider policy. This is typically done through the use of the community attribute. Providers typically configure policies to match on particular community values, which when matched perform some operation, such as setting the preference level of the route to a particular value, or influencing the advertisement of the route. The network configured in Section 4 made use of such a policy when announcing routes to AS 200 by tagging these routes with an appropriate community value.

Zhang et al. [10] describe four commonly occurring types of dynamically adjustable policies used by ISP's. These policy types allow customers to do the following:

1. Adjust preference levels of announced routes.
2. Suppress advertisement of routes to peers by peer type.
3. Suppress advertisement of routes to peers by peer AS number.
4. Prepend ISP AS Number to routes announced to peers with a certain AS Number.

To allow a customer to adjust preference levels, an ISP with AS Number 1000 could add policy to match routes having community attributes 1000:80, 1000:90, ... 1000:120 and set the local preference to 80, 90,..., 120, respectively. To suppress by peer type, AS 1000 could add filters matching community values 1000:210, 1000:220, and 1000:230 where these filters cause matching routes not to be advertised to providers, peers, or customers, respectively. To suppress by AS number,

AS 1000 could add a filter for every neighboring network, with AS number A, matching community 65000:A and suppressing advertisement to A upon matching. To achieve dynamically adjustable prepending, AS 1000 would add a filters for every neighboring network, with AS number A, of the following form: match community 65x00:A, where x is a digit from 0-9, and upon matching, prepend 1000 to the AS path x times.

It is easy to see that such policy will dramatically increase the size of the router configurations with many simple definitions and that writing these by hand will substantially increase the likelihood of introducing errors into configurations. A better solution is to capture these configuration patterns as functions producing Nettle expressions, and instantiating these patterns in configurations by calling these functions. Indeed, we have done exactly this in the module *Nettle.DynamicCustomerPolicy*.

We define a function *adjustablePrefs* which returns customer-adjustable Nettle preferences, given a list of preference levels which may be set. An example is

```
adjustablePrefs homeASNum custASNum (always 100) [80, 100, 120]
```

which denotes preference policy which allows customer number *custNum* to adjust the preference level of their routes among 80, 100, and 120 by tagging their routes with communities *homeASNum*:::80, *homeASNum*:::100, and *homeASNum*:::120 respectively, and otherwise defaults to preference level 100. The implementation of *adjustablePrefs* is relatively straightforward, so we show it here:

```
adjustablePrefs cust home prefElse prefLevels =
  foldr f prefElse prefLevels
  where f p e = cond (routeFrom cust ∧ taggedWith (home :: p)) p e
        routeFrom asn = asSeqIn (repeat (i asn) ▷ repeat any)
```

The *routeFrom cust* predicate ensures that only the specified customer can adjust their own routes.

We can accomplish dynamic prepending with the *adjustablePrepending* function, and an example of its use is the following:

```
adjustablePrepending homeNum custNum [(91, 1), (92, 2), (93, 3), (94, 4)]
  [44000, 13092, 6231] (always ident)
```

which denotes connection-dependent advertising modifier policy which allows customer *custNum* to adjust the prepending done by the provider network when advertising to networks 44000, 13092, and 6231 such that communities 91:44000, 92:44000, ..., 94:44000, 91:13092, ..., 94:6231 correspond to prepending 1, 2, 3, and 4 times, to the specified network, respectively. The implementation of *adjustablePrepending* is more involved than for *adjustablePrefs*, but is nonetheless not too complicated:

```
adjustablePrepending home cust octetsAndTimes nets gmodElse conn =
  if peerASNum conn `member` nets
```

```

then foldr f gmodElse octetsAndTimes
else gmodElse
where f (o, t) e = cond (routeFrom cust
                            $\wedge$  taggedWith (o :: (peerASNum conn)))
                           (prepends t home) e

```

## 6 Implementation

We have implemented the Nettle language as an domain-specific embedded language (DSEL) hosted in Haskell. The embedding confers significant benefits on Nettle, the most important of these being safe, comprehensive and uniform abstraction mechanisms and powerful calculation abilities. The examples in Sections 3, 4, and 5 illustrate how we can take advantage of these to write policy in a high-level and safe way.

We also take advantage of Haskell’s type system, in particular its support for phantom types, generalized algebraic data types, type classes, and existentially qualified types to ensure that Nettle programs satisfy a number of logical constraints. An example of this is the design of the *RoutePred a* datatype, which carries a phantom type indicating the type of routes being predicated over. This phantom type prevents users from combining predicates over incompatible route types. For instance, it would be incorrect to intersect a predicate over static routes with a predicate over BGP routes, and our combinators appropriately prevent this. As more protocols are added to Nettle, it will become increasingly important to ensure predicates are constructed in sensible ways.

The phantom type of *RoutePred a* is used again when constructing route redistribution policies. As explained above, route redistribution injects some subset of routes, denoted by a route predicate, from one protocol to another. A route redistribution policy is then essentially specified by naming the exporting and importing protocols as well as a predicate over routes of the exporting protocol.

A network configuration may include several redistribution policies and we would like to collect these into a single list. However, since each of these policies may have predicates of different types, we need to hide the types in a quantified type. On the other hand, the Nettle-to-XORP compiler will need to know the exporting and importing protocols in order to generate correct code. To accomplish this we need to package the route redistribution policy with values indicating the exporting and importing protocols. In doing this, we want to ensure that these values correspond with the predicates appropriately. We accomplish all this by creating a type, whose values represent protocols and whose type also carries the protocol information:

```

data ProtocolValue a where
  BGPProtocol :: ProtocolValue BGP
  StaticProtocol :: ProtocolValue StaticT

```

We then use this in our definition of route redistribution policies:

```

data RedistPolicy = forall a b
  RedistPolicy (RoutePred a) (ProtocolValue a) (ProtocolValue b)

```

With these types, we can hide the predicate type in the redistribution policy so that it can be collected in a list, while providing values which can be used by the compiler by pattern matching against the constructors of the *ProtocolValue a* type. The phantom types ensure that the compiler will never be given an incompatible protocol value and route predicate. For example, *RedistPolicy (taggedWith (1000:::200)) BGPProtocol StaticProtocol* type checks, whereas *RedistPolicy (taggedWith (1000 ::: 200)) StaticProtocol BGPProtocol* does not.

Furthermore, we take advantage of Haskell's type class mechanism to implement overloaded operators, thereby simplifying the syntax of Nettle. Two examples of this are the sequential composition operator  $\triangleright$  and the Boolean operators  $\vee$  and  $\wedge$ . Sequential composition is used to denote concatenation of regular expressions as well as function composition of route modifiers. In this case we have made the appropriate data types instances of the *Data.Monoid* type class and made  $\triangleright$  synonymous with the *mappend* function of this type class. The Boolean operations are defined in the *Boolean* type class and both route predicates and route predicate-valued functions are made instances of this, where for the latter the operations are defined pointwise, as follows:

```

class Boolean b where
  ( $\vee$ ) :: b  $\rightarrow$  b  $\rightarrow$  b
  ( $\wedge$ ) :: b  $\rightarrow$  b  $\rightarrow$  b
  all :: b
  none :: b

instance Protocol p  $\Rightarrow$  Boolean (r  $\rightarrow$  RoutePred p) where
  b1  $\vee$  b2 =  $\lambda r \rightarrow b1\ r \vee b2\ r$ 
  b1  $\wedge$  b2 =  $\lambda r \rightarrow b1\ r \wedge b2\ r$ 
  all      = const all
  none     = const none

```

## 7 Related Work

In analyzing BGP policy, Caeser and Rexford [12] emphasize the need for languages which express BGP policies more directly and allow for the expression of common policy patterns and Ramachandran [13] and Griffin [9] make contributions in how such languages should be constructed. Several other efforts, including the path vector algebra approach of Sobrinho [14] and Griffin [15] and the “Declarative Networking” approach of Loo et al [16] are promising approaches to this problem. These approaches differ from Nettle's approach in that they give languages for expressing both a protocol and a policy, whereas Nettle focuses solely on the kinds of policies supported by the current BGP protocol. In that regard, Nettle is much more closely related to configuration and policy languages such as XORP and RPSL, and in the following sections we describe the relationship of Nettle to those languages.

## 7.1 XORP

XORP routers are configured using a XORP configuration file. Unlike Nettle, which describes an entire network in a single program (though not necessarily a single file), XORP requires one file per router, and these files contain both router details such as details about network interfaces, and BGP routing policy. Policy is divided into “import” and “export” policy, which are similar to Nettle’s usage and advertising policy. These import and export policies in turn consist of a sequence of guarded actions, similar to Nettle.

Xorp provides some limited facilities for naming and reusing elements. For example, it provides a construct for creating and naming sets of network prefixes that can be referred to in policies and it provides the ability to name a conjunction of basic predicates (XORP calls these “policy subroutines”) for reuse. In contrast, Nettle provides much more extensive ability to name arbitrary policy components, as the above tutorial has demonstrated. The following Nettle example illustrates naming route modifiers, a simple form of naming which is currently not possible in XORP:

```

m1 = tag (5000 :: 130)
m2 = prepend 65000
m3 = unTag (5000 :: 90)
f = m1 ▷ m2
g = m1 ▷ m3

```

Even more importantly, XORP does not provide any functional abstractions. This prevents users from expressing their policy patterns in a reusable way and prevents users from designing their own higher-level policy languages or policy calculation methods.

## 7.2 RPSL

Routing Policy Specification Language (RPSL) is a stand-alone BGP policy language which, like Nettle, takes a network-wide view of BGP configuration. Like XORP it provides several constructs for naming policy elements, such as sets of routes and filters. Like XORP it does not provide comprehensive naming or functional abstraction.

Like Nettle, RPSL is vendor neutral and can be used to generate router configuration files. The *RtConfig* tool[17] generates a router configuration file from RPSL policy and script file containing router-specific information. Essentially, the *RtConfig* script file is a vendor-specific router configuration file with special commands inserted which instruct *RtConfig* to insert information from the RPSL policy. While this does manage to separate router specific details from BGP policy, it has the disadvantage that router-specific details are external to the language and there are therefore no mechanisms for reusing parts of these router-specific configurations or abstracting the patterns which generate them. In contrast, Nettle separates policy from router details, yet embeds both within a single language, Haskell, which gives users a uniform tool for manipulating both policy and router details.

## 8 Limitations and Future Work

Nettle currently only supports BGP and static routing. In order to be a practical alternative to vendor-specific router configuration scripts, Nettle will need to add support for the most common IGP, such as RIP and OSPF. Nettle is also missing some BGP features commonly supported on all major platforms, including setting and testing the MED attribute and supporting route aggregation.

More significantly, while Nettle provides greater expressiveness than other routing configuration languages, it is still fairly low-level and does not fully address the issue of configuration correctness. For example, in the extended example of Section 4, the intended network behavior and the assumptions made of the network are expressed informally. The BGP configurations which achieve these objectives do not directly express the intentions, and the correctness of the configurations can only be judged with respect to those intentions. Furthermore, the assumptions about the network are essential in reasoning about the correctness of the configurations. We conclude that further work is needed in order to address the issue of policy correctness.

**Acknowledgements.** Thanks to Vijay Ramachandran, whose work on network routing policy inspired this effort. Thanks also to the anonymous reviewers who supplied helpful feedback on an earlier draft of this paper. This research was supported in part by NSF grant CCF-0728443 and DARPA grant STTR ST061-002 (a subcontract from Galois, Inc.).

## References

1. Rekhter, Y., Li, T., Hares, S.: A Border Gateway Protocol 4. Internet Engineering Task Force (2006)
2. Alaettinoglu, C., Villamizar, C., Gerich, E., Kessens, D., Meyer, D., Bates, T., Karrenberg, D., Terpstra, M.: Routing Policy Specification Language (RPSL). Internet Engineering Task Force (June 1999)
3. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming* 13(1), 0–255 (2003)
4. Xorp, Inc.: Extensible Open Routing Platform, Xorp User Manual, Version 1.6 (January 2009)
5. Varadhan, K., Govindan, R., Estrin, D.: Persistent route oscillations in inter-domain routing. *Computer Networks* 32(1), 1–16 (2000)
6. Mahajan, R., Wetherall, D., Anderson, T.: Understanding BGP misconfiguration. *SIGCOMM Comput. Commun. Rev.* 32(4), 3–16 (2002)
7. Hudak, P.: Building domain specific embedded languages. *ACM Computing Surveys* 28A (December 1996) (electronic)
8. Hudak, P.: Modular domain specific languages and tools. In: *Proceedings of Fifth International Conference on Software Reuse*, pp. 134–142. IEEE Computer Society, Los Alamitos (1998)
9. Griffin, T.G., Jaggard, A.D., Ramachandran, V.: Design principles of policy languages for path vector protocols. In: *SIGCOMM 2003: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 61–72. ACM, New York (2003)

10. Zhang, R., Bartell, M.: BGP Design and Implementation. Cisco Press (2003)
11. Gao, L., Rexford, J.: Stable internet routing without global coordination. SIGMETRICS Perform. Eval. Rev. 28(1), 307–317 (2000)
12. Caesar, M., Rexford, J.: BGP routing policies in isp networks. IEEE Network 19(6), 5–11 (2005)
13. Ramachandran, V.: Foundations of Inter-Domain Routing. PhD thesis, Yale University (May 2005)
14. Sobrinho, J.L.: Network routing with path vector protocols: theory and applications. In: SIGCOMM 2003: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 49–60. ACM, New York (2003)
15. Griffin, T.G., Sobrinho, J.L.: Metarouting. In: SIGCOMM 2005: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 1–12. ACM, New York (2005)
16. Thau, B., Joseph, L., Hellerstein, M., Stoica, I., Ramakrishnan, R.: Declarative routing: Extensible routing with declarative queries. In: Proceedings of ACM SIGCOMM 2005 (2005)
17. Meyers, D., Schmitz, J., Orange, C., Prior, M., Alaettinoglu, C.: Using RPSL in Practice. Internet Engineering Task Force (August 1999)

# Generic Libraries in C++ with Concepts from High-Level Domain Descriptions in Haskell

## A Domain-Specific Library for Computational Vulnerability Assessment

Daniel Lincke<sup>1</sup>, Patrik Jansson<sup>2</sup>, Marcin Zalewski<sup>2</sup>, and Cezar Ionescu<sup>1</sup>

<sup>1</sup> Potsdam Institute for Climate Impact Research, Potsdam, Germany

{daniel.lincke,ionescu}@pik-potsdam.de

<sup>2</sup> Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden

{patrikj,zalewski}@chalmers.se

**Abstract.** A class of closely related problems, a problem domain, can often be described by a domain-specific language, which consists of algorithms and combinators useful for solving that particular class of problems. Such a language can be of two kinds: it can form a new language or it can be embedded as a sublanguage in an existing one. We describe an embedded DSL in the form of a library which extends a general purpose language. Our domain is that of vulnerability assessment in the context of climate change, formally described at the Potsdam Institute for Climate Impact Research. The domain is described using Haskell, yielding a domain specific sublanguage of Haskell that can be used for prototyping of implementations.

In this paper we present a generic C++ library that implements a domain-specific language for vulnerability assessment, based on the formal Haskell description. The library rests upon and implements only a few notions, most importantly, that of a monadic system, a crucial part in the vulnerability assessment formalisation. We describe the Haskell description of monadic systems and we show our mapping of the description to generic C++ components. Our library heavily relies on *concepts*, a C++ feature supporting generic programming: a conceptual framework forms the domain-specific type system of our library. By using functions, parametrised types and concepts from our conceptual framework, we represent the combinators and algorithms of the domain. Furthermore, we discuss what makes our library a domain specific language and how our domain-specific library scheme can be used for other domains (concerning language design, software design, and implementation techniques).

## 1 Introduction

The road from a domain of expert knowledge to an efficient implementation is long and perilous, often resulting in an implementation that bears little apparent resemblance to the domain. Still, the practitioners of high-performance computing are willing to pay the price of overwhelmingly low-level implementations to guarantee performance. In the current day and age, however, it is possible to increase the level of abstraction by applying *generic programming* techniques to provide a high-level library for a domain that allows structured use of existing, efficient code. In this paper, in particular,

we show how to first describe a domain in a high-level *functional* style, close to mathematical notation but executable, and, then, transform this description into a strongly generic library that corresponds almost directly to the initial description. Specifically, we present the process by which we derived a generic library for vulnerability assessment (in the context of environmental sciences) in C++ (ANS, 2003; Stroustrup, 1997).

The concept of “vulnerability” plays an important role in many research fields, including for instance climate change research. As we will discuss in section 2, there is no unique definition of this term. However, there is a simple common structure underlying the definitions in the usage of “vulnerability”, which can be shortly described as measuring the possible harm for a given system.

Our C++ implementation is directly based on the high-level description given in Haskell (Peyton Jones, 2003) by Ionescu (2009). Ionescu relies on functional features of Haskell, such as function types and type constructors, to represent the underlying category-theoretical notions of, among others, functors, coalgebras, monads, and, crucially, of a construct of *monadic dynamical systems*, which we will introduce and explain in section 2.2. While the description can be actually executed, its design is not focused on efficiency. Furthermore, it is not generic enough to be applied in a practical setting of existing implementations, tool sets, and so on.

Our C++ library maintains the abstract, high-level correspondence with the domain notions introduced by Ionescu but it makes efficient implementations possible by providing a generic interface for use of high-performance components. We are able to achieve the abstraction by basing the library on a hierarchy of *concepts*, a language mechanism in C++ supporting generic programming (Gregor et al., 2006, 2008c)<sup>1</sup>. While the use of concepts is not novel itself (e.g., Gregor et al., 2008b), we use a conceptual framework to generically specify “types of types” corresponding to the types from the domain description. The library consists of the three levels:

- a conceptual framework specifying concepts (type classes) for functions, functors, monads, and so on;
- parametrised types, themselves “typed” by concepts, that represent domain constructs and combinators;
- and, finally, parametrised functions representing the algorithms applicable in a given domain.

The library can be seen as a domain specific language embedded into C++, complete with its own type system and language constructs.

A library like ours is much more complex and verbose than its high-level specification in Haskell but it is surprisingly similar and nimble from the point of view of the user. For example, vulnerability is cleanly specified in Haskell as a function of the following type (see Sect. 2 for details):

---

<sup>1</sup> Haskell provides a similar mechanism of *type classes* (Wadler and Blott, 1989) that could potentially serve as the basis for a Haskell library similar to our C++ library. For comparison of Haskell type classes and C++ concepts, see the work of Bernardy et al. (2008).

---

```

1 vulnerability :: (Functor m, Monad m) =>
2     (i -> x -> m x) -> ([x] -> harm) ->
3     (m harm -> v) -> [i] -> x -> v

```

---

In the C++ library, the type of the function has to be translated to the conceptual framework (see Sects. 3 and 4 for details):

---

```

1 template <MonadicSystem Sys, Arrow Harm, Arrow Meas, ConstructedType Inputs>
2 requires ConstructedType<Harm::Domain>, ConstructedType<Meas::Domain>,
3 SameType<Harm::Domain::Inner, Sys::Codomain::Domain>,
4 SameType<Meas::Domain::Inner, Harm::Codomain>,
5 SameTypeConstructor<Meas::Domain, Sys::Codomain::Codomain>,
6 SameType<Inputs::Inner, Sys::Domain>,
7 FMappable<Harm, Rebind<Sys::Codomain::Codomain, Harm::Domain>::Computed>;
8
9 Meas::Codomain
10 vulnerability_sys(Sys, Harm, Meas, Inputs, Sys::Codomain::Domain);

```

---

In C++, the type of the Haskell `vulnerability` function must be expressed in terms of concepts that form the basis of the library (see Sect. 5 for discussion of the translation scheme). While clearly more complex from the point of view of a library developer, the complexity is hidden when the function is used:

---

```
vulnerability_sys(my_sys, my_harm, my_measure, my_inputs, my_state);
```

---

The user must declare the necessary *concept maps* that describe how the types to which `vulnerability_sys` is applied model the necessary concepts. The mapping through concept maps makes it possible to use the library with the existing high-performance implementations, keeping the complexity manageable, since the necessary concept maps are specified one at a time, and with the right use of C++ concept features many concept maps can be generated by the compiler.

In order to give an overview over the library, the core concepts from the library are outlined in Fig. 1. In the left column we give the signature of the C++ concept and refinement relations. On the right side we show how the parameters of the concept match the Haskell signature (we write  $C \sim t$  for “C++ type  $C$  corresponds to Haskell type  $t$ ”). Furthermore we give a brief explanation of what the concept is intended to model. The concepts are divided into a few groups. This grouping does not imply any hierarchy of the used concepts, it is only for representation reasons.

In the remainder of the paper, we outline the progression from the domain to an efficient implementation, illustrated by our particular path from the domain of vulnerability assessment to our C++ library. In Sect. 2, we present the concept of vulnerability as used in the climate change community and its mathematical formalisation in Haskell. Section 3 shows the C++ implementation of the domain specific constructs (concepts, parametrised types and functions) and Sect. 4 shows how the library can be used.

Arrow types	
Arrow<class Arr>	Arr ~ a -> b : describes general function types
CurriedArrow<class CArr> : Arrow <CArr>	CArr ~ a -> (b -> c) : functions whose codomain is another function
Coalgebra<class Arr> : Arrow<Arr>	Arr ~ x -> f x : functions whose codomain is a functor application
CoalgebraWithInput<class Arr> : Arrow<Arr>	Arr ~ (a, x) -> f x : coalgebras with an additional input
MonadicCoalgebra<class Arr> : Coalgebra<Arr>	Arr ~ x -> m x : functions whose codomain is a monad application
MonadicCoalgebraWithInput<Arr> : CoalgebraWithInput<Arr>	Arr ~ (a, x) -> m x : monadic coalgebras with an additional input
Type constructors	
ConstructedType<class T>	T ~ f x : the type T is a constructor application
SameTypeConstr<class T1, class T2>	T1 ~ f x, T2 ~ f y : two constructed types have the same type constructor
Functors and Monads	
FMapable<class Arr, class Type>	Arr ~ a -> b, Type ~ f a : provides operation FMap for the function Arr and the type Type
MBindable<class MX, class Arr>	MX ~ m x, Arr ~ x -> m y : provides operation MBind for the type MX and the function Arr
MReturnable<class MX>	MX ~ m x : provides operation MReturn for MX
Miscellaneous	
Monoid<class Op>	Op ~ t -> t -> t : Op is a monoid operation
Monadic system	
MonadicSystem<class Sys> : CurriedArrow<Sys>	Sys ~ t -> (x -> m x) : Sys is a monadic system

Fig. 1. Overview of the C++ concepts in the library

The process by which we arrived at C++ code is highly reusable: while some human ingenuity is required much of the work is mechanical, translating a mathematical description into a generic library. Furthermore, some parts of our library, such as generic representation of function types and type constructors, can be directly reused when implementing other domains. In fact, in Sect. 5 we present a scheme for translation of functional Haskell types into our reusable, “types of types” concepts. Yet, although our library is generic, we still made some particular design choices. In Sect. 6 we survey related work and in Sect. 7 we discuss some of the choices and possible alternatives, including the ideas on how to further generalise the C++ representation.

## 1.1 Preliminaries

We use Haskell code for describing the mathematical model and C++ code for describing the implementation. Here we give a very short description of some of the notation used. It is worth noting already from the start that `:` is used in two different ways: in Haskell it is the “has type” keyword, while in C++ it is the “scope resolution” operator.

**Haskell.** For specification of the mathematical model we use the Haskell programming language (Peyton Jones, 2003), without any of the language extensions. We make

frequent use of the type class `Monad m` with members `return` and `(>=)` (pronounced “bind”) and the class `Functor f` with the member `fmap`.

---

```
class Functor f where
  fmap   :: (a -> b) -> f a -> f b
class Monad m where
  (>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

---

For Kleisli composition we use `(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)` and normal function composition is `(.) :: (b -> c) -> (a -> b) -> (a -> c)`.

We use two recursion operators for lists — `foldr` and `scanr` — which are best introduced by example. If we define `sum = foldr (+) 0` and `sums = scanr (+) 0` then `sum [3,2,1]` is 6 and `sums [3,2,1]` are the partial sums `[6,3,1,0]`. Following a similar pattern we use `foldr` to compose lists of monadic actions:

---

```
compose  :: Monad m => [x -> m x] -> (x -> m x)
compose  = foldr (<=<) return
```

---

For testing we write predicates (parametrised test cases) as normal Haskell functions run by QuickCheck Claessen and Hughes (2000). We use the type `MyFloat` (a wrapper around a normal `Float`) to enable “approximate equality” (needed for testing floating point computations). We use the type constructor `SimpleProb` for probability distributions with finite support.

**C++.** In C++, genericity is achieved through the use of templates (Vandervoorde and Josuttis, 2002), which are pieces of code parametrised by types and values. Type parameters of templates are constrained by concepts and implementations are declared to model particular concepts in concept maps; when templates are instantiated, concept maps are used to bind names dependent on type parameters to the implementations provided by the user (Gregor et al., 2006, 2008c).

A concept can be seen as a *predicate* (or relation) over types. When a type (or a tuple of types) satisfies this predicate, we say that it forms (or they form) a *model* of the concept. The body of a concept can specify a number of *associated entities*: values, functions, types and axioms. Definitions for these entities are provided separately, in concept maps, for all models of the concept. The following snippet of code is a complete example of using concepts:

---

```
1 concept LessThanComparable<typename T> { bool operator<(T x, T y); }
2
3 template<typename T> requires LessThanComparable<T>
4 T min(T x, T y) { return x < y ? x : y; }
5
6 concept_map LessThanComparable<int> { }
7
8 int x = min(2, 3);
```

---

The concept `LessThanComparable`, defined on Line 1, defines a predicate on a single type and requires an associated operator `<`. The concept is then used, on Line 3, to constrain the sole type parameter of the `min` algorithm, making the values of the parameter type comparable with the operator `<`. The built-in type `int` is declared to model `LessThanComparable`, on Line 6. The definition of the associated operator is not given explicitly. In such situations, the compiler is required to attempt to find a matching definition in the context, and, in this case, it finds the built-in operator for `int`. Finally, the usage of concept-constrained `min` is demonstrated on Line 8: the application `min(2, 3)` concept-checks, since the required concept map has been defined.

To use concepts in generic code fragments the keyword `requires` can be used to specify the requirements of template parameters. Unary concept requirements can be specified directly in the template parameters, that is

---

```
template<class A> requires Concept_B<A> ...
```

---

can be written as `template<Concept_B A> ...` and we frequently use this notation.

In C++ we do not use an extra data type for real numbers. We use floating point types and if we need “approximate equality” we use an overloaded operator `==`. All the C++ code examples in this paper were implemented and tested with the ConceptGCC compiler<sup>2</sup> (version alpha 7). It is based on the gcc compiler, version 4.3.

Furthermore, in all C++ code we omit unnecessary details such as `const` qualifiers or reference (`&`) type modifiers — while these are important for performance they do not contribute to the exposition. Finally, in template parameter lists, the keywords `class` and `typename` are freely exchangeable — the variations in the paper are motivated by space constraints.

## 2 Vulnerability Modelling

In the past decade, the concept of “vulnerability” has played an important role in the fields such as climate change, food security, or natural hazard studies. Vulnerability studies have been useful in alerting policymakers to the possibility of precarious situations in the future.

However, definitions of vulnerability vary: there seem to be almost as many definitions of vulnerability as case studies conducted. Wolf et al. (2008), for instance, analyse the usage of 20 definitions of vulnerability. An allegory frequently used to describe the terminology is the Babylonian confusion, and the need for a common understanding has repeatedly been expressed in the literature (Brooks, 2003; Janssen and Ostrom, 2006).

While definitions of vulnerability are usually imprecise, Ionescu (2009) proposes a mathematically defined framework that underlies the definitions and the usage of the term *vulnerability*. Ionescu’s framework applies to those studies that “project future conditions” by using computational tools and that can thus be designated as computational vulnerability assessment. This common framework for vulnerability assessment consist of three primitive concepts: an *entity* (which is subject to possible future

---

<sup>2</sup> <http://www.generic-programming.org/software/ConceptGCC>

harm), *possible future evolutions* (of the entity), and *harm*. Next, we briefly summarise Ionescu’s development. First, we discuss the primitive concepts and, then, their extension to *monadic dynamical systems*.

## 2.1 A Simple Mathematical Model

An entity, its current situation, and its environment are described by a state. In vulnerability computations, the current state of an entity is only a starting point; it is the *evolution* of the state over time that is the main focus of interest. A single evolution path of an entity is called a *trajectory*. In a first, simple vulnerability model, the evolution of an entity is computed in a single step:

---

```
possible  :: State -> F Evolution
```

---

The `possible` function maps a state into an  $F$ -structure of future evolutions. Two types and one type constructor are involved: `State` is the type of the state of an entity, `Evolution` is the type of the evolution of an entity, and  $F$  is the type constructor representing the structure of evolutions (e.g., a set). In this simple model we take `Evolution` = `[State]`: an evolution is a finite sequence of the states that an entity successively assumes over time. This notion is sufficient, as most vulnerability assessments are made for a given horizon, which is expressible as a finite evolution. The constructor  $F$  is a functor (see Sect. 1.1) that describes the nature of the future computation. For example, if the future of an entity is deterministic, one would have  $F = \text{Id}$ , the identity functor, giving a single evolution. Sometimes, a more sophisticated model may be necessary. For instance, a stochastic model might be used, which computes probability distributions over trajectories; the stochastic functor introduced in Sect. 1.1 captures such situations ( $F = \text{SimpleProb}$ ). Some other models might produce just lists of trajectories, without assigning a specific probability to each possibility. For these models, the trajectories are enclosed in the list functor ( $F = [1]$ ).

The second ingredient of a vulnerability computation is a harm judgement function:

---

```
harm      :: Evolution -> Harm
```

---

The `harm` function maps evolutions into values of type `Harm`, which describes the nature of the harm assessment. One of the simplest harm assessments is a threshold measurement. Examples of thresholds that can be crossed include temperature that grows too high or a budget that is too low. Correspondingly, threshold harm measurement judges two possibilities, namely “yes, there is harm for this evolution” or “no, there isn’t any harm when the entity evolves in this way.” In such cases, harm is simply a boolean value ( $\text{Harm} = \text{Bool}$ ). In other models, the harm value for each trajectory may be measured as a real value ( $\text{Harm} = \mathbb{R}$ ). A more sophisticated model might measure multiple dimensions of harm, all expressed as real values, which would lead to  $\text{Harm} = \mathbb{R}^N$ .

The `harm` function measures the harm of a single evolution, but a model produces an `F`-structure of evolutions, given some implementation of the function possible. The `harm` function must then be mapped to the evolutions, using the `map` function of the functor `F` (see Sect. 1.1). A measure function takes the resulting `F`-structure of `harm` values and summarises them into an overall measurement:

---

```
measure :: F Harm -> V
```

---

The codomain of the `measure` function could be, for example, the real numbers (to get the result as a single number, a way of expression policymakers prefer) or a multidimensional space (to express various dimensions of vulnerability instead of collapsing it into a single number). Common examples for the `measure` function are, for instance, `measure = maximum` in the case of `F = []` and `Harm = Bool`, or `measure = expectation` in the case of `F = SimpleProb` and `Harm = ℝ`.

Combining these three functions gives the vulnerability computation:

---

```
vulnerability :: State -> V
vulnerability = measure . fmap harm . possible
```

---

This computation captures the idea behind many vulnerability definitions: we *measure* the *harm* that might be *possible*. Ionescu (2009) specifies certain conditions that the ingredients of the vulnerability computation must fulfil; the `measure` function, for example, has to fulfil a monotonicity condition.

The following example illustrates the use of the vulnerability computation framework introduced in this section:

---

```
1 type Economy_State = (Float, Float,      Float, ...)
2           -- GDP,  GDP-growth, Unemployment Rate, more ...
3 type Economy_Evolution = [Economy_State]
4
5 possible_economy :: Economy_State -> [ Economy_Evolution ]
6 possible_economy x = -- ... implement a model of an economy here
7
8 economic_harm :: Economy_Evolution -> Float
9 economic_harm ec_ev =
10  -- ... return 0 if GDP-growth is always >= 0,
11  --      otherwise the absolute value of the biggest GDP-loss
12
13 economic_vulnerability :: Economy_State -> Float
14 economic_vulnerability = maximum . (fmap economic_harm) . possible_economy
```

---

The implementation of the `economy evolution` function is unspecified, but the type indicates that it is performed non-deterministically. `Harm` is defined as the greatest decrease in Gross Domestic Product (GDP) over a trajectory; the `harm` function returns zero if the GDP is always increasing. In the final vulnerability computation, the `maximum` function is used as `measure`, defining vulnerability as the greatest economic loss over all possible evolutions of the economy in a given state.

## 2.2 Monadic Systems and Vulnerability

The simple model of vulnerability presented in the previous section has two main disadvantages. First, it is not very generic: as shown in the economy example we have to write a new function for every vulnerability computation. This can easily be fixed by making possible, harm and measure inputs to the vulnerability computation. Second, the model is inflexible: the evolution function of the simple model only consumes the initial state of a system as the input. In practise, however, an evolution function often has more inputs. In particular, an evolution may depend on auxiliary control arguments that are not part of the initial state of a system.

Ionescu (2009) explores possible structures which might provide the required flexibility. One the one hand the multitude of systems which might be involved has to be represented, on the other hand we want an easy way of computing iterations. It turns out there, that a functorial structure on the possible evolutions is not suitable for the trajectory computations and that the use of a monad instead of a functor gives the possibility to iterate a system. Therefore, the notion of a *monadic dynamical systems* (MDS) was introduced. In contrast to the possible function, trajectories of a model defined by an MDS can be computed in multiple separate steps. For example, trajectory of an MDS can be computed up to a certain point, the result archived, and the computation continued later with this intermediate result as a starting point. An MDS is a mapping with the signature  $T \rightarrow (X \rightarrow M X)$ , where  $M$  is a monad (see Sect. 1.1). An MDS takes a value of type  $T$  (e.g.,  $T = \text{Nat}$ ) to a (*monadic*) *transition function* of type  $X \rightarrow M X$ . Ionescu requires some further conditions, but we don't need them for this paper.

An MDS need not be directly defined by the user. Often it is built by constructor functions, given a transition function. In the simplest case, the function describes only the transition from one state into the possible next states. The following function defines for instance a non-deterministic transition (with state type  $\text{Int}$  and the list monad):

---

```

1 my_discrete_trans :: Int -> [Int]
2 my_discrete_trans x = [x-1, x+1]

```

---

We can construct an MDS from `my_discrete_trans` using `discrete_sys`:

---

```

1 type Nat = Int
2 discrete_sys :: (Monad m) => (x -> m x) -> (Nat -> x -> m x)
3 discrete_sys f 0      =  return
4 discrete_sys f (n + 1) =  (>>= f) . discrete_sys f n

```

---

Not all transitions of interest are that simple. Often, a transition function does not only map a state into an  $M$ -structure of states, but also takes an additional control parameter. This parameter describes some external input which is not a part of the state. Such function describes a family of transition functions, one for each control input. Consider the following example:

---

```

1 my_input_trans :: MyFloat -> (MyFloat -> SimpleProb MyFloat)
2 my_input_trans delta x = SP [(x + delta, 0.75),
3                               (x - delta, 0.25)]

```

---

Here, every input `delta` gives a different transition function. The function itself is not an MDS (because it fails to satisfy the MDS laws). But it can be used to construct a monadic system by using the following constructor:

---

```

1 input_system :: (Monad m) => (i -> x -> m x) -> ([i] -> x -> m x)
2 input_system trans = compose . map trans

```

---

Furthermore, given an MDS, two iteration functions can be defined: the macro trajectory function, `macro_trj`, computes the evolution of an M-structure of states, and the micro trajectory, `micro_trj`, computes the M-structure of evolutions (sequences of states). Both computations start with a sequence of inputs of type `i` and an initial state. The macro trajectory describes all possible states at each step of the computation, but does not connect them:

---

```

1 macro_trj :: (Monad m) => (i -> (x -> m x)) -> [i] -> x -> [m x]
2 macro_trj sys ts x = scanr (\i mx -> mx >= sys i) (return x) ts

```

---

The micro trajectory records the single evolution paths including all intermediate states:

---

```

1 micro_trj :: (Monad m) => (i -> (x -> m x)) -> [i] -> x -> m [x]
2 micro_trj sys ts x = compose (map (addHist . sys) ts) [x]
3
4 addHist :: (Monad f) => (x -> f x) -> [x] -> f [x]
5 addHist g (x : xs) = liftM (:(x : xs)) (g x)

```

---

Computing the trajectories for the MDS constructed out of `my_discrete_trans` from the initial state `myx = 0` and the input sequence `myts = [1,1,1]` yields:

---

```

1 macro_trj (discrete_sys my_discrete_trans) myts myx ==
2 [ [-3,-1,-1,1,-1,1,1,3],
3   [-2,0,0,2],
4   [-1,1],
5   [0]]
1 micro_trj (discrete_sys my_discrete_trans) myts myx ==
2 [ [-3,-2,-1, 0], [-1,-2,-1, 0],
3   [-1, 0,-1, 0], [ 1, 0,-1, 0],
4   [-1, 0, 1, 0], [ 1, 0, 1, 0],
5   [ 1, 2, 1, 0], [ 3, 2, 1, 0]
6 ]

```

---

The example clarifies the nature of the two trajectory computations: The macro trajectory provides the information that the state of this system is 0 at the beginning. After one step the state is either -1 or 1, after two steps it can be in state -2, 0 or 2 and so on. The micro trajectory contains all single trajectories the system can take, from  $[-3, -2, -1, 0]$ , where we always take -1 path, to  $[3, 2, 1, 0]$ . Note, that the computed states are prepended to the list, leading to the unintuitive ordering of states with the final state at the head of the list. The input of `myts` are processed in the same order: starting from the end of the list and proceeding to the first element.

Many computational models used in the climate change community already implicitly implement the structure of an MDS. For instance, the climate model CLIMBER (Petoukhov et al., 2000) has an internal state which represents the current configuration of certain physical parameters. In addition, CLIMBER takes as an input the concentration of greenhouse gases. The CLIMBER model can be described as an MDS built from a family of transition functions with signature `climber :: GHG -> (Climber_st -> Id Climber_st)`, where `GHG` is the type of values representing concentration of greenhouse gases, and evolution uses the `Id` monad, since CLIMBER is deterministic.

To tie back to vulnerability assessment, the `micro_trj` computation for an MDS fits the signature of the `possible` function in a vulnerability computation. Thus, the `possible` can be computed as the structure of micro-trajectories of an MDS. Taking this into account we can compute vulnerability as:

---

```

1 vulnerability' :: (Functor m, Monad m) =>
2           (i -> x -> m x) -> ([x] -> harm) ->
3           (m harm -> v) -> [i] -> x -> v
4 vulnerability' sys harm measure ts = measure . fmap harm . micro_trj sys ts

```

---

Using the above definitions, the examples of the `possible` functions from the previous section, with `F = Id`, `F = []`, or `F = SimpleProb` as the resulting structures, can all be easily translated to an MDS, since all these functors are also monads.

### 3 C++ Implementation of the Vulnerability Model

The Haskell description of vulnerability which has been provided in the previous section was rather on a mathematical level than a computational library. Due to its laziness and its clean notation Haskell is useful for modelling and prototyping language, which enables one to write operational definitions that can be executed.

As seen in the previous section, the mathematical description of vulnerability and monadic systems is a generic one, for instance a monadic system has three type parameters involved. To take advantage of this flexibility we implemented generic Haskell software components. If we want to translate these description into a C++ library, we have to use generic programming techniques, which are nowadays well supported (by the C++ concepts language extension).

However, Haskell is not the first choice for practitioners of high-performance computing when implementing generic libraries. In Haskell there are overheads in terms

of the runtime system, garbage collection, dictionary passing etc. Many of these problems can be overcome by tuning the GHC compiler to do clever optimisations and specialisations, but in most production environments there is a lack of Haskell expertise. Furthermore C++ has many existing scientific computing libraries which can be used to implement computations in transition functions for monadic systems. In addition C++ is widely distributed and supports multiple software development paradigms on different level of abstraction. Therefore C++ was chosen as the implementation language.

As shown by Bernardy et al. (2008) there are many similarities between generic programming in Haskell (using type classes) and C++ (using concepts). But there are some important differences between these two languages which make a direct translation of the Haskell model from the previous section difficult. Two of the differences are:

- Haskell offers a built in type for functions, C++ does not ;
- Type constructors (parametrised types) are well supported in Haskell, one can use them as parameters in generic functions or combine them. Type constructors (template classes) in C++ do not offer this first-class support.

Due to these problems, we have to implement a conceptual framework specifying a system of “types of types”

### 3.1 The Conceptual Framework

We use concepts to specify types which are not part of standard C++, such as functions, functors and monads. Function types, for instance, are encoded by the following `Arrow`<sup>3</sup> concept, where we hint at the relation to Haskell types with a comment:

---

```

1 // F ~ a -> b
2 concept Arrow<class F> {
3     typename Domain;
4     typename Codomain;
5
6     Codomain operator () (F, Domain);
7 };

```

---

This concept is related to the design pattern of function objects. A function object is simply any object that can be called as if it is a function, thus providing a application operator. In that sense, the concept `Arrow` provides a static interface for function objects. Its name comes from the fact that we do not state that these types model functions in the mathematical sense, for instance side effects might be involved. Furthermore the concept framework provides additional concepts for special kinds of arrows, e.g. the concept `CurriedArrow` describing mappings of type  $f :: A \rightarrow (B \rightarrow C)$ .

Another important issue is how to describe types which are constructed by explicitly applying a type constructor. We will call these types constructed types. As we cannot (due to the technical limitations) describe type constructors directly, the concept `ConstructedType` is used to describe these types. An associated type is used to keep the original type:

---

<sup>3</sup> This concept does not correspond to the Haskell type class `Arrow` ( $a :: * \rightarrow * \rightarrow *$ ).

---

```

1 // T ~ g Inner
2 concept ConstructedType<typename T> {
3   typename Inner;
4 };

```

---

A parametrised concept mapping can be used to declare all STL-containers to be constructed types and therefore instances of this concept:

---

```

1 template<std::Container C>
2 concept_map ConstructedType<C> {
3   typedef C::value_type Inner;
4 };

```

---

Furthermore there are concepts modelling relations between types and/or type constructors: the built-in concept `SameType` decides if two types are equal and the concept `SameTypeConstructor` is fulfilled for all pairs of constructed types sharing the same type constructor.

Based on these rather simple concepts we can model more complex concepts. For instance, we implement the `MBindable` concept as a relation between types:

---

```

1 // MX ~ m x;   Arr ~ x -> m y
2 concept MBindable<class MX, class Arr> {
3   requires ConstructedType<MX>, Arrow<Arr>,
4   ConstructedType<Arr::Codomain>,
5   SameType<MX::Inner, Arr::Domain>,
6   SameTypeConstructor<MX, Arr::Codomain>,
7   CopyConstructible<Arr::Codomain>;
8
9   Arr::Codomain MBind(MX, Arr);
10 };

```

---

It states that there is an `MBind` operation for a pair of types `MX` and `Arr` if certain conditions (stated by requirements) are fulfilled. This member operation implements the monadic bind. In Haskell the bind operation is associated with the monadic type constructor, but here bind is associated with the types of its arguments. A similar concept `MReturnable` models types supporting `mreturn` and `FMappable` is used to model that there is an `fmap` operation.

Using `FMappable` we can describe *coalgebras*: arrows of type  $X \rightarrow F X$  where  $F$  is a functor. Similarly, using `MBindable` and `MReturnable` we can describe *monadic coalgebras*. We implemented concepts for both of them, the `MonadicCoalgebra` for instance looks as follows:

---

```

1 // Arr ~ x -> m x
2 concept MonadicCoalgebra<class Arr> : Coalgebra<Arr> {
3   requires MBindable<Codomain, Arr>, MReturnable<Codomain>;
4 };

```

---

Moreover, we provide a concept `MonadicCoalgebraWithInput` which describes arrows of type  $f :: (A, X) \rightarrow M X$  where  $M$  is a monad. This arrow type is necessary to implement monadic systems with an external control (realised as an input to the system).

On the top of the hierarchy of concepts we specify the main concept of our type specification system: monadic systems. As monadic systems are defined as a special form of a curried mapping, we refine the concept `CurriedArrow`:

---

```

1 // Sys ~ t -> (x -> m x)
2 concept MonadicSystem<class Sys> : CurriedArrow<Sys> {
3     typename Op; // Op ~ (t, t) -> t
4     requires Monoid<Op>, MonadicCoalgebra<Codomain>;
5 };

```

---

Note that there are more conditions, in particular we have the compatibility conditions for the mapping, C++ concepts offer the possibility to express them as axioms which can be embedded concepts. The expression and processing of axioms is beyond the scope of this paper.

### 3.2 The Type System

The concepts presented so far build the type specification system of the language. However the language does not only contain abstractions of datatypes in the form of concepts, we also provide specific constructors realised as parametrised types. These types are on two levels: constructs which support the language by implementing helpful operations and constructs which are part of the domain. Helper constructs are for instance function operations like arrow composition, arrow pairing and arrow currying. The operation `kleisli_compose` used in the description of the second axiom above implements a special composition of monadic coalgebras and is therefore also a helper construct. Examples for domain specific constructs are constructors for monadic systems, e.g. a constructor for a monadic system with input, which is similar to the `input_system` constructor we have in the Haskell implementation:

---

```

1 // Trans ~ (a,x) -> m x; Cont ~ [a]
2 template<MonadicCoalgebraWithInput Trans, ConstructedType Cont>
3 requires SameType<Trans::Domain::First, Cont::Inner>
4 class Monadic_System_Input {
5     public:
6         typedef Cont Domain;
7         typedef InpMonadicSysCodom<Trans, Cont> Codomain;
8         typedef Concatenation<Cont> Op1;
9
10    Monadic_System_Input(Trans trans) : my_trans(trans) {}
11
12    Codomain operator() (Domain l) {
13        InpMonadicSysCodom<Trans, Cont> ret(my_trans, l);
14        return ret;
15    }
16
17     private:
18     Trans my_trans;
19 };

```

---

The codomain of this monadic system is a class `InpMonadicsysCodom` representing a monadic coalgebra of type  $X \rightarrow MX$  which is constructed out of a function of type  $(T, X) \rightarrow MX$  and a sequence of type  $[T]$ . The type `Monadic_System_Input` models the concept of a monadic system. That is made explicit in the code by a `concept_map`, but we omit it as it adds no new information.

### 3.3 Algorithms

The library contains generic functions which implement the algorithms developed in the formalisation, including the two trajectory computations from Sect. 2. For example, the `macro_trj` function is defined as follows:

---

```

1 macro_trj :: (Monad m) => (i -> (x -> m x)) -> [i] -> x -> [m x]
2 macro_trj sys ts x  =  scanr (\i mx -> mx >>= sys i) (return x) ts

```

---

This function translates to the following C++ implementation:

---

```

1 template <MonadicSystem Mon_Sys, BasicOutputIterator Out_Iter>
2 requires SameType<Mon_Sys::Codomain::Codomain, Out_Iter::value_type>
3 void macro_trj_init(Mon_Sys sys, Out_Iter ret, Mon_Sys::Codomain::Domain x)
4 {
5     typedef Mon_Sys::Codomain MacroState; // MacroState ~ m x
6     MacroState mx = mreturn(x);
7     *ret = mx; ++ret;
8 }
9
10 template <MonadicSystem Mon_Sys, ForwardIterator Inp_Iter,
11           BasicOutputIterator Out_Iter>
12 requires SameType<Inp_Iter::value_type, Mon_Sys::Domain>,
13           SameType<Out_Iter::value_type, Mon_Sys::Codomain::Codomain>
14 void macro_trj(Mon_Sys sys, Inp_Iter controls_bg, Inp_Iter controls_end,
15                  Out_Iter ret, Mon_Sys::Codomain::Codomain mx) {
16
17     while(controls_bg != controls_end) {
18         mx = mbind(mx, sys(*controls_bg));
19         *ret = mx; ++ret; ++controls_bg;
20     }
21 }

```

---

As we can see the Haskell one-liner translates to a much longer version in C++ and it is also split into two functions. The first one, `macro_trj_init`, initialises the macro trajectory data structure and corresponds to the second argument to `scanr` in the Haskell version. The second one, `macro_trj`, is the implementation of the iterative step (the first argument to `scanr`) in the Haskell version. Here it should be noted that we use the C++ iterator mechanism to implement a function which is even more generic than the equivalent Haskell version: in contrast to the Haskell function, where we handle all sequences as Haskell lists, we do not have a fixed sequence type for the input in the C++ version. Besides the two trajectory computations we have also implemented the actual vulnerability computations. The implementation is straightforward but omitted here due to space constraints — the function signature was presented in Sect. 1 and an example use is presented below, in Sect. 4.

## 4 The Library as a Domain Specific Language

What makes the library described in the last section a domain specific language embedded in C++? To clarify this we want to emphasise that the concept “domain specific language” is more of a mindset than a formal definition to us. Therefore we show here by example how the library can be used by experts from the vulnerability community which do not have a deep technical knowledge of C++.

The user can use the library by providing objects of appropriate types to plug them in into the library components we provide. When modelling a system the user has to implement a transition function as a monadic coalgebra which is a rather simple task:

---

```

1 struct InputTransition {
2     typedef std::pair<float, float> Domain;
3     typedef SimpleProbability<float> Codomain;
4
5     Codomain operator() (Domain p) {
6         float x = p.second;
7         float delta = p.first;
8         Codomain sp1(x+delta);
9         Codomain sp2(x-delta);
10        Codomain sp3(sp1,sp2,0.75);
11        return sp3;
12    }
13 };
14
15 concept_map MonadicCoalgebraWithInput<InputTransition> { ... }

```

---

Writing such a function object can be done in an almost algorithmic fashion. First, one has to come up with a name for the function and provide class (in C++, **struct** is a class with default public visibility) for it; next, in this class, there have to be two associated types called **Domain** and **Codomain**<sup>4</sup>, which define the signature of the function<sup>5</sup>; finally, there has to be an application operator: **Codomain operator()** (**Domain** x) { ... }. The **concept\_map** can also be done in a more or less mechanical way. Once such a monadic coalgebra is implemented, it is easy to construct a monadic system out of it. Just create an object of the function type implemented for the monadic coalgebra and use this object in the constructor of an appropriate monadic system type. After being constructed, a monadic system can be used in a vulnerability computation:

---

<sup>4</sup> One must choose appropriate types for a monadic coalgebra: the domain and codomain have to be related by the **MBBindable** concept.

<sup>5</sup> In the current version we only have unary arrows. However, functions which are not unary can be modelled by a unary arrow which has as domain a tuple type.

---

```

1 InputTransition inp_trans;
2 Input_Monadic_System<InputTransition> sys(inp_trans);
3 Harm h;
4 Meas m;
5 vector< vector<float> > controls;
6 // fill it, so that we get
7 // controls = [[3.0], [1.0]]
8
9 Meas::Codomain res = vulnerability_sys(sys, h, m, 3.0, controls);

```

---

In this example we assume that we have function objects (implemented in the way described for the transition) `Harm` for the harm computation and `Meas` for the measure function. We do not list them here because they are not interesting.

The library therefore has some typical features of a domain specific language, in particular

- we use a specific terminology;
- the use of the library is quite easy to learn;
- we abstract from technical details, the user does not have to care about the `MBind` operation or the iterators mechanism inside the trajectory computations.

Besides these feature we presented in Sect. 3 that the library comes along with its own type system (a concept hierarchy) and language constructs (generic classes and algorithms). For these reasons we claim that our library implements a domain specific language embedded into C++.

## 5 Embedding a DSL Using Concepts

In previous sections, we have shown how we developed a generic C++ library for vulnerability assessment based on a high-level, mathematical description of the domain in Haskell (Ionescu, 2009). In this section we outline how a similar process can be applied to other domains, resulting in generic C++ libraries that are efficient but maintain close correspondence with the respective domain. While the process is not fully automated, it is a precise design pattern with parts of it amenable to full automation. Furthermore, parts of the basic framework we have built up for our vulnerability library can be almost directly reused in other developments.

A high-level description, such as the description of vulnerability assessment by Ionescu (2009), consists of several layers:

1. basic mathematical notions on which the description depends;
2. definitions of domain notions;
3. combinators for easy composition of basic building blocks into more complex structures; and
4. domain-specific algorithms that can be applied to the objects in the domain.

In Sect. 3, we describe how each of these layers is designed in our library. In this section, we give a scheme that could be used to derive C++ libraries from Haskell descriptions of other domains. Our examples are from a DSL which is heavily influenced by category theory but the translation scheme could be used also in other cases.

## 5.1 Basic Types

The mathematical description of the vulnerability assessment domain depends on some basic mathematical notions (see Chapt. 2 Ionescu, 2009, for a complete discussion). Some of these basic notions are directly represented in Haskell, others, such as set membership, for example, are implicit; in particular, the representations of the basic notions of functions, functors, and monads must be translated from the Haskell description to a C++ representation. Haskell provides direct support for functions with built-in function types of the form  $X \rightarrow Y$  where  $X$  is the domain of the function and  $Y$  is the codomain. Support for functors is partially provided by *type constructors* which are type functions of kind  $* \rightarrow *$  where  $*$  is the kind of non-constructor types. The *object mapping* part of the functor can be expressed by a type constructor, like `[]` or `SimpleProb`. The *function mapping* part is given by the method `fmap` of the `Functor` class (see Sect. 1.1). Another basic concept used in the vulnerability assessment domain (and elsewhere) is the `Monad` class that describes a specific kind of functors (again, see Sect. 1.1). Next, we discuss how Haskell types are represented in our C++ encoding and we give a scheme that can be used to represent type constructor classes, such as `Functor` and `Monad`.

In C++ parts of Haskell types are represented by type parameters in templates. The structure of Haskell types is encoded using the three concepts introduced in Sect. 3, `Arrow`, `ConstructedType`, and `SameTypeConstructor`, along with the built-in C++ concept `SameType`. First, function types structure is encoded (Alg. 1.), then, type constructors (Alg. 2.), and finally, type constructors and type constructor identities (Alg. 3.).

*Input:* The C++ type  $T$  to be constrained and the Haskell type  $X$  to be translated.

*Output:* A list of concept requirements  $ARR$  encoding the arrow structure of  $X$ .

*Notation:* An application of the *Arrow Structure* algorithm is denoted by  $as(T, X)$ .

**A1.** [Generate arrow requirement.] If  $X$  is a function type of the form  $Y \rightarrow Z$ , generate a requirement `Arrow<T>` and add it to  $ARR$ .

**A2.** [Recursive application.] Generate requirements  $as(Arrow<T>::Codomain, Y)$  and  $as(Arrow<T>::Domain, Z)$ . Add the generated constraints to  $ARR$ .

### Algorithm 1. Arrow Structure

The above translation of types into concept requirements is then used in translation of all other constructs. Algorithm 4. gives the translation scheme for constructor type classes, used to generate C++ representation of Haskell `Functor` and `Monad`.

## 5.2 Domain Notions

Translation of domain notions is mostly an application of the algorithms introduced in the previous section. In the vulnerability assessment domain, every domain notion corresponds to a computation and is described using a Haskell function type. For example, a coalgebra is defined as a function of type `Functor f => x -> f x` and a monadic system is defined as a function of type `Monad m => t -> x -> m x`, where

*Input:* A C++ type  $T$ , the list of arrow constraints  $ARR$  generated by Alg. 1. for  $T$ , and the Haskell type  $X$ , represented by  $T$ .

*Output:* A list of concept requirements  $TC$  encoding type constructors in  $X$ .

**A1.** [Find type constructors.] Scan the type  $X$  for occurrences of type constructor applications.

**A1-1.** If the type constructor application is on the left of a function type arrow ( $\rightarrow$ ), generate a constraint  $\text{ConstructedType}\langle A : \text{Domain} \rangle$ , where  $A$  is the constraint in  $ARR$  corresponding to the arrow.

**A1-2.** If the type constructor application is on the right of a function type arrow ( $\rightarrow$ ), generate a constraint  $\text{ConstructedType}\langle A : \text{Codomain} \rangle$ , where  $A$  is the constraint in  $ARR$  corresponding to the arrow.

**A1-3.** If type constructor is not a part of a function type ( $X$  is just a constructor application) then generate a constraint  $\text{ConstructedType}\langle T \rangle$ .

### Algorithm 2. Type Constructors

*Input:* A C++ type  $T$ , the list of arrow constraints  $ARR$  generated by Alg. 1. for  $T$ , the list of type constructor constraints  $TC$  generated by Alg. 2. for  $T$ , and the Haskell type  $X$  represented by  $T$ .

*Output:* A list of concept requirements  $S$  encoding type and type constructor identities.

**A1.** [Types.] For every pair of types in  $X$  named identically, generate a constraint  $\text{SameType}\langle A, B \rangle$ , where  $A$  and  $B$  are drawn from  $ARR \cup TC$  in the following fashion:

- If  $A$  (or  $B$ ) refers to a type occurring in a type constructor, then  $A$  (or  $B$ ) is  $X : \text{Inner}$ , where  $X$  is the constraint drawn from  $TC$  that corresponds to the constructor application.
- If  $A$  (or  $B$ ) refers to a type on the left of a function type arrow ( $\rightarrow$ ), then  $A$  (or  $B$ ) is  $X : \text{Domain}$ , where  $X$  is the constraint drawn from  $ARR$  that corresponds to the arrow.
- If  $A$  (or  $B$ ) refers to a type on the right of a function type arrow ( $\rightarrow$ ), then  $A$  (or  $B$ ) is  $X : \text{Codomain}$ , where  $X$  is the constraint drawn from  $ARR$  that corresponds to the arrow.

### Algorithm 3. Type and constructor identities

some additional conditions have to be satisfied (see Sect. 2.2). Such domain notions are translated to concepts with a single type parameter, with appropriate concept constraints reflecting the Haskell type of a notion. Both the type structure and the context requirements (context is given before  $\Rightarrow$  in Haskell types) are translated. The first step, of specifying type structure, requires application of the algorithms from the previous section, resulting in an appropriate list of requirements to be included in a domain notion concept. The second step, in which the context is translated, can be itself split into two steps.

First, non-constructor classes are translated to concept refinement. Note, that a description such as that of Ionescu (2009) makes certain context requirement implicit. For example, the definition of monadic coalgebra given by Ionescu states that a monadic coalgebra is also a coalgebra, but this requirement is not explicitly reflected in the type. In our C++ translation, the monadic coalgebra concept indeed explicitly refines the coalgebra concept:  $\text{MonadicCoalgebra}\langle \text{class } \text{Arr} \rangle : \text{Coalgebra}\langle \text{Arr} \rangle$ .

Next, after non-constructor context is translated, constructor classes are translated. Since in our representation constructors can be only “seen” in their applications, we cannot directly translate a context such as `Functor f` or `Monad m`. Instead, our translation includes as many operation requirements (e.g., `FMappable` or `MBindable`) as possible in a given context. For example, the `MonadicCoalgebra` concept includes two monad-related requirements (see Sect. 3.1). Consequently, a client of the `MonadicCoalgebra` concept must explicitly state any monad-related requirements that are not already given in `MonadicCoalgebra`.

*Input:* A constructor type class `Cs` with polymorphic methods.

*Output:* A set `R` of C++ concepts, one for each method in `Cs`.

*Auxiliary operations:*

`gen-name(cname,mname)`, where `cname` is the name of the class `Cs` and `mname` is the name of a method in that class. In our translation scheme `gen-name(Functor,fmap)` generates `FMappable`, for example, but a different naming scheme can be substituted in other situations.

`gen-name(t)`, where `t` is a Haskell type. In our scheme, Haskell types are rewritten in uppercase, type constructors are concatenated with their arguments and rewritten in uppercase, and function types are rewritten to `Arr` and `Arr1`, `Arr2`, and so on, if there is more than one functions that need to be named.

`gen-name(cname)`, where `cname` is the name of a C++ concept. In our translation scheme, `FMappable` generates `FMap`, for example.

**A1.** [Iterate methods.] For each method `m` in `Cs`:

**A1-1.** Generate a concept `Ct` named `gen-name(Cs,m)` with a list of parameters `P`, containing `gen-name(xi)` when `m` has a type of the form `x1 -> ... -> xn` (note that `xi` may be an arrow itself if parentheses occur in the Haskell type) and simply `gen-name(t)` when the type `t` of `m` does not contain arrows.

**A1-2.** Place constraints in the concept `Ct` by successively applying Alg. 1., Alg. 2., and Alg. 3..

**A1-3.** Add an associated function `fn` named `gen-name(Ct)` to the concept `Ct`. The return type of `fn` is the last type parameter in `P` and the rest of type parameters are argument types of `fn` (`void` if `P` contains only one type argument).

**A1-4.** Add `Ct` to `R`.

#### Algorithm 4. Constructor type classes

Finally, in the case of the `MonadicSystem` concept, we translate monadic system laws into C++ concept axioms. Currently, the translation is not easily mechanisable. We expect that laws given as QuickCheck predicates can be mapped almost directly to concept axioms but this is currently a future work item.

### 5.3 Combinators

Combinators are implemented as generic classes, the domain concepts specify the type parameters used to construct these new types. The constructed types are implemented

*Input:* A Haskell function combining two types (which we translated into C++ concepts) into a third one `comb :: X1 -> X2 -> X3`

*Output:* A generic C++ template class for this function `comb`

**A1.** [Template specifications] Write a template class with two template parameters and specify their type by an requirements. Furthermore identify all type identities between the template parameters or associated types of template parameters and state them by `SameType` requirements. Put additional necessary requirements.

**A2.** [Private members] Put two private members of the two template types into the class.

**A3.** [Input Constructor] Implement a input constructor taking two arguments of the two template parameter types copies them.

**A4.** [Class structure] Implement the required structure of the class in order to model the result type of the Haskell function. Put associated types and necessary member operations.

**Algorithm 5.** Translating a Haskell combinator function into a C++ combinator class

as classes which provide the needed infrastructure and operations. In Alg. 5. we give a general translation scheme for combinator operations. In the library this scheme was in particular used to translate different arrow compositions, and domain combinator such as `Monadic_System_Input` (see Sect. 3.2).

The translation scheme application is quite visible in this case. To make explicit that the constructed result type (in this example `Composed_Arrow`) is a model of the required concept, a (straightforward) concept map is needed.

## 5.4 Algorithms

Algorithms are implemented as generic functions, where we ensure proper use of them by specifying their arguments with concepts for domain notions. For every generic function of the Haskell model we implemented at least one generic C++ function (for technical reasons some Haskell functions are translated into more than one C++ function, see for instance the macro trajectory computations).

In Alg. 6. we present how we can translate the signatures of generic Haskell functions into C++ template specifications. The implementation of such functions remains a creative task to the programmer. However, in some cases, for example for the function `vulnerability_sys`, the translation of the implementation is straightforward.

## 6 Related Work

In the realm of generic programming, the use of concepts (or type classes) to abstract from representations of data or to capture certain properties of types is a common practise. For example, the C++ Standard Template Library (STL) (Austern, 1998; Stepanov and Lee, 1994) introduces a hierarchy of concepts for data containers and for iterators used to generically traverse ranges of data, and the Boost Graph Library (Siek et al., 2002) introduces concepts representing different kinds of graphs. In the traditional approach, functional notions such as monadic system, for example, are either

*Input:* A Haskell function with signature  $f :: X_1 \rightarrow X_2 \rightarrow \dots \rightarrow X_n$

*Output:* A generic C++ template specification for this function  $f$

- A1.** [Template type parameters] Introduce a template parameter for each variable in the type of  $f$ .
- A2.** [Template function parameters] Identify all  $X_i$  in  $f$  which are functions and put a template parameter for them.
- A3.** [Template function parameters requirements] For every type parameter which encodes a function type, put a requirement stating its type (`Arrow` or `MonadicCoalgebra` and so on).
- A4.** [Type constructor requirements] Identify all template parameters or associated types of template parameters which are constructed types and put a `ConstructedType` requirement on them.
- A5.** [Type equality] Identify all type identities between template parameters or associated types of template parameters which have to be stated explicitly and put a `SameType` requirement for every necessary identity.
- A6.** [Type constructor equality] Identify all type constructor identities a `SameTypeConstructor` requirement for them.
- A7.** [Type relations] Identify all pairs of (monadic) Coalgebras and template parameters or associated types of template parameters which have to be used in a `fmap` or a `bind` operation and put a `FMappable` or a `MBindable` requirement for them.

#### Algorithm 6. Generic C++ function signature from a Haskell function

not represented explicitly at all (in C++ libraries) or are represented by a particular type (as in Haskell libraries). In our generic library for vulnerability assessment, we take abstraction further in conceptually representing functions as well as data. In Haskell, almost all libraries are generic in the sense that they use and define type classes. Some more advanced examples are (Chakravarty et al., 2005; Claessen and Hughes, 2000; Jansson and Jeuring, 2002; Oliveira et al., 2006)

Recently, new concept applications have been enabled (and inspired) by the linguistic support for concepts in C++0x (Gregor et al., 2006). For example, much of the standard library of C++ has been conceptualised, that is, rewritten using the new concepts feature (see for example Gregor et al. (2008a,b)). This body of code and specifications provides a good example of how generic libraries will be written when concepts officially become a feature of C++. Our work goes further than STL concepts. The first difference is that we systematically tackle type constructor concepts; the only similar concept in STL is the `Allocator` concept (Halpern, 2008), which requires compiler support to verify constraints that are expressed using `SameTypeConstructor` concept in our framework. The other difference is in handling of computations: in our framework a computation is represented by a self-contained `Arrow` concept that knows its domain and codomain types, while in the conceptualised STL a computation is represented by a `Callable` family of concepts that require domain types as concept arguments.

One aspect of our work is implementation of functional programming constructs in C++. Much similar work has been done before, including the Boost Lambda library (Järvi et al., 2003), FC++ library (McNamara and Smaragdakis, 2004), and Boost Fusion library (de Guzman and Marsden, 2009). All the previous approaches rely heavily on template meta-programming, while we encode the types of functions on the level of concepts.

Hudak (1996) proposes the use of domain-specific embedded languages. In this approach, the syntactic mechanisms of the base language are used to express the idioms of a domain; examples of such languages include FPIC (Kamin and Hyatt, 1997), a language for picture drawing, and a robot control language (Peterson et al., 1998). Our library can be considered an embedded language as well but instead of exploiting syntactic mechanisms of the language we rely on the concept-checking mechanisms to facilitate domain idioms.

This paper can also be seen as continuing the line of work relating concepts in Haskell and C++: Bernardy et al. (2008); Garcia et al. (2007); Zalewski et al. (2007).

## 7 Design Questions and Future Work

Our C++ representation of domain notions is based on some specific design choices. In this section, we discuss our representation of functions and domain-level algorithms, as two examples of such choices. For each, we discuss advantages and disadvantages of competing designs. Such consideration is one of the main directions of our future research. Making design choices is not limited to C++; in the last part of this section, we briefly outline how a generic library similar to the one described in this paper could be developed in Haskell.

### 7.1 Functions

Functions are represented by the `Arrow` concept. In this case there is the alternative to represent functions by a concrete datatype, for instance the function type provide by the Boost library<sup>6</sup>. The use of such a function type could make the code at some points more readable and understandable, however we have decided to use no function types for two reasons: first of all, every function type in C++ introduces inefficiency due to the internal use of function pointers. And secondly the use of a specific type would decrease the level of genericity: everyone who wants to use our library would be bound to this specific type. Using a concept instead of a type gives more flexibility. If a user wants to use a specific type for functions he just has to declare this type to be an instance of our arrow concept. Concept maps for common function types could be part of the library in future versions.

Otherwise the design of the concept `Arrow` is not unique. In our version the type of the domain and of the codomain are associated types. One could also make the domain type (or both types) a parameter of the concept. The `std::CallableX` concepts do this:

---

```

1 concept Callable1<typename F, typename T1> {
2     typename result_type;
3     result_type operator()(F&, T1);
4 };

```

---

We have chosen our version because it seems to be closer to mathematical ideas. However there are advantages of the `CallableX` family of concepts, for instance they

---

<sup>6</sup> <http://www.boost.org/>

allow different calling conventions for the same function object. Exploring all the implications of the different design of these two concepts is beyond the scope of this paper and will be subject to future work.

## 7.2 Representation of Domain-Level Algorithms

Domain-level algorithms can be encoded as concepts with associated functions or concepts with associated Arrow types. For instance, the bind operation of monads, `MBBindable`, can be implemented as show in Sect. 3, or with an associated type representing a function of the type `MBind_Arr :: m x -> m y`:

---

```

1 concept MBBindable<class MX, class Arr> {
2   requires ConstructedType<MX>, Arrow<Arr>,
3   ConstructedType<Arr::Codomain>,
4   SameType<MX::Inner,Arr::Domain>,
5   SameTypeConstructor<MX,Arr::Codomain>;
6
7   Arrow MBind_Arr;
8   requires SameType<MX,MBind_Arr::Domain>,
9   SameType<Arr::Codomain,MBind_Arr::Codomain>;
10 };

```

---

An associated function can be directly used in a generic algorithm, while the associated function type requires a constructed object before it can be used. On the other hand, an associated function cannot be easily used in higher-order combinators, where a function object can be passed as a parameter.

## 7.3 From Haskell to C++ and Back Again

Recent additions of support for associated types to the GHC Haskell compiler (Chakravarty et al., 2005; Schrijvers et al., 2008) allows us to express the C++ concept framework also in Haskell (with extensions). We have ported parts of our C++ implementation “back to Haskell” and interesting future work would be to complete this port and compare the expressivity and efficiency of the generated code. This is a sample of the Haskell version using associated types.

---

```

1 class Arrow arr where
2   type Domain arr
3   type Codomain arr
4   (!) :: arr -> (Domain arr -> Codomain arr)

```

---

## References

- C++ Standard, ISO/IEC 14882:2003(E). ANSI-ISO-IEC, ANSI standards for information technology edition (2003)  
 Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, Reading (1998)

- Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A comparison of C++ concepts and Haskell type classes. In: Proc. ACM SIGPLAN Workshop on Generic Programming, pp. 37–48. ACM, New York (2008)
- Brooks, N.: Vulnerability, risk and adaptation: A conceptual framework. Tyndall Center Working Paper 38 (2003)
- Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP 2005: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, pp. 241–253. ACM, New York (2005)
- Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000: International Conference on Functional Programming, pp. 268–279. ACM, New York (2000)
- Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An extended comparative study of language support for generic programming. *J. Funct. Program* 17(2), 145–205 (2007)
- Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 291–310. ACM Press, New York (2006)
- Gregor, D., Marcus, M., Witt, T., Lumsdaine, A.: Foundational concepts for the C++0x standard library (revision 4). Technical Report N2737=08-0247, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (August 2008a)
- Gregor, D., Siek, J., Lumsdaine, A.: Iterator concepts for the C++0x standard library (revision 4). Technical Report N2739=08-0249, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (August 2008b)
- Gregor, D., Stroustrup, B., Widman, J., Siek, J.: Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC1/SC22/WG21 - C++ (August 2008c)
- de Guzman, J., Marsden, D.: Fusion library homepage (March 2009),  
<http://www.boost.org/libs/fusion>
- Halpern, P.: Defects and proposed resolutions for allocator concepts. Technical Report N2810=08-0320, ISO/IEC JTC1/SC22/WG21 - C++ (December 2008)
- Hudak, P.: Building domain-specific embedded languages. *ACM Comput. Surv.*, 196 (1996)
- Ionescu, C.: Vulnerability Modelling and Monadic Dynamical Systems. PhD thesis, Freie Universität Berlin (2009)
- Janssen, M.A., Ostrom, E.: Resilience, vulnerability and adaptation: A cross-cutting theme of the international human dimensions programme on global environmental change. *Global Environmental Change* 16(3), 237–239 (2006) (Editorial)
- Jansson, P., Jeuring, J.: Polytypic data conversion programs. *Science of Computer Programming* 43(1), 35–75 (2002)
- Järvi, J., Powell, G., Lumsdaine, A.: The lambda library: Unnamed functions in C++. *Software: Practice and Experience* 33(3), 259–291 (2003)
- Kamin, S.N., Hyatt, D.: A special-purpose language for picture-drawing. In: Proc. Conference on Domain-Specific Languages (DSL), pp. 297–310. USENIX Association (1997)
- McNamara, B., Smaragdakis, Y.: Functional programming with the FC++ library. *J. of Functional Programming* 14(4), 429–472 (2004)
- Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Nilsson, H. (ed.) Trends in Functional Programming, pp. 199–216 (2006)
- Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 91–105. Springer, Heidelberg (1999)
- Petoukhov, V., Ganopolski, A., Brovkin, V., Claussen, M., Eliseev, A., Kubatzki, C., Rahmstorf, S.: CLIMBER-2: a climate system model of intermediate complexity. Part I: model description and performance for present climate. *Climate dynamics* 16, 1–17 (2000)

- Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
- Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: ICFP 2008: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, pp. 51–62. ACM, New York (2008)
- Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Reading (2002)
- Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, Revised in, as tech. rep. HPL-95-11 (May 1994)
- Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)
- Vandervoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley, Amsterdam (2002)
- Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), pp. 60–76. ACM Press, New York (1989)
- Wolf, S., Lincke, D., Hinkel, J., Ionescu, C., Bisaro, S.: A formal framework of vulnerability. Final deliverable to the ADAM project. FAVAIA working paper 8, Potsdam Institute for Climate Impact Research, Potsdam, Germany (2008),  
<http://www.pik-potsdam.de/favaia/pubs/favaiaworkingpaper8.pdf>
- Zalewski, M., Priesnitz, A.P., Ionescu, C., Botta, N., Schupp, S.: Multi-language library development: From Haskell type classes to C++ concepts. In: Striegnitz, J. (ed.) Proc. 6th Int. Workshop on Multiparadigm Programming With Object-Oriented Languages (MPOOL) (July 2007)

# Domain-Specific Language for HW/SW Co-design for FPGAs

Jason Agron

Dept. of Computer Science and Computer Engineering,  
University of Arkansas,  
504 J.B. Hunt Building, Fayetteville, AR  
jagron@uark.edu

**Abstract.** This article describes FSMLanguage, a domain-specific language for HW/SW co-design targeting platform FPGAs. Modern platform FPGAs provide a wealth of configurable logic in addition to embedded processors, distributed RAM blocks, and DSP slices in order to help facilitate building HW/SW co-designed systems. A technical challenge in building such systems is that the practice of designing software and hardware requires different areas of expertise *and* different description domains, i.e. languages and vocabulary. FSMLanguage attempts to unify these domains by defining a way to describe HW/SW co-designed systems in terms of sets of finite-state machines – a concept that is reasonably familiar to both software programmers and hardware designers. FSMLanguage is a domain-specific language for describing the functionality of a finite-state machine in such a way that its implementation can be re-targeted to software or hardware in an efficient manner. The efficiency is achieved by exploiting the resources found within modern platform FPGAs – namely the distributed RAM blocks, soft-core processors, and the ability to construct dedicated communication channels between FSMs in the reconfigurable fabric. The language and its compiler promote uniformity in the description of a HW/SW co-designed system, which allows a system designer to make partitioning and implementation strategy decisions later in the design cycle.

## 1 Introduction

Modern platform FPGAs provide both a "sea" of logic gates dedicated towards implementation of custom hardware, as well as typical general purpose processors that can be programmed using traditional software techniques [1]. Typically the hardware portion of such a system is described in an hardware design language (HDL) and the software portion is described in a traditional software language, often C or C++. The dichotomy of these two paradigms makes it difficult to program, reason about, and debug systems built on platform FPGAs especially when the pieces of a system are expressed in fundamentally different terms. Unifying these descriptions of software and hardware gives promise to be able to more effectively build and reason about these hardware-/software co-designed systems. Domain-specific languages can be used to construct a uniform level on which to express such systems. FSMLanguage is a domain-specific

language for describing finite-state machines that can be efficiently compiled to either software- or hardware-implementations. The language includes abstractions for memories and communication channels that allow a designer to programmatically describe a system's communication infrastructure while making efficient use of the resources on the platform FPGA.

## 1.1 Motivation

In general, finite-state machines (FSMs) provide a concise vocabulary to describe a set of behaviors using the notion of states, transitions, and actions [2,3,4]. The concept of what an FSM is and does is familiar to both software programmers and hardware designers, thus making it a realistic and well-suited abstraction for describing HW/SW co-designed systems.

The main purpose of FSMLanguage is to provide a common form, or single source language, to describe a HW/SW co-designed system. Programs written in FSMLanguage can be compiled to different implementation targets, currently VHDL and C, and the different targets are able to communicate and synchronize with one another using abstractions built into the language itself. The available abstractions include shared memories, local memories, and FIFO channels all of which are readily available on modern FPGA architectures. A programmer is able to make use of these structures in FSMLanguage without having to understand the low-level protocols and timing requirements normally associated with HW/SW co-design. This has two important, positive effects: (1) it unifies system descriptions - allowing them to be looked at as a cohesive whole, and (2) it eliminates the need for a programmer or designer to manually integrate system components [5].

Currently, many other projects use C-like languages for HW/SW co-design [6,7,8,9]. However, these languages differ widely in how they support generation of hardware implementations. Some languages are merely HDLs that offer a C-like syntax, while others provide true support for mapping software constructs to hardware implementations. In general, all of these languages target HW/SW co-design, but each focuses on different sub-domains within the area.

The "look" of C combined with variance in the amount and function of existing C operators has a tendency to make these languages confusing for both software and hardware programmers. The main problem is that these languages advertise themselves as being a traditional software-like language, however the actual semantics, best practices, and restrictions are in truth very different [10,7].

The C language, while widely considered a high-level general-purpose language, is inherently a domain-specific language for sequential processors; and transforming it into a hardware/software co-design language has proven to be a great challenge [7]. Another approach would be to find a middle-ground between software and hardware development practices. FSM-based design is common in both software and hardware design, which makes it a suitable candidate for HW/SW co-design environments. This is the primary motivation for creating an FSM-based hardware/software co-design environment.

## 1.2 Goal

When compared to general-purpose languages, domain-specific languages (DSLs) are often more narrowly focused and provide a way to concisely describe a limited problem type [11]. The use of a smaller, concise language can lead to a much more clear description of a program or system; leading to a more re-targetable representation, as more high-level information about the "intent" of the program exists. This has 2 major positive effects: (1) it makes it easier for a compiler writer to create multiple, compatible implementations of programs written in a DSL, (2) while also making it easier for a programmer to describe their problem within the context of the DSL [12].

FSMLanguage was developed to provide a common form for describing components in HW/SW co-designed systems. Thus it provides a way for software programmers and hardware designers to program in the same language, while allowing implementations of such programs to target either side of the hardware/software boundary. This eliminates the need for programmers and system designers to create custom middleware to link hardware and software, which is a time-consuming and error-prone task [5]. Additionally, FSMLanguage does not have to be solely used as a tool for programmers. It can be used as a compiler target, or intermediate format, itself. Allowing higher-level imperative and functional languages to become re-targetable to both hardware and software-based implementations.

## 2 Background

An FSM-based description was chosen as it is quite easy for software programmers and hardware designers to comprehend. Additionally it provides a way to write programs that have both sequential (series of states) and parallel (within one state) sections of code. FSMs are also easily visualizable [13] in terms of control-flow graphs (CFGs) and fit very easily into static analysis frameworks (control flow, reachability, liveness, etc.). Many other specification tools have used an FSM-based approach, as it is a natural way of describing systems [14].

DOT2VHDL [15] is a tool that translates graphical representations of FSMs in DOT to VHDL via the KISS2 intermediate format. The DOT descriptions supported by this tool do not support higher-level arithmetic (only pure Boolean assignment) which compared to a modern HLL or HDL, is extremely limiting. FPGA-vendors have developed their own graphical FSM entry tools, such as Xilinx's StateCAD or Altera's Max+II Graphical Editor, however these tools only target hardware implementations of specified FSMs (in the form of VHDL, Verilog, and ABEL). Additionally, none of these tools provide programmer abstractions for using the available embedded IP cores and memory blocks present in modern FPGAs.

StateWorks [16] uses the concept of VFSMs [17,14], or Virtual Finite State-Machines, to model software systems. The StateWorks development environment provides programmers with a way to specify a system composed of sets of VFSMs. The environment also includes a run-time system for executing system models, however StateWorks does not provide a way to generate executable code for use outside of the development environment. Deployment of a StateWorks model requires the use of the StateWorks run-time

system along with additional programmer-specified libraries for I/O and user-interface routines.

AsmL, or the Abstract State-Machine Language, was developed by Microsoft Research [18] for use in system and components modeling as well as executable specifications. The asmL toolset is mainly focused on robust modeling, visualization, and analysis tools, but an increasing number of features geared towards executable specifications are appearing [19]. While asmL programs can now be executed via interpretation in Haskell [18], no public tools exist that produce re-targetable implementations of asmL programs.

These tools are all useful for modeling portions of systems, but modeling does not lead directly to system implementations. On the other hand, FSMLanguage is capable of system modeling as well as targeting *embedded* platforms, namely modern FPGA offerings from Xilinx. This allows programmers and system designers to use FSMLanguage as an implementation tool that has the potential to replace traditional tools that are less flexible. The focus of the aforementioned toolsets encompasses a large area, however this area does not focus on re-targetable compilation, nor does it include the breadth of operators and abstractions that are currently available in platform FPGAs.

## 3 Design

### 3.1 Simplification through Mechanization

Hardware Description Languages, or HDLs, are one of the most prevalent languages used to describe FSMs destined for hardware implementation. Languages such as VHDL, Verilog, and ABEL are capable of behaviorally describing both the interface and function of a FSM. HDL descriptions of FSMs are executable in that they can be run in a simulation environment, or synthesized into an executable component for use in an FPGA or ASIC implementation. HDLs, in general, are designed to allow programmers to describe arbitrary types of hardware components. Therefore most HDLs contain many more features than those required to describe FSMs. Although FSMs are straight-forward to describe in most HDLs, the complexity and verbosity of HDLs can make them cumbersome to use. For instance, adding a single new state variable to an FSM in VHDL requires the following modifications (assumes a 2-process FSM [20]):

- Definition of a new signal (or set of signals – current and next.)
- Modifying the sensitivity list of the synchronous transition process (STP).
- Modifying the sensitivity list of the asynchronous logic process (ALP).
- Adding a default value assignment to the ALP.
- Adding reset behavior and transition behavior for the signal in the STP.

Unfortunately, any errors or omissions made when performing these modifications do not necessarily result in compilation/synthesis errors. However, mistakes can result in incorrect simulation results, improper reset behavior, and improper logic inference (latches vs. registers). Even worse, HDL tool vendors have different requirements for describing FSMs [20,21]. The verbosity of HDLs paired with the chances to make un-noticed coding mistakes makes FSM description a prime candidate for a domain-specific language (DSL). The DSL can improve programmer efficiency by reducing

code size, clutter, and verbosity. Additionally the language can support "correct-by-construction" FSM design, thereby making it impossible to make many of the mistakes that can be made when coding FSMs in HDLs.

### 3.2 The FSM Domain-Specific Language

FSMLanguage is a domain-specific language (DSL) for describing finite-state machines (FSMs). The language, and its associated compiler, targets the configurable logic, embedded memories, and soft-core processors that can be found in modern platform FPGAs [1]. FSMLanguage eliminates the need for a programmer to manually control sensitivity lists, state enumerations, FSM reset behavior, and FSM default output behavior. The language also guarantees that all FSM state variables are correctly inferred as flip-flops which greatly improves the resource utilization and timing characteristics of an FSM. The resulting FSM description is much smaller, and less cluttered, than equivalent code written in an HDL. Additionally, the FSMLanguage compiler is re-targetable – currently capable of producing FSM implementations for software (in C) and hardware (in VHDL). The hardware and software implementations generated by the FSMLanguage compiler are compatible with one another in that the language's built-in abstractions for communication are able to operate across the hardware/software boundary.

The program constructs that allow this form of transparent communication include primitives for declaring and accessing channels and memories. Channels are bi-directional communication channels that can be used to connect FSMs together in a CSP-style framework [22]. Memories are dual-ported array-like structures that can be internal or external to an FSM. An internal memory can only be used by a single FSM, as the FSM has control over both of the memory's ports, while an external memory can be shared among FSMs (each FSM has access to a single memory port). The memory primitives are mapped directly onto the embedded Block RAM (BRAM) components found within platform FPGAs, whereas the channel primitives are implemented using independent FIFO-based Fast-Simplex Links (FSLs) [23]. FSLs and BRAMs are often used within an HDL environment, however in this case, a programmer is responsible for correctly implementing the access protocol required by such components. FSMLanguage provides a clear and easy-to-use syntax for accessing these components, so the programmer need not worry about the access protocols of such components. For instance when interacting with RAMs, one must account for the memory access latency (which may be technology specific) after performing a read operation in order to get a valid result. Also, when reading from a FIFO one must ensure that data exists before continuing on with the rest of the program. These problems can be avoided through mechanization, as FSMLanguage contains special syntactic constructs that can be elaborated by the compiler into the correct access protocols.

### 3.3 Example - Memory Access

FSMLanguage programs specify memory reads/writes using an array-like syntax familiar to both software and hardware designers. This syntax makes the intent of a memory read/write more clear by making the operation wholly visible in a single line of code

within the FSM, as opposed to several lines and states within a traditional VHDL-based FSM as shown in Figure 1. The FSMLanguage syntax is not only more simple, but it also prevents the programmer from making timing and synchronization mistakes. These mistakes are common in VHDL as the programmer is responsible for defining idle states for handling BRAM latencies. It is important to note that the VHDL snippet shown does not include the extra code needed to enumerate extra idle states, nor the default signal assignments involved with de-asserting BRAM control signals.

```
state1 -> state2 where
{
    a'  <= my_mem[x];
}
```

(a) FSMLanguage Memory Read Syntax

```
when state1 =>
    my_mem_addr      <= x;
    my_mem_read_enable <= '1';
    next_state        <= state1_int;

when state1_int =>
    next_state <= state1_final;

when state1_final =>
    a  <= my_mem_data_out;
    next_state <= state2;
```

(b) VHDL Memory Read Syntax

**Fig. 1.** Memory Read Syntax Comparison

### 3.4 FSMLanguage Programs

The structure and syntax of an FSMLanguage program is shown in Figure 2. FSMLanguage programs consist of the following 10 sections:

- State Names - internal names for FSM state variables
- Generics - compile time variables
- Ports - inputs/outputs from/to the outside world
- Connections - permanent connections of output ports to FSM signals
- Memories - internal/external memory blocks
- Channels - FIFO ports to the outside world
- Signals - internal FSM state
- Initial - initial state definition for the FSM
- Transitions - logic/behavior of an FSM
- VHDL - optional section for linking in libraries of native VHDL constructs

The body, or logic, of an FSM is fully described within the *Transitions* section, while the remainder of the sections are used for declarations for the FSM program itself, such as for memories, ports, and local FSM signals. The statements contained inside of a transition must be assignment statements, in which the left-hand side (LHS) is either a signal, a memory, or a channel, and the right-hand side (RHS) is an expression. Expressions can be composed of data accesses: signals, input ports, memories, channels; arithmetic operators: addition, subtraction, multiplication, division-by-a-constant, bit-concatenation (&), and bit-slicing; boolean operators: and, or, not, xor; as well as function calls. Function calls enable FSMLanguage programs to access external libraries of

```

-- **** Internal state signal names ****
CS: <current_state_signal_name>;
NS: <next_state_signal_name>;

-- **** Generics (compile-time vars) ****
GENERICs:
  (<genName>, <type>, <static_value>;)*

-- **** Input/Output ports ****
PORTS:
  (<portName>, <in|out>, <type>;)*
CONNECTIONS:
  (<outputPortName>  <=  <rhs>;)*

-- **** Definitions of memories ****
MEMS:
  (<mName>,
  <dataWidth>, <addrWidth> [,EXTERNAL];)*

-- **** Definitions of FIFO channels ****
CHANNELS:
  (<channelName>, <dataWidth>;)*

-- **** Internal FSM signals ****
SIGS:
  (<sigName>, <type>;)*

-- **** Internal State Definition ****
INITIAL: <stateName>

-- *** Definition of logic/transitions ***
TRANS:
  (<curr_st> [|<bool_guard>] -> <next_st>
  [where
  {
    (<lhs>  <=  <rhs>;)*
  }])*

-- **** Native VHDL Defs. ****
VHDL: <un-parsed VHDL code>

```

**Fig. 2.** FSMLanguage Program Structure and Syntax

code, whether in C or VHDL. This allows programmers to extend the abilities of the language, by encapsulating new operations within VHDL and C functions.

FSMLanguage uses a Mealy machine model in which FSM outputs depend on both the current FSM state as well as the FSM inputs. State transitions are defined as atomic guarded transitions; where the entire body of a transition is executed atomically when the guard statement found to be true. The syntax for defining a state transition can be seen in Figure 3. Multiple guarded transitions can be defined for a single start state so that a single given state can have multiple behaviors based off of inputs or internal FSM state. The transitions are prioritized by the compiler according to their order of appearance in an FSMLanguage program, allowing a programmer to tune the precedence of the transitions. Guard expressions must be boolean, and are not allowed to contain memory/channel accesses, however, the result of such an access can be used within a guard expression.

```

<cur_st> [|<bool_guard>] -> <next_st>
[where
{
  (<lhs>  <=  <rhs>;)*
}]

```

**Fig. 3.** FSMLanguage Guarded Transition Syntax

The *Generics* section allows a programmer to define a set of generics, or compile-time variables, that can be used as static constants throughout an FSMLanguage program. These generics are identical to the generics found in VHDL, which can be used to change the width and size of internal variables and ports at compile time.

The *Ports* and *Connections* sections allow a programmer to define dedicated input and output ports to the outside world. This section is commonly used to provide a way

to connect external inputs and outputs, such as control signals, to an FSM. *Ports* provide external I/O connections to the outside world, while *connections* provide the ability to tie internal FSM signals to output ports. The connections are very similar to concurrent assignment in VHDL in that they allow an output port to be constantly driven by the internals of the FSM. Output ports are not able to be driven directly from within the body of the FSM. Instead, output ports are driven indirectly by a signal that is tied to a given port via a connection.

The *Memories* and *Channels* sections allow a programmer to declare individual memories and channels to be used within the body of their FSM. Memories can be declared with a special *EXTERNAL* keyword to indicate that the memory is shared with another object, therefore only one of the ports of a dual-ported Block RAM will be used by this FSM. The syntax for accessing a location in a memory is identical to the syntax for accessing arrays in C by pairing a memory name with a square-bracketed index. Memory reads and writes are differentiated by the location of the access itself. If the access is on the left-hand side of an assignment statement, then it is a write, while a read would have a memory access on the right-hand side of an assignment statement as shown in Figure 4. Channels also support read/write operations by using a special syntax. Channel accesses are similar to memory accesses, in that the type of access is determined by the location of the access in an assignment statement as shown in Figure 5. Channels allow FSMs to communicate with one another via message-passing, and the channel-specific syntax in FSMLanguage is used to highlight states in which inter-FSM communication occurs. The channel abstraction allows FSMs to be composed in a CSP-style framework [22].

The *Signals* section of an FSMLanguage program is used to define the internal state variables of the finite-state machine. Definitions for a signal include the signal's name and data type as shown in Figure 2. A programmer defines the initial state of the FSM in this *Initial* section. The state name defined as *initial* is used to provide well-defined FSM behavior during reset and start up.

The *VHDL* section is primarily targeted to users wanting to develop hardware-only implementations of FSMLanguage programs. This section is dedicated for hand-written VHDL functions, procedures, and signal definitions. This allows programmers to take advantage of some of the more advanced features of VHDL in a library-like fashion. FSMLanguage supports a function call syntax that allows the body of an FSMLanguage program to call VHDL library functions (such as `conv_std_logic_vector`) defined in the *VHDL* section, VHDL standard libraries, or even in a user-defined library.

```
x' <= m[a] + 1; // Memory read
m[a] <= x + 2; // Memory write
```

**Fig. 4.** FSMLanguage Memory Reads and Writes

```
x' <= #c; // Channel Read
#c <= x + 2; // Channel write
```

**Fig. 5.** FSMLanguage Channel Reads and Writes

A concise state machine definition is formed by combining all of the individual FSMLanguage sections. The FSMLanguage compiler can use this description as input to generate hardware and software implementations of the program, suited for execution on Platform FPGAs.

## 4 Implementation

The FSMLanguage compiler is implemented in Haskell using the Parsec monadic parser library [24,25]. Haskell lends itself very well to language and compiler development as Haskell data structures are able to easily and directly represent BNF-style grammars. The compiler is composed of a parser, a VHDL back-end, a C back-end, and a CFG back-end for producing DOT visualizations of FSMs [26]. Currently, the entire FSMLanguage compiler implementation has been performed in less than 5,000 lines of Haskell code.

### 4.1 VHDL Compiler Back-End

The VHDL back-end for FSMLanguage builds a synchronous FSM in VHDL using a basic 2-process model. The generated code handles reset logic as well as transition logic for the programmer. Additionally, the structure of the generated code ensures that all FSM signals are inferred as registers and not latches which greatly improves the timing of an FSM during synthesis. The memory constructs of FSMLanguage are directly elaborated into the correct read and write access protocols to the Block RAMs (BRAMs) internal to Xilinx Platform FPGAs. The channel constructs of FSMLanguage are transformed into FIFO interfaces that are compatible with the Fast-Simplex Link (FSL) standard from Xilinx [23].

The memory constructs of FSMLanguage are directly elaborated into read/write accesses to the Block RAMs (BRAMs) internal to Xilinx Platform FPGAs. The generated VHDL abides by the BRAM access protocol and is capable of using both ports of dual-ported BRAMs simultaneously allowing parallel reads/writes from the same memory. Additionally, every memory in an FSMLanguage program can be accessed in parallel as these memories are implemented with physically different BRAMs within the FPGA. The BRAMs embedded in Xilinx FPGAs have 2 clock-cycle read latency, and 1 clock-cycle write latency, which allows a programmer to access data with very low overhead, and no jitter.

The channel constructs of FSMLanguage are transformed into FIFO interfaces that are compatible with the Fast-Simplex Link (FSL) standard from Xilinx [23]. These interfaces provide FIFO status signals which aid in the construction of both blocking and non-blocking channel accesses. By default, the FSMLanguage compiler produces blocking reads and writes, however the language does allow a programmer to query a channel interface to see if the channel has data (data exists) or if the channel is full. The ability to query a channel allows a programmer to construct non-blocking read and write operations in an application-specific way.

All VHDL produced by the compiler uses inference templates for instantiated objects such as FSMs, BRAMs, and FSLs. The templates are architecture-independent, behavioral VHDL code that can be altered individually. This allows the compiler to be tuned

for different FPGA architectures, chip families, synthesis tools, as well as changing the layout of generated code.

## 4.2 C Compiler Back-End

The C back-end for FSMLanguage builds a "giant" switch [27] style of FSM. This allows for the C-implementation to accurately reflect the characteristics of an FSM. Namely, that all actions, or assignment statements, within a given state transition appear to happen atomically as they do in the VHDL models of FSMLanguage-based FSMs. Additionally, the "giant" switch allows for an accurate estimation of execution time (measured in clock cycles) of an FSM when implemented in VHDL. While the VHDL models of an FSM are deterministic, the execution of a C model may not be due to a myriad of factors in software-based systems. The code is ascribed with a built-in counter to estimate execution times that can be compared to results from hardware simulation models. The ability to estimate execution times allows programmers who have no knowledge of hardware simulation tools to get an idea of how efficient their program implementation will be when implemented in hardware.

The C back-end is intended to be used with Xilinx's MicroBlaze soft-core processor [28]. This processor contains built-in FSL ports that are accessible via put/get instructions in the MicroBlaze ISA [28]. Compilation directly translates FSMLanguage channel constructs into C-macros that make use of the MicroBlaze's FSL ports. This architecture allows software-based FSMs to interact directly with other FSMs that use the channel abstraction available in FSMLanguage; thus enabling transparent communication across the hardware/software boundary.

The type system of FSMLanguage is currently based directly on an HDL-like type system: composed solely of individual bits (`std_logic`) and arrays of bits (`std_logic_vectors`). The flexibility of this type system along with the ability to express arbitrary bit-arithmetic, bit-slicing, and bit-concatenation can result in extremely complex, inefficient, and hard-to-read C code. This complexity is the direct result of trying to map a flexible, arbitrary bit-width set of operations, into a less flexible, fixed bit-width language. The resulting code must make use of objects (structs) and high-level functions instead of native data types and operations, thus introducing a significant performance overhead.

A less flexible, software-oriented type system would allow for simpler representations of expressions in both C and VHDL, but at the cost of reduced specialization. Future versions of FSMLanguage may use a more software-like type system in order to allow for more efficient software code generation. Currently the C back-end uses native C data types, and does not support arbitrary bit-width operations. A prototype back-end that does support arbitrary bit-width operations through the use of structured data types is currently being developed, and will most likely be transformed into a C++ back-end to make use of operator overloading. Figure 6 demonstrates how arbitrary bit-width operations are handled in both FSMLanguage, VHDL, and C. Note that in the C version, a total of 8 function calls are required in order to pack (box) and unpack (un-box) the objects used to represent arbitrary bit-width values. This style of C code, even with function inlining, is not as efficient as using native C data types. This overhead

```
my_mem[a(16 to 31)] <= my_mem[a(15 to 30)] + 10;
```

(a) FSMLanguage Representation

```
when state0 =>
    my_mem_addr0 <= a(16 to 31);
    my_mem_rENA0 <= '1';
    next_state <= state1;
when state1 =>
    next_state <= state2;
when state2 =>
    my_mem_addr0 <= a(15 to 30);
    my_mem_dIN0 <= my_mem_dOUT0 + 10;
    my_mem_wENA0 <= '1';
    my_mem_rENA0 <= '1';
    next_state <= state3;
```

(b) VHDL Representation

```
bit_vec_t addr0 = get_slice(
    a,
    bit_vec_CREATE(16, 32),
    bit_vec_CREATE(31, 32));
bit_vec_t addr1 = get_slice(
    a,
    bit_vec_CREATE(15, 32),
    bit_vec_CREATE(30, 32));

my_mem[addr0.val] = bit_vec_ADD(
    my_mem[addr1.val],
    bit_vec_CREATE(10, 32));
```

(c) C Representation

**Fig. 6.** Examples of Arbitrary Bit-Width Representation

can be avoided by using the native C compiler and encapsulating all arbitrary bit-width operations inside of hand-written functions that can be linked against at compile-time.

## 5 Experimental Results

The following sections describe some of the experimental results of using FSMLanguage to develop software- and hardware-implementations of finite-state machines. The FSM implementations were all synthesized and tested on a Xilinx ML507 Development Board containing a Virtex-5 FXT-70 FPGA. Each test is evaluated in terms of performance, circuit size (chip area), and code size. While performance and circuit size are the dominant factors in HW/SW co-design, programmer and designer productivity is also important. Productivity can be related to code size in an abstract way; and in these tests, is measured in terms of lines-of-code (LoC). The LoC metric is included in this paper to illustrate how a single language can replace the use of multiple, distinct languages while simultaneously reducing the total amount of code required to describe a system.

## 5.1 Producer/Consumer Example

A simple producer/consumer example, previously illustrated in KIWI [29] and similar to a test in PRET [30], has been constructed to illustrate the use of the communication abstractions available in FSMLanguage. The implementation of each FSM can be targeted to software or hardware while still remaining compatible with other types of FSMs, regardless of their implementation type.

```
-- ****
-- Producer Example
-- ****
-- Generates a set of outputs
-- through a channel
-- ****

CS: current_state;
NS: next_state;

GENERICs:
DWIDTH, integer, 32;      -- Data width

PORTS:

CONNECTIONS:

MEMS:

CHANNELS:
chan1, DWIDTH;

SIGS:
counter, std_logic_vector(0 to DWIDTH-1);

INITIAL: reset;

TRANS:

reset -> initialize

-- Initialize counter to 0
initialize -> genOutput where
{
    -- Initialize counter
    counter' <= ALL_ZEROS;
}

-- Generate 10 outputs
genOutput | (counter < 10) -> genOutput where
{
    -- Increment the counter
    counter' <= counter + 1;
    -- Output the current value
    #chan1  <= counter;
}
genOutput -> halt

-- Halt (remain in halt state forever)
halt -> halt

VHDL:
```

**Fig. 7.** Producer FSMLanguage Program

```

-- ****
-- Consumer Example
-- ****
-- Infinitely consumes inputs
-- and generates outputs
-- ****
CS: current_state;
NS: next_state;

GENERICs:
DWIDTH, integer, 32;      -- Data Width

PORTS:

CONNECTIONS:

MEMS:

CHANNELS:
chan1, DWIDTH;
chan2, DWIDTH;

SIGS:
counter, std_logic_vector(0 to DWIDTH-1);

INITIAL: reset;

TRANS:

reset -> grabInput

-- Grab a value from the input channel
grabInput -> genOutput where
{
    counter'    <= #chan1;
}
-- Generate a new value on the ouptut channel
-- and repeat (loop back)
genOutput -> grabInput where
{
    #chan2    <= counter + counter;
}

VHDL:

```

**Fig. 8.** Consumer FSMLanguage Program

The producer, in this example, generates a stream of integers, starting at zero, that are monotonically increasing until a pre-defined limit is reached, at which time the producer stops executing. The producer's data stream is sent over an FSMLanguage channel so that it can be accessed by the consumer. The job of the consumer is to consistently take data from the producer, multiply this data by 2, and produce an output value on another outgoing channel. If the consumer receives an input from the producer, then it will generate a new output. The consumer will then "block" until another input from the producer is received.

The FSMLanguage programs for the producer and consumer can be seen in Figure 7 and Figure 8 respectively. The channel interface used between the two FSMLanguage programs is an abstraction that is compatible with both the MicroBlaze soft-core processor as well as with custom logic, thus allowing each of the FSM programs to be executed either in hardware or software. The producer/consumer example was tested

**Table 1.** Producer/Consumer Testing Configurations

Producer	Consumer	System Size	Correct?
SW	SW	2,608 LUTs	Yes
SW	HW	1,460 LUTs	Yes
HW	SW	1,508 LUTs	Yes
HW	HW	360 LUTs	Yes

**Table 2.** Lines of Code (LoC) for Producer/Consumer

	Language		
	FSMLang	VHDL	C
Producer	53	273	156
Expansion Factor	5.1x	2.9x	
Consumer	43	277	159
Expansion Factor	6.4x	3.6x	

for correctness in several different configurations listed in Table 1. All configurations correctly execute on a Xilinx ML507 development board. It is important to note that the software-based implementations of the producer and consumer also require extra Block RAM for storing instructions and data that the hardware-based implementations do not require at all.

This benchmark highlights the ability to encapsulate hardware/software interfaces within FSMLanguage. None of configurations require the programmer to make changes to the application code. This allows a programmer to very easily perform design space exploration without requiring reimplementation of the application. FSMLanguage also eliminates the need for a programmer to write driver routines to interact with architecture-specific features such as dedicated memories or communication channels. Instead, a programmer describes such interaction with a special syntax that is then elaborated by the FSMLanguage compiler; thus eliminating this error-prone task from the design and implementation cycle [5].

The producer/consumer example was also evaluated using a lines-of-code metric (LoC) as shown in Table 2. The inversion in both the VHDL and C LoC metric for the producer and consumer relates to the fact that the consumer uses an additional channel interface. The extra channel interface, which requires only 1 LoC in FSMLanguage, requires a multitude of changes to a VHDL program (10 lines of code). These changes affect a VHDL program's port and signal declarations, sensitivity lists, and synchronous state transition process. These changes are similar to the changes required in a C program generated by the FSMLanguage compiler except for those involved in sensitivity lists.

## 5.2 Sorting Benchmark

A benchmark is constructed using a combination of bubblesort and mergesort to sort a large array of integers using the hthreads platform, an operating system designed for hybrid CPU/FPGA environments [31,32]. The purpose of this benchmark is to show

how FSMLanguage can be used to develop custom hardware components to replace software components that are performance critical. In this benchmark the generation of data as well as the merging of sorted data is always performed in software, while the sorting of each "section" of data can be performed in either software or hardware, and with varying numbers of threads.

The sorting application generates a set of random data to be sorted, in this case 1 MB of 32-bit integer data to be sorted in a divide-and-conquer manner. After generating the data, the main thread then spawns a number of sorting threads, either in software or hardware, and feeds data to each thread in an on-demand manner through software-based mailboxes in chunks of 2,048 words of data. Each hardware-based sorting thread has a thin software-based wrapper that performs the mailbox operations and marshals data into and out of the FSMLanguage-based sorting core, as programs written in FSMLanguage are not currently able to use the hthreads OS API calls at this time. The data marshaling done by the wrapper thread involves copying the data to be sorted to a memory-mapped dual-ported BRAM that is connected directly to the FSMLanguage-based bubblesort core, as well as monitoring the go/done control signals emanating from the hardware core. The main thread continues to feed data to the sort threads until all data chunks have been sorted. Next, merging is done on all of the chunks of data and a correctness check is done to make sure that the data has been correctly sorted and merged.

The performance results of the sorting threads in both software and hardware are shown in Table 3. In all tests the instruction and data caches of the CPU, a PowerPC440, were enabled and the compiler optimization level was set to -O2. The body of the FSMLanguage implementation of bubblesort can be see in Figure 9.

The speedup achieved by using multiple hardware threads comes from the fact that the hardware threads can truly execute in parallel, while in a single-CPU system, software threads merely execute pseudo-concurrently in time. Therefore no additional speedup is achieved when using multiple software threads, as they were all being time-multiplexed on a single CPU. The speedup achieved by using a single hardware thread can be understood by comparing the steps taken during bubblesort on the PowerPC440 to the steps taken by the hardware core generated from FSMLanguage.

The control-flow graphs (CFGs) for each application have been generated from their respective executable formats: PowerPC440 assembly language and VHDL. The CFGs shown only represent the "kernel" of the bubblesort algorithm, which is the inner for-loop which iterates over the array while swapping items. Figure 11 represents the CFG of the PowerPC440 assembly implementing the bubblesort routine. When executing on the PowerPC440, this for loop contains 11 instructions when a swap occurs, and 7 instructions when no swap occurs. Given that the PowerPC440 has a 7-stage pipeline

**Table 3.** Sorting Results (1 MB of Data)

	Number of Threads			
	1	2	3	4
<b>SW Exec. Time</b>	18.4 s	18.4 s	18.4 s	18.4 s
<b>HW Exec. Time</b>	9.50 s	4.70 s	3.10 s	2.30 s
<b>Speedup</b>	1.93	3.91	5.93	8.00

```

TRANS:
reset -> idle
idle | (go = '0') -> idle where
{
    stopped'     <= '1';
}
idle | (go = '1') -> begin_sort where
{
    stopped' <= '0';
    n'        <= sort_length;
    n_new'   <= sort_length;
    swapped' <= '1';
}

begin_sort | (swapped = '0') -> halt
begin_sort | (swapped = '1') -> for_loop where
{
    -- Initialize variables before FOR loop begins
    swapped' <= '0';
    i'        <= ALL_ZEROS;

    -- Prefetch the "1st" data1 before the for loop begins
    data1'  <= array[ALL_ZEROS];
}

for_loop | (i >= n) -> begin_sort where
{
    n'    <= n_new;
}
for_loop | (i < n) -> cond_check where
{
    -- Fetch data2, while data1 has already
    -- been calculated during the last iteration
    data2'  <= array[i+1];
}

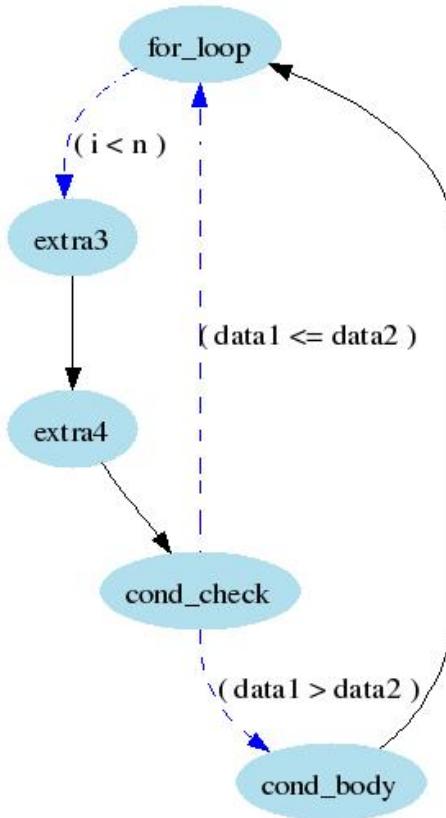
cond_check | (data1 <= data2) -> for_loop where
{
    -- Move value in data2 to data1 for
    -- next pass (now only need to fetch the new data2)
    data1'  <= data2;
    i'      <= i + 1;
}
cond_check | (data1 > data2) -> cond_body where
{
    -- 1/2 update (only one write per state)
    array[i]  <= data2;
}

cond_body -> for_loop where
{
    -- the other 1/2 of the update (only one write per state)
    -- Note, this is the "next" data1 value so keep it around
    array[i+1] <= data1;
    n_new'    <= i;
    swapped'  <= '1';
    i'        <= i + 1;
}

halt -> idle where
{
    stopped' <= '1';
}

```

Fig. 9. BubbleSort FSMLanguage Program

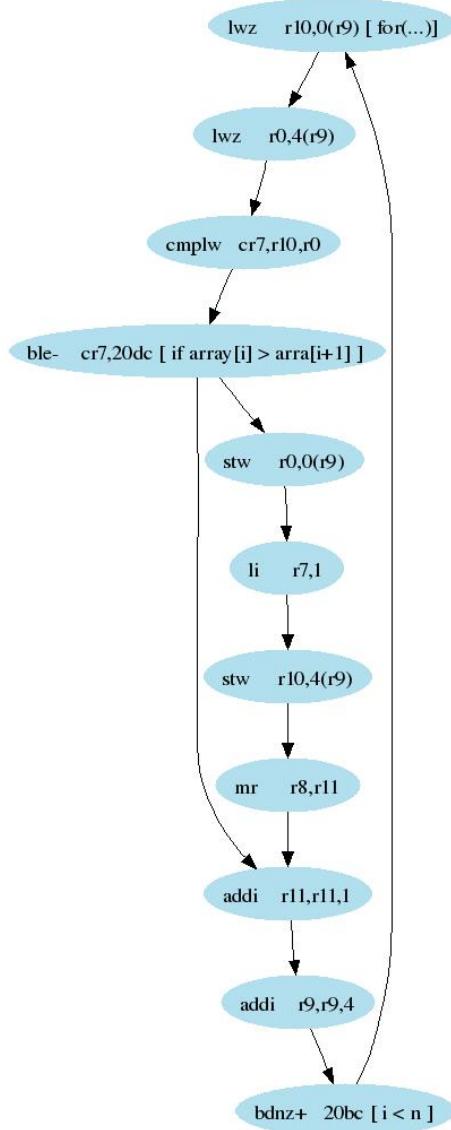


**Fig. 10.** BubbleSort CFG - VHDL from FSMLanguage

this means that a best-case estimate of the execution time of this loop is 14 cycles assuming no pipeline stalls, no cache misses, and no branch mispredictions.

The CFG of the VHDL generated from the FSMLanguage implementation of bubble-sort can be seen in Figure 10. The for loop kernel in this CFG contains 5 state transitions when a swap occurs, and 4 cycles for when no swap occurs. Given that the circuit generated from FSMLanguage is completely deterministic, this means that the worst-case execution time of this kernel is 5 clock cycles, a 2.8x improvement over the PowerPC. The speedup of 1.93 achieved when using a single hardware thread does not meet the ideal figure due to overhead incurred by copying data into and out of the local BRAM attached to each hardware thread.

Additional speedup can be achieved by increasing the number of hardware threads used for sorting, as each of the hardware threads can run concurrently. Each hardware thread operates out of their own local BRAM, which prevents unwanted system bus contention, and leads to an almost linear speedup. The logic overhead incurred from using the hardware-based sorting core is also very minimal as each sorting core only requires 453 slice registers and 582 slice LUTs (look-up tables) on the Virtex-5 FXT



**Fig. 11.** BubbleSort CFG - PPC440 Assembler

chip. This space usage represents only 1% of the capacity available on the FPGA. Additionally, each sorting core requires 8 Block RAMs, which represents only 5% of the Block RAM resources of the Virtex-5 FXT 70 chip. This resource usage is quite small when considering that a MicroBlaze soft-core processor, in synthesized form, requires more than twice as much logic resources to implement on the same FPGA architecture (approx. 1300 slices including memory buses and controllers).

**Table 4.** Lines of Code (LoC) for BubbleSort (C\*\* is a handwritten BubbleSort function in C)

	Language			
	FSMLang.	VHDL	C	C**
Sort	96	365	203	23
Expansion Factor	3.8x	2.1x	0.23x	

**Table 5.** Interpreter Instruction Set

Name	Format	Description
Add	ADD Rd Ra Rb	$Rd = Ra + Rb$
Subtract	SUB Rd Ra Rb	$Rd = Ra - Rb$
Multiply	MULT Rd Ra Rb	$Rd = Ra * Rb$
Logical And	AND Rd Ra Rb	$Rd = Ra \text{ 'and' } Rb$
Logical Or	OR Rd Ra Rb	$Rd = Ra \text{ 'or' } Rb$
Exclusive Or	XOR Rd Ra Rb	$Rd = Ra \text{ 'xor' } Rb$
Shift-Right Arithmetic	SHRA Rd Ra	$Rd = Ra \text{ 'shr' } 1, (\text{MSB' } = \text{LSB})$
Shift-Right Logical	SHRL Rd Ra	$Rd = Ra \text{ 'shr' } 1, (\text{MSB' } = 0)$
Shift-Left	SHL Rd Ra	$Rd = Ra \text{ 'shl' } 1, (\text{LSB' } = 0)$
Load Low	LLOW Rd Imm	$Rd = (Rd \text{ 'and' } 0xffff0000) \text{ 'or' } (\text{Imm' and' } 0x0000ffff)$
Load High	LHI Rd Imm	$Rd = (Rd \text{ 'and' } 0x0000ffff) \text{ 'or' } (\text{Imm' and' } 0xffff0000)$
Jump Equal To Zero	JEZ Rc Ra	If $(Rc == 0)$ Then $PC = Ra$
Load	LOAD Rd Ra Roff	$Rd = \text{MEM}[Ra + Roff]$
Store	STORE Rd Ra Roff	$\text{MEM}[Ra + Roff] = Rd$

A lines-of-code (LoC) comparison of the sorting benchmark can be seen in Table 4. While the FSMLanguage bubblesort program is approximately four times longer than an equivalent handwritten function in C, it is still 3.8 times the size of a VHDL implementation of the algorithm. The C code generated by the FSMLanguage compiler uses a "giant switch" style of FSM coding that is not as compact as handwritten C code.

### 5.3 Interpretation System

Interpreters are easily thought of and implemented as "giant" case/switch statements [27], and as such are easily converted to an FSM description. The purpose of this benchmark is to show how easily an interpreter can be re-targeted when written in FSMLanguage. In this test a small interpreter is written in FSMLanguage that implements the instruction set listed in Table 5. The interpreter is designed to have 3 separate interfaces: (1) a control interface, (2) a memory interface for instructions and data, and (3) a "state" interface used to perform context switching. The internal state of the interpreter is composed of a 256-entry 32-bit memory used as a register file, as well as a 10 other registers used for bookkeeping (instruction decode, intermediate results, program counter, etc.). A snippet of the interpreter's FSMLanguage program is shown in Figure 12. This code snippet highlights the fetch/decode/execute cycle of the interpreter for arithmetic instructions.

The FSMLanguage description of the interpreter is compiled to both software and hardware implementations and tested for correctness using a set of recursive programs (Fibonacci, factorial, and McCarthy91). These programs exercise a majority of the interpreter's control and data path through data/stack manipulation, control flow, and arithmetic operations. The code structure of the interpreter uses HDL-specific bit manipulation routines that are not available in C, so a small library of C routines was created by hand to duplicate this functionality. The language-specific bit manipulation routines can be encapsulated in functions in both C and VHDL so that an FSMLanguage program can make use of the functions in an implementation-independent way. This problem will be solved in the future by outfitting the FSMLanguage compiler with additional support for implementing arbitrary bit-manipulation routines in the generated C code.

The hardware implementation of the interpreter requires a total of 730 slice LUTs, 306 slice registers, and 1 Block RAM for implementing the register file, which is approximately half of the size of the MicroBlaze processor implemented in the same FPGA technology. Another advantage of the hardware implementation is that it operates in a completely deterministic manner. Instruction fetches always take 3 clock cycles (1 cycle read setup, and 2 cycle read latency for BRAM), decode requires an additional 3 cycles as the register file is also implemented using BRAM. Instruction execute and write-back latency is deterministic, but varies for each instruction type. Execution requires 2 cycles for arithmetic operations, 3 cycles for multiplies, 5 cycles for load immediate instructions, 5 cycles for stores, and 7 cycles for loads. This results in a worst-case execution time for an interpreter instruction of 13 clock cycles. The

```
-- **** Fetch Next Instruction ****
fetch | (go = '1' and mode = CMD_INTERPRET) -> decode where
{
    instr' <= prog_mem[pc];
}
--- **** Decode Instruction ****
decode | (opcode_type = TYPE_ARITHMETIC) -> do_arithmetic where
{
    -- Fetch arguments
    a' <= regfile[r_arg_a];      -- Contents of registerA
    b' <= regfile[r_arg_b];      -- Contents of registerB
}
--- **** Execute Stage (ARITHMETIC) ****
do_arithmetic | (opcode = OPCODE_ADD) -> writeback where
{
    regfile[r_dest] <= a + b;
}
do_arithmetic | (opcode = OPCODE_SUB) -> writeback where
{
    regfile[r_dest] <= a - b;
}
.
.
.
-- **** Writeback Stage ****
writeback -> fetch where
{
    -- Increment PC to go to the next instruction
    pc' <= pc + pcInc;
}
```

**Fig. 12.** Code Snippet - Interpreter FSMLanguage Program

fetch/decode cycle of a software-based interpreter requires multiple accesses to memory and often involves bus operations that require an order of magnitude more time (100s to 1000s of cycles) to complete. The fetch/decode process in the hardware implementation is able to make use of a dual-ported BRAM as the interpreter's register file. This architecture allows simultaneous access to the dual-ports of the register file, as found in the decode stage shown in Figure 12. Additionally, the BRAM-based register file and memories makes the hardware implementation fully deterministic, whereas the software implementation has non-determinacy introduced by cache misses and branch mispredictions. Overall, the results of the interpreter benchmark show that a simple FSMLanguage description of an interpreter can be translated into an efficient hardware implementation without requiring a programmer to have detailed knowledge of hardware design techniques.

## 6 Conclusion

FSMLanguage provides programmers with the ability to concisely describe Mealy finite-state machines in a form that allows the code to be targeted to efficient software and hardware implementations. The language and compiler are designed to take advantage of all the resources provided by modern Platform FPGAs; namely custom logic, distributed Block RAMs, soft-core processors, and FIFO channel connections. Individual implementation strategies can be changed for each FSM without affecting overall system operation, as FSMLanguage communication abstractions are able to transparently cross HW/SW boundaries. This makes it much easier for designers to explore their system's design space by eliminating the need to manually re-implement higher-level descriptions of system components.

The three micro-benchmarks demonstrate that FSMLanguage can reduce code size and verbosity, while also providing choices in terms of implementation style, speed, and resource usage. The producer/consumer benchmarks highlights the ability of FSMLanguage programs to cross the HW/SW boundary as well as the ability to make changes to the HW/SW partitioning of a system after initial design and implementation has already occurred. The sorting benchmark highlights the ability to produce efficient hardware implementations of FSMLanguage programs that can be used as application accelerators or co-processors. Finally, the interpreter use-case highlights the ease of re-targeting FSMLanguage programs, and the different features of the software and hardware implementation options.

Overall, the purpose of FSMLanguage is to demonstrate that simple domain-specific languages can be effective for hardware/software co-design for FPGAs. A re-targetable language, such as FSMLanguage, allows a single program specification to be implemented in a variety of ways without forcing a programmer to undergo re-implementation. FSMLanguage permits program specifications to be coded, compiled, debugged, and tested in both a hardware- and software-environment. The hardware and software implementations of FSMLanguage programs remain compatible with one another, allowing the hardware/software partitioning of a system to be altered post-design time.

## References

1. Xilinx: Programmable logic devices, <http://www.xilinx.com/> (last accessed March 25, 2009)
2. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite-State Machines – A Survey. *Proceedings of the IEEE* 84(8), 1090–1123 (1996)
3. Sgroi, M., Lavagno, L., Sangiovanni-Vincentelli, A.: Formal Models for Embedded System Design. *IEEE Design & Test* 17(2), 14–27 (2000)
4. Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Sangiovanni-Vincentelli, A.: A Formal Methodology for Hardware/Software Codesign of Embedded Systems. *IEEE Micro.* 14, 26–36 (1994)
5. Jerraya, A.A., Wolf, W.: Hardware/Software Interface Codesign for Embedded Systems. *IEEE Computer* 38(2), 63–69 (2005)
6. <http://www.systemc.org> (SystemC Last accessed March 25, 2009)
7. Edwards, S.A.: The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers* 23(5), 375–386 (2006)
8. <http://www.impulsec.com> (ImpulseC Last accessed March 25, 2009)
9. <http://www.celoxica.com> (Celoxica Last accessed March 25, 2009)
10. Landin, P.J.: The Next 700 Programming Languages. *Communications of the ACM* 9(3), 157–166 (1966)
11. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4), 316–344 (2005)
12. Spinellis, D.: Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software* 56(1), 91–99 (2001)
13. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3), 231–274 (1987)
14. Wagner, F., Wagner, T., Wolstenholme, P.: Closing the Gap Between Software Modelling and Code. In: *IEEE International Conference on the Engineering of Computer-Based Systems*, vol. 0, pp. 52 (2004)
15. Abdel-Hamid, A., Zaki, M., Tahar, S.: A Tool Converting Finite-State Machine to VHDL. In: *Canadian Conference on Electrical and Computer Engineering*, May 2004, vol. 4, pp. 1907–1910 (2004)
16. <http://www.stateworks.com/active/download/TN2-WhatIsStateWORKS.pdf>: What is StateWorks? StateWorks Technical Document
17. Wagner, F.: VFSM Executable Specification. In: *CompEuro 1992. 'Computer Systems and Software Engineering'*, Proceedings, May 1992, pp. 226–231 (1992)
18. Barnett, M., Börger, E., Gurevich, Y., Schulte, W., Veane, M.: Using Abstract State Machines at Microsoft: A Case Study. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) *ASM 2000. LNCS*, vol. 1912, pp. 367–379. Springer, Heidelberg (2000)
19. Glsser, U., Gurevich, Y., Veane, M.: High-Level Executable Specification of the Universal Plug and Play Architecture. In: *Hawaii International Conference on System Sciences*, vol. 9, p. 283 (2002)
20. <http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf>: Xilinx XST User Guide - Finite State Machine HDL Coding Techniques, pp. 239–245 (last accessed March 25, 2009)
21. [http://www.altera.com/literature/hb/qts/qts\\_qi51007.pdf](http://www.altera.com/literature/hb/qts/qts_qi51007.pdf): Altera Quartus II Handbook – Recommended HDL Coding Styles, ch. 6, pp. 51–57 (last accessed March 25, 2009)

22. Hoare, C.A.R.: Communicating Sequential Processes. *Communications of the ACM* 21(8), 666–677 (1978)
23. [http://www.xilinx.com/support/documentation/ip\\_documentation/fsl\\_v20.pdf](http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf): FSL Bus Product Specification. Fast Simplex Link (FSL) Bus (v2.11a) Data Sheet
24. <http://www.haskell.org> (Haskell Community Website Last accessed March 25, 2009)
25. <http://legacy.cs.uu.nl/daan/parsec.html> (Parsec, A Monadic Parser Library For Haskell Last accessed March 25, 2009)
26. <http://www.graphviz.org/> (Graphviz - DOT and DOTTY Last accessed March 25, 2009)
27. Ertl, M.A., Gregg, D.: The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) *Euro-Par 2001. LNCS*, vol. 2150, pp. 403–412. Springer, Heidelberg (2001)
28. [http://www.xilinx.com/support/documentation/sw\\_manuals/mb\\_ref\\_guide.pdf](http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf): MicroBlaze Processor Reference Guide. Xilinx Datasheet
29. Greaves, D., Singh, S.: Exploiting System-Level Concurrency Abstractions for Hardware Descriptions. *ACM Transactions on Reconfigurable Technology and Systems* 5(N), 1–29 (2008)
30. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable Programming on a Precision Timed Architecture. In: *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems* (October 2008)
31. Andrews, D., Sass, R., Anderson, E., Agron, J., Peck, W., Stevens, J., Baijot, F., Komp, E.: Achieving Programming Model Abstractions For Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16(1), 34–44 (2008)
32. Agron, J., Peck, W., Anderson, E., Andrews, D., Komp, E., Sass, R., Baijot, F., Stevens, J.: Run-Time Services for Hybrid CPU/FPGA Systems On Chip. In: *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)* (December 2006)

# A Haskell Hosted DSL for Writing Transformation Systems

Andy Gill

Information Technology and Telecommunication Center,  
Department of Electrical Engineering and Computer Science,  
The University of Kansas,  
2335 Irving Hill Road,  
Lawrence, KS 66045  
andygill@ku.edu

**Abstract.** KURE is a Haskell hosted Domain Specific Language (DSL) for writing transformation systems based on rewrite strategies. When writing transformation systems, a significant amount of engineering effort goes into setting up plumbing to make sure that specific rewrite rules can fire. Systems like Stratego and Strafunski provide most of this plumbing as infrastructure, allowing the DSL user to focus on the rewrite rules. KURE is a strongly typed strategy control language in the tradition of Stratego and Strafunski. It is intended for writing reasonably efficient rewrite systems, makes use of type families to provide a delimited generic mechanism for tree rewriting, and provides support for efficient identity rewrite detection.

## 1 Introduction

Sometimes its easy to say what you want to do, but tedious to get the scaffolding in place. Such a situation is an ideal candidate for a Domain Specific Language (DSL) where the language provides the scaffolding as a pre-packaged service. For example, scaffolding is needed to promote local, syntax directed rewrites into a global context. Take the first case rewriting rule from the Haskell 98 Report [1] which says:

(a)  $\text{case } e \text{ of } \{ \text{alts} \} = (\backslash v \rightarrow \text{case } v \text{ of } \{ \text{alts} \}) \ e$   
where  $v$  is a new variable

This rule is used to simplify a case over an arbitrary expression, and convert it to a case over a *variable*. We can express this rule directly in Haskell, using a rewrite in the Haskell Syntax provided by Template Haskell [2], using the function `rule_a`.

```
rule_a :: ExpE -> Q ExpE
rule_a (CaseE e matches) = do
    v <- newName "v"
    return $ AppE (mkLamE [VarP v] $ CaseE (VarE v) matches) e
rule_a _ = fail "rule_a not applicable"
```

This rule reflects our syntax rewrite rule almost directly, and cleanly utilizing a monadic name supply for a fresh variable. The rule also acts locally, and needs additional machinery to perform this rewrite on whole programs. Strategic programming is a paradigm which builds on rewrite primitives like `rule_a` by using combinators to construct complex and powerful rewrites and transformation systems.

This paper introduces KURE, a strategic programming DSL hosted in Haskell being developed at the University of Kansas. KURE provides a small set of combinators that can be used to build parameterized term rewriting and general user-defined rule application. In this paper, we discuss in detail the design of KURE, and how we used the tradition of type-centric DSL design to drive our implementation. This paper makes the following specific contributions.

- We move both rewrites and rewrite strategies into a strongly typed world. Throughout the paper we make detailed comparisons with previous attempts to providing typing to rewrite strategy systems.
- More specifically, we provide typing for term traversing strategies using a new capability, a lightweight and customized generic term traversal mechanism implemented using associated type synonyms [3].
- We introduce the ability to record equality over terms, which is traditionally a weakness of purely functional rewrite engines. Equality is important because the use-case for many transformations in practice is to iterate until nothing else can be achieved.
- We abstract our combinators explicitly over a user-defined context, allowing concerns like environment or location to be represented. We show the generality of this contribution by implementing path selection. This design choice allows a particularly clean relationship between the semantics being hosted by the DSL, and the implementation using our DSL.
- We lay out the methodology used in the design of our hosted DSL, with the intent that others can use our design template.

KURE builds on many years of research and development into rewrite strategies, especially the work of Stratego [4] and Strafunski [5], and our own experiences with HERA [6]. We compare KURE to these systems throughout this paper, as well as make different design decisions explicit, and summarize the differences in section 11. In recognition of the contributions made by these systems, we reuse where possible the naming conventions of Stratego and Strafunski.

## 2 Introducing Strategic Programming

Stratego and other strategic programming languages provide a mechanism to express rewrites as primitive strategies, and a small set of combinators that act on strategies, giving the ability to build complex custom strategies. In this section, we introduce the basic combinators of strategic programming used in Stratego. This will guide our design of KURE.

**Table 1.** Combinators in Stratego

Combinator	Purpose
<code>id</code>	identity strategy
<code>fail</code>	always failing strategy
$S \leftrightarrow S$	local backtracking
$S ; S$	sequencing
<code>all(S)</code>	apply $S$ to each immediate child

A basic strategy is a rewrite from one abstract syntax term to another abstract syntax term, for example:

$$\text{NOT1} : \text{Not}(\text{Not}(e)) \rightarrow e$$

This strategy, called `NOT1`, attempts to match a term, and if the top node is `Not`, and the immediate child is also `Not`, then Stratego replaces this whole term with the immediate child of the second `Not` and terminates successfully. If the term is not matched, the strategy fails. In functional parlance, strategies take a term, and return a new term, or fail.

As for parsing combinators [7], a rich algebra of combinators can be built on top of the concept of strategies. If  $S$  is a strategy, the key combinators in Stratego are listed in Table 1. `id` is the identity transformation, `fail` is a transformation that always fails,  $\leftrightarrow$  is a choice operator with local backtracking,  $;$  sequences two transformations, and `all` provides a shallow traversal, a traversal of only the immediate children of a node. There are others, but these capture the spirit of the programming paradigm.

We can use these primitive combinators to write other combinators, like `try`

$$\text{try}(s) = s \leftrightarrow \text{id}$$

which attempts a rewrite, and if it fails, performs the identity rewrite instead. `try` has the nice property, therefore, that it never fails.

All of the combinators introduced so far are shallow, and only act on at most a single level of a term. We can use these to implement deeper rewrites, which act over *every* sub-node in a tree.

$$\text{topdown}(s) = s ; \text{all}(\text{topdown}(s))$$

Stratego also provides the ability to invoke a strategy from within a rule, by enclosing the strategy in angle brackets.

$$\text{EvalAdd} : \text{Add}(\text{Int}(i), \text{Int}(j)) \rightarrow \text{Int}(\langle \text{addS} \rangle(i, j))$$

Here, `addS` is itself a rewrite strategy which takes a 2-tuple of integers, and generates the result of the addition.

By using application inside rules strategy-based programming jumps between rules and strategies, and back again. Basic user strategies are named rules, and rules can use strategies to express rewrites over terms, collectively forming a productive environment to implement complex rewrites. Critically, locally applicable rules are given the opportunity to act over many sub-terms in a controlled matter. Using this programming idiom several rewrite systems have been implemented on top of Stratego and related tools, including Stratego itself, optimizers, pretty printers, and COBOL engineering tools. The idiom has demonstrated the ability to have all the qualities of a great DSL, giving leverage to the rewrite author. We have only given a cursory overview of the essence of Stratego and strategic programming, and the interested reader is referred to the Stratego website, <http://strategoxt.org/>.

When considering a rewriting system using strategies hosted in Haskell, Strafunski is currently the most mature library. Strafunski is a strategic programming implementation which follows in the tradition of Stratego, adds typing to primitive transformations, and uses “scrap your boilerplate” (SYB) generics [8] to implement shallow (single-level) and deep (multi-level) traversals over arbitrary term types. Strafunski provides both type-preserving rewrites and unifying rewrites (rewrites that map to a single common type). However, there are shortcomings that merit revisiting the design decisions of Strafunski. KURE is an attempt to revisit these design decisions. Specifically, KURE replaces the powerful hammer of SYB generics provided in Strafunski with a more precise, user configurable and lightweight generics mechanism. KURE also provides a number of parameterization opportunities over Strafunski, as well as other differences, as discussed in section 9.

### 3 Design of the KURE Kernel

Our design and implementation of KURE follows the classical DSL hosted in Haskell approach, namely

- we propose specific functionality for the primitive combinators of our DSL,
- we unify the combinators around a small number of (typically abstract) types,
- we postulate the monad that is contained inside the computation of these primitives,
- we invent some structure around this monad, to provide the significant user-level types in our DSL,
- and at this point, our combinators are largely implemented using routine plumbing between monadic computations.

There are structures other than monads that can be used to model computations, but monads are certainly the tool of choice in the Haskell community for modeling such things.

The major primitive combinators in a DSL for strategic programming are well understood, and have been discussed in section 2. We want to add two

new facilities, both of which have analogues in Stratego, but are implemented in a functional and strongly typed way in KURE. First, we add the ability to understand the context of a rule. Rather than introduce dynamically scoped rules [9] we want our rules to execute in a readable environment that reflects the context within which the rule is being executed. We also add the ability to create new global bindings from within local rules.

Most importantly, we want our DSL to use types to reflect the transformations taking place. The next section reviews how other systems have added types to strategic programming.

## 4 Previous Uses of Types in Strategic Programming

There have been a number of attempts to add typing to strategy based programming. There are two independent issues to resolve. Firstly, giving a type to a strategy  $\mathcal{S}$ . Secondly, giving general types to functions that do both shallow and deep rewrites, like `all` and `topdown`. We address each of these in turn.

### 4.1 Giving a Type to $\mathcal{S}$

Giving a type to  $\mathcal{S}$  is straightforward. Consider a typed strategy or transformer, called  $\mathcal{T}$ . We can give  $\mathcal{T}$  two type parameters, the initial term's type, and the type of the term after rewriting. Thus, our strategies have the type:

$$\mathcal{T} \ t_1 \ t_2$$

We can now give types to the depthless combinators in Stratego, which we do in Table 2.

**Table 2.** Types for Depthless Combinators

Combinator	Type
<code>id</code>	$\forall t_1. \mathcal{T} t_1 t_1$
<code>fail</code>	$\forall t_1, t_2. \mathcal{T} t_1 t_2$
$\mathcal{S} \leftrightarrow \mathcal{S}$	$\forall t_1, t_2. \mathcal{T} t_1 t_2 \rightarrow \mathcal{T} t_1 t_2 \rightarrow \mathcal{T} t_1 t_2$
$\mathcal{S} ; \mathcal{S}$	$\forall t_1, t_2, t_3. \mathcal{T} t_1 t_2 \rightarrow \mathcal{T} t_2 t_3 \rightarrow \mathcal{T} t_1 t_3$

For combinations of these combinators, the type system works unremarkably. For example, the function `try` can be given a straightforward type.

$$\begin{aligned} \mathbf{try} &:: \forall t_1. \mathcal{T} t_1 t_1 \rightarrow \mathcal{T} t_1 t_1 \\ \mathbf{try}(s) &= s \leftrightarrow \mathbf{id} \end{aligned}$$

Transforms that are type-preserving are common in strategy based programming, so we give such transforms their own type name,  $\mathcal{R}$ .

$$\mathcal{R} \ t_1 = \mathcal{T} \ t_1 \ t_1$$

$\mathcal{R}$  is simply an abbreviation for  $\mathcal{T}$  for notational convenience. Using  $\mathcal{R}$ , **try** can be giving the equivalent type

$$\mathbf{try} :: \forall t_1. \mathcal{R} t_1 \rightarrow \mathcal{R} t_1$$

Giving types to application and abstraction using  $\mathcal{T}$  is straightforward once a monad for rules has been selected, so we defer giving the actual types until section 6 in the context of our KURE monad, and observe that there are no technical issues with their typing.

## 4.2 Types for Shallow and Deep Combinators

Adding types is more problematic for shallow and deep combinators. Consider the combinator **all**, which acts on immediate children, but leave the root node unmodified. This implies a type-preserving rewrite, and we would expect something of the form

$$\mathbf{all} :: \forall t_1. \mathcal{R} t_1 \rightarrow \mathcal{R} t_1$$

This combinator can only apply its argument to immediate children *of the same type*, a significant shortcoming, but is the approach taken by Dolstra [10], when he attempts to host strategies directly in Haskell.

If we give **all** a more general type, other issues surface

$$\mathbf{all} :: \forall t_1, t_2. \mathcal{R} t_1 \rightarrow \mathcal{R} t_2$$

Unfortunately, the type  $t_1$  is completely unrelated to  $t_2$ , so it is well understood that **all** can never actually use its argument in any interesting or non-trivial way, *without runtime type comparisons*. This is the the approach taken by Dolstra [11]; Dolstra and Visser [11,12], who invent a new language with its own type system which includes runtime type-case internally inside **all**.

This is also essentially the approach taken in Strafunski [5]. Rather than work on polymorphic rewrites directly, Strafunski implements **all** by wrapping  $\mathcal{R}$  in an abstraction **TP**. Here, we use the notation from [13] for expressing type contexts.

$$\mathbf{TP} = \forall t_1 \langle \mathit{Term} \; t_1 \rangle \Rightarrow \mathcal{R} t_1$$

This means that **TP** can be used to express any rewrite  $\mathcal{R}$ , as long as the type of the candidate syntax admits *Term*, which is the ability to decompose a data-structure into a universally typed constructor and arguments, and recompose the data-structure back again. Such an approach is not unique to Strafunski. Other typed rewrite systems have also taken the same approach using a universal representation, for example [14]. Using **TP**, we can give **all** the type

$$\mathbf{all} :: \mathbf{TP} \rightarrow \mathbf{TP}$$

Now it is possible to pass rewrites to **all** with an arbitrary type, using SYB style generics [8]. Strafunski provides functions for getting into and out of the **TP** abstraction. Two of the functions for building **TP** are **adhocTP** and **failTP**.

$$\begin{aligned} \mathbf{adhocTP} &:: \forall t_1 \langle \mathit{Term} \; t_1 \rangle \; \mathbf{TP} \rightarrow \mathcal{R} t_1 \rightarrow \mathbf{TP} \\ \mathbf{failTP} &:: \mathbf{TP} \end{aligned}$$

Using these combinators, it is possible to use `all`. We can chain different types of rewrites through our TP abstraction,

```
all ((failTP 'adhocTP' rr1) 'adhocTP' rr2)
```

Here, we have built a TP that can perform two possible rewrites to the immediate children. These two rewrites can be at completely different types. For terms that are not of a matching type, the `failTP` combinator is applied, causing the rewrite to abort with failure.

The use of generics have significant ramifications for our mission of having a strongly typed DSL for rewrites. There are three shortcomings:

- Even though the generic mechanism itself can be implemented efficiently using a type-safe cast, deep traversals, using recursive invocations of `all`, become prohibitively expensive, because the universal-type nature of TP results in every single sub-node being considered as a rewrite candidate. For example, when implementing compiler passes, there is a considerable cost to examining every character of every string as a candidate for rewriting.
- The TP type is universal, and rewrites over two completely unrelated syntaxes use the same abstraction. We want the type of our traversal combinators to at least reflect something about the types they operate on. This issue could be at least partially addressed in Strafunski by creatively using a phantom type [15] on TP.
- TP is not a  $\mathcal{R}$ , therefore not a  $\mathcal{T}$  either. We want to build a combinator library around operators on  $\mathcal{T}$  (our design decision) and each new user-level type complicates the DSL and the abstractions it is attempting to capture.

## 5 Supporting Shallow and Deep Traversals in KURE

What we want to support in KURE is a version of `all` that accepts a set of potentially distinctly typed rewrites. Consider a rewrite function, `all`, to which every relevant correctly typed rewrite is passed explicitly as an element of a tuple.

$$\text{all} :: \forall t, t_1, \dots, t_n \langle t_i \in \text{childrenOf } t \rangle \Rightarrow (\mathcal{R} t_1 \times \mathcal{R} t_2 \times \dots \times \mathcal{R} t_n) \rightarrow \mathcal{R} t$$

where `childrenOf` is a function from a *type* to the *set of types* of possible children. Implementing `all` is now a plumbing problem, and straightforward for any specific  $t$ .

In principle, this idea for implementing `all` works; the rewriting inside `all` knows what constructors are being rewritten, so can select the rewrite of the appropriate type. But this version of `all` has a number of shortcomings. First, the argument to `all` is difficult to generalize, given that it depends on the number of distinct types contained in the constructors of type  $t$ . Second, and more importantly, the argument is no longer first class, in the sense that we are building a framework for manipulating rewrites and transforms, and this is a tuple. It would be *possible* to construct a new sequencing operation over tuples, but this would not make good use of the existing machinery in our DSL.

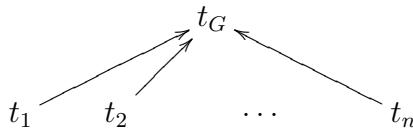
A cleaner and preferable approach is to reuse the  $\mathcal{R}$  rewrite type as the single argument to `all`. As discussed in section 4.2, this argument to `all` can not be parametrically polymorphic, but we can consider how to use our (type) function `childrenOf` to give a useful and implementable type for `all`. The question to be considered is can we encode

$$(\mathcal{R} t_1 \times \mathcal{R} t_2 \times \dots \times \mathcal{R} t_n)$$

inside

$$\mathcal{R} t_G$$

What type is  $t_G$ ? If  $t_G$  was a common super-type of each of  $t_i$ , then this relationship would hold



and it would be possible to encode and decode each of the  $\mathcal{R} t_i$  rewrites inside a single  $\mathcal{R} t_G$ , using this typing relationship. In Haskell, we can model such a typing relationship using sums, so

$$t_G = t_1 + t_2 + \dots + t_n$$

The type of `all` would therefore be

$$\text{all} :: \forall t, t_1, \dots, t_n. \langle t_i \in \text{childrenOf } t \rangle \Rightarrow \mathcal{R} (t_1 + t_2 + \dots + t_n) \rightarrow \mathcal{R} t$$

We can simplify the type of `all`, by inventing a type function,  $\mathcal{G}$ , which maps  $t_i$  to its super-type  $t_G$ .

$$\mathcal{G} t_i = t_G = t_1 + t_2 + \dots + t_n \text{ where } 0 < i \leq n$$

That is, for each of the type  $t_i$ , we can use  $\mathcal{G}$  to look up the generic type,  $t_G$ . Furthermore, if we make  $t$  one of the elements in our sum type, then

$$\mathcal{G} t = \mathcal{G} t_i = t_G = t + t_1 + t_2 + \dots + t_n \text{ where } 0 < i \leq n$$

and we can also use the type  $t$  to look up the type of  $t_G$ .

This gives us a clean type for `all`

$$\text{all} :: \forall t. \langle \mathcal{G} t \rangle \Rightarrow \mathcal{R} (\mathcal{G} t) \rightarrow \mathcal{R} t$$

where  $\langle \mathcal{G} t \rangle$  means that  $\mathcal{G}$  is defined for  $t$ .  $\mathcal{R} (\mathcal{G} t)$  replaces `TP`, as used in Strafunski. Unlike Strafunski, the `all` used in KURE acts over a specific type, not the universal or generic type. In the same way as in Strafunski uses `TP` for deep traversals, we use  $\mathcal{R} (\mathcal{G} t)$  to provide deep traversal combinators in KURE.

We will examine the implementation of  $\mathcal{G}$  and shallow and deep traversals in section 7.

$\mathcal{G}$  is the type function that provides our generic mapping to a local universal type,  $t_G$ . By design, we make available to the DSL user the ability to specify what types can map through  $\mathcal{G}$  to  $t_G$ . In some cases, there might only be a single type that maps to  $t_G$ , and in larger syntaxes, there might be many. This design flexibility allows the KURE user to craft the type scope of the deep traversals, and for example, avoid considering rewriting strings.

The construction of the sum-type and  $\mathcal{R}$  used as an argument to `all` results in a  $\mathcal{T}$  that *can* represent non type-preserving translations, because

$$\mathcal{R}(t_1 + t_2 + \dots + t_n) = \mathcal{T}(t_1 + t_2 + \dots + t_n)(t_1 + t_2 + \dots + t_n)$$

We mitigate against this by providing correct-by-construction promotion functions, and defining translations that are not type-preserving as failed transformations. So a rewrite of type  $\mathcal{R} t$  promoted to a  $\mathcal{R}(\mathcal{G} t)$  never fails because of incorrect typing, and can be safely combined with other promoted translations, at the  $\mathcal{R}(\mathcal{G} t)$  type.

There are other ways of providing deeper traversals, for example, by using fold algebras [16]. Like passing in tuples of  $\mathcal{R}$ , we could provide combinators that take a representation of a (typed) algebra, and perform a fold-like rewrite. In fact, this could be coded up in KURE. We choose, however, to keep our design simple, where everything is a transform.

Finally, there is one `all` like combinator we adopt from Strafunski, the `crush` combinator, which crushes together the results of all the rewrites on the immediate children of a node into one common type. Reusing our type function  $\mathcal{G}$ , this function would have type.

$$\text{crush} :: \forall t. (\mathcal{G} t \text{ and } r \text{ is a Monoid}) \Rightarrow \mathcal{T}(\mathcal{G} t) r \rightarrow \mathcal{R} r$$

## 6 Computations Inside KURE

In KURE, we want to promote small, local rewrites into strategies than can be applied over large syntax terms. We provide a monad, called  $\mathcal{M}$ , for authoring small rewrites, which also provides some basic services, and a larger abstraction  $\mathcal{T}$ , which will contain the services of this monad. So, we propose that our rewrites are of the form

$$\mathcal{T} t_1 t_2 = t_1 \rightarrow \mathcal{M} t_2$$

where  $\mathcal{M}$  is our as yet undefined monad. We choose to put all our services into this monad, but there are other choices.

In order to understand the shape of  $\mathcal{M}$ , we enumerate the capabilities we want to provide in our DSL, towards a generic strategic programming engine in Haskell, and propose how to implement each facility using a specific monad.

- We provide the ability to represent failure, for denoting transformations that fail. This failure does not carry any environment or memory. We implement this using the failure monad.

- We provide the capability to detect an identity transformation. The Stratego family of rewrite systems offers equality for the cost of a pointer comparison, as provided by the underlying term representation. Knowing that a rewrite performed the identity transformation is critical to a number of important strategy idioms, for example finding a fix-point of a transformation. We implement this status using a count of the non-identity preserving translations performed, and we propagate this using the writer monad.
- By design, we allow the construction of rules that have the ability to generate new global bindings in our candidate syntax. We implement this using the writer monad.
- We provide the ability for a rule to understand and have visibility into its context in a parameterizable manner. We implement this using the reader monad.
- Finally, we provide the ability for KURE users to add other arbitrary capabilities, like unique name generation, and others that may be specific to the underlying grammar.

Combining these capabilities is straightforward using monad transformers [17]. After unfolding our monad transformers, we reach a flexible and parameterizable  $\mathcal{M}$  monad

$$\mathcal{M} \alpha = \text{env}_{\text{read}} \rightarrow m((\alpha \times \text{env}_{\text{write}} \times \text{count}_{\text{write}}) + \text{Fail})$$

where  $m$  is another monad. In KURE, we implement the  $\mathcal{M}$  monad directly using this equation. We will return to the shape of  $\text{count}_{\text{write}}$  shortly, but we require both  $\text{env}_{\text{write}}$  and  $\text{count}_{\text{write}}$  to be monoids. We explicitly use the reader and writer monad rather than the state monad to allow the possibility of a concurrent implementation of KURE in the future.

In addition to the monadic operators, we provide two additional operations on  $\mathcal{T}$  and  $\mathcal{M}$ , specifically the coercions between the two structures.

$$\begin{aligned} \text{translate} &:: (t_1 \rightarrow \mathcal{M} t_2) \rightarrow \mathcal{T} t_1 t_2 \\ \text{apply} &:: \mathcal{T} t_1 t_2 \rightarrow t_1 \rightarrow \mathcal{M} t_2 \end{aligned}$$

`translate` promotes a monadic translation into a  $\mathcal{T}$  rewrite, allowing rewrite rules expressed monadically to be constructed. `apply` allows this monadic code to itself make applications of  $\mathcal{T}$  structures; it is the direct equivalent of the ‘`<...>`’ syntax in Stratego.

## 6.1 Counting Translations and the Equality Flag in $\mathcal{M}$

An important question to ask after a term has been translated is “has something changed because of this translation?”. Comparisons over terms should be cheap if this question is asked often. Checking for equality in Stratego requires a straightforward pointer equality because terms are represented using maximal sharing. In KURE, we want to carefully record an equality approximation, specifically the identity rewrites, and the congruence traversals that transitively only perform identity rewrites. Of course, it is possible for the user to write by hand an

identity rewrite and generate a false negative, and we intend to perform future measurements to determine how effective our system is in practice. Providing this equality approximation tracking introduces an important challenge.

- The identity rewrite should be tagged as making no change to its argument.
- Also, every `translate` that contains user-defined code should, by default, be recorded as performing a non-identity translation. That is we assume by default that a function of the form  $t_1 \rightarrow \mathcal{M} t_2$  is not simply returning its argument, but has altered the return value in some way.
- However, some translation rules are *transparent*, in that they perform identity rewrites if the applications of `apply` inside the translation rule are all transitively the identity rewrite. The conceptual model is that the translation is transparent, and non-identities show through the rule specification.

This is the dilemma. We use `translate` to promote for both *leaf* monadic actions, which are not identity translations, and also use `translate` for building the *nodes*, which in many cases are identity preserving if the internal calls to `apply` are also identity preserving. Rather than have two types of `translate`, we provide a new combinator which can mark a specific translation as transparent.

$$\text{transparently} :: \forall t_1, t_2. \mathcal{T} t_1 t_2 \rightarrow \mathcal{T} t_1 t_2$$

By default, `translate` marks the resulting translation as non-identity preserving by adding one to the count of non-identity sub-translations. `transparently` is an adjective which modifies `translate` to not add any additional count value to the non-identity sub-translation count.

Obviously, only an  $\mathcal{R}$  can actually be an identity translation, but many combinators are implemented with their most general type (for example `<+` and `'`), so we want the ability to also mark these translations as transparent. When `transparently` is used in a non-type preserving way, the leaf rewrite that actually does the non-type preserving transformation is marked as non-identity, and therefore the whole transformation is also marked as non-identity.

Our use of this checking for equality has an unfortunate consequence; a `Translate` is not an arrow [18].  $\mathcal{T}$  do not hold for the arrow laws, specifically the law

$$\text{pure id} \ggg arr = arr$$

does not hold where  $arr$  is a transformation that has been marked as identity preserving. This means that KURE can not use the arrow interface or laws.

## 6.2 Implementing Rewrites and Translations

We have two structures,  $\mathcal{T}$ , our transformer, and  $\mathcal{M}$ , our monad, which jointly form the basis of our rewrite system. We used our list of requirements, and the well understood technologies around monads to informally derive a basic implementation for our structures. We now address the question of how we implement  $\mathcal{T}$  and  $\mathcal{M}$  in Haskell to build a pragmatic DSL for defining rewrites by

**Table 3.** Principal Types in KURE

Name	Type	Implementation	Interface
$\mathcal{R} e$	$\text{Rewrite } m \text{ dec } e$	$\text{Translate } m \text{ dec } e \ e$	Synonym
$\mathcal{T} e1 e2$	$\text{Translate } m \text{ dec } e1 \ e2$	$e1 \rightarrow \text{RewriteM } m \text{ dec } e2$	Abstract
$\mathcal{M} e$	$\text{RewriteM } m \text{ dec } e$	$\text{dec} \rightarrow m \ (\text{RewriteStatusM } dec \ e)$	Abstract

introducing our primary Haskell data-types, and various functions that act as combinators on these functions.

In general, KURE provides syntax rewrites by **providing** a library of depthless functions, like ‘;’ and **try**, then **requiring** the user to write shallow traversal combinators for their candidate syntax, then **providing** a library of deep traversal combinator which use the supplied shallow traversal combinators.

Table 3 maps our abstract names onto their type, implementation and interface. **Rewrite** is a synonym to allow combinators like ‘;’ to be shared between **Rewrite** and **Translate**. In Table 3, **m** is a monad and **dec** is the (declaration based) environment. **RewriteStatusM** can represent success or failure, directly reflecting the internals of the type definition of  $\mathcal{M}$ .

Table 4 gives the depthless functions provided by KURE, split into functions that perform a monadic computation, functions that generate a **Translate**, and functions that generate a **Rewrite**, with brief descriptions. Most of these combinator directly inherit the abilities of their Stratego equivalent. There is also functionality for handling identity detection, provided by **changedR**, ‘.+’ and ‘!->’. **changedR** turns an identity rewrite into a failed rewrite, mirroring **tryR** which turns a failing rewrite into an identity rewrite, ‘.+’ which backtracks on identity, and ‘!->’ which only performs the second rewrite after the first rewrite if the first rewrite was a non-identity rewrite.

## 7 Constructing Shallow Traversal Combinators

In this section, we employ the KURE DSL to build a rewrite DSL over a small syntax, adapted from [19, Grammar 3.1]. We distinguish between DSL code written using KURE and the implementation of KURE itself by placing code fragments that are uses of the KURE DSL (or similar) inside boxes.

We are going to build a set of combinators that act over **Stmt** and **Expr**, using **Translate** and **Rewrite**. We can use the predefined KURE combinators to write rewrites immediately.

This rewrite attempts to change right associative additions into left associative additions at a single level.

On top of these rewrites, we provide the opportunity to write congruence and shallow traversal combinators, and then take advantage of deep combinators if the DSL user provides the shallow combinators. In KURE we provide two separate styles of structural rewrites, following the conventions of Strafunski. We

**Table 4.** Principal Monadic, Translate and Rewrite Functions in KURE

Name	Type	Purpose
All functions in this table have context	<code>(Monad m,Monoid d) =&gt; ...</code>	
<code>apply</code>	<code>:: Translate m d e1 e2 -&gt; e1 -&gt; RewriteM m d e2</code>	Translate application.
<code>fail</code>	<code>:: String -&gt; RewriteM m d a</code>	Failure.
<code>liftQ</code>	<code>:: m a -&gt; RewriteM m d a</code>	Lift a monadic operation.
<code>translate</code>	<code>:: (e -&gt; RewriteM m d e2) -&gt; Translate m d e1 e2</code>	Build a <code>Translate</code> from a monadic translation.
<code>rewrite</code>	<code>:: (e -&gt; RewriteM m d e) -&gt; Rewrite m d e</code>	Build a <code>Rewrite</code> from a type-preserving monadic translation.
<code>&lt;+</code>	<code>:: Translate m d e1 e2 -&gt; Translate m d e1 e2 -&gt; Translate m d e1 e2</code>	Local backtracking.
<code>&gt;-&gt;</code>	<code>:: Translate m d e1 e2 -&gt; Translate m d e2 e3 -&gt; Translate m d e1 e3</code>	Sequencing.
<code>failT</code>	<code>:: Translate m d e1 e2</code>	Always failing <code>Translate</code> .
<code>transparently</code>	<code>:: Translate m d e1 e2 -&gt; Translate m d e1 e2</code>	Mark a <code>translate</code> as transparent.
<code>idR</code>	<code>:: Rewrite m d e</code>	The identity <code>Rewrite</code> .
<code>failR</code>	<code>:: Rewrite m d e</code>	The failing <code>Rewrite</code> .
<code>tryR</code>	<code>:: Rewrite m d e -&gt; Rewrite m d e</code>	Attempt a <code>Rewrite</code> , otherwise perform the identity <code>Rewrite</code> .
<code>changedR</code>	<code>:: Rewrite m d e -&gt; Rewrite m d e</code>	Fail if the argument <code>Rewrite</code> has no effect.
<code>.+</code>	<code>:: Rewrite m d e -&gt; Rewrite m d e -&gt; Rewrite m d e</code>	Backtracking on identity.
<code>!-&gt;</code>	<code>:: Rewrite m d e -&gt; Rewrite m d e -&gt; Rewrite m d e</code>	Sequencing for non-identity.
<code>acceptR</code>	<code>:: (e -&gt; Bool) -&gt; Rewrite m d e</code>	Identity or failure, depending on the predicate.

have *type-preserving* rewrites, suffixed with R, and *unifying via a common type* rewrites, suffixed with U. There are other possible translation patterns possible, for examples based on fold [16]. Our objective is to build a set of translations specific to this syntax, common-up these transformation inside a shallow tree traversal `all`-like function, and then host our deep generic tree traversals on top of our `all` combinator.

```

data Stmt = Seq Stmt Stmt
          | Assign Name Expr

data Expr = Var Name
          | Lit Int
          | Add Expr Expr
          | ESeq Stmt Expr

type Name = String

```

```

make_left_assoc = rewrite $ \ e -> case e of
  (Add e1 (Add e2 e3)) -> return $ Add (Add e1 e2) e3
  _ -> fail "make_left_assoc"

```

## 7.1 Supporting Congruence and Friends

It is possible to immediately define direct support for congruence and shallow translations in KURE. As an example, we will construct a congruence combinator and the unifying translation for the `ESeq` constructor. These will have the type.

```

eseqR ::  $\mathcal{R}$  Expr  $\rightarrow$   $\mathcal{R}$  Expr  $\rightarrow$   $\mathcal{R}$  Expr
eseqU ::  $\forall u$  (Monoid  $u$ )  $\Rightarrow$   $\mathcal{T}$  Expr  $u$   $\rightarrow$   $\mathcal{T}$  Expr  $u$   $\rightarrow$   $\mathcal{T}$  Expr  $u$ 

```

Their implementation is tedious but straightforward. The `eseqR` combinator is marked as `transparently` rewriting its target, but the `eseqU` is never an identity transformation. Both combinators use `apply` to invoke the transformation on the arguments of the specific constructor.

```

eseqR :: (Monad m, Monoid dec)
      => Rewrite m dec Stmt
      -> Rewrite m dec Expr
      -> Rewrite m dec Expr
eseqR rr1 rr2 = transparently $ rewrite $ \ e -> case e of
  (ESeq s1 s2) -> liftM2 ESeq (apply rr1 s1)
                    (apply rr2 s2)
  _ -> fail "eseqR"
eseqU :: (Monad m, Monoid dec, Monoid r)
      => Translate m dec Stmt r
      -> Translate m dec Expr r
      -> Translate m dec Expr r
eseqU rr1 rr2 = translate $ \ e -> case e of
  (ESeq s1 s2) -> liftM2 mappend (apply rr1 s1)
                    (apply rr2 s2)
  _ -> fail "eseqU"

```

By jumping into the monadic expression world and using `apply`, writing such congruence combinators is mechanical and prone to cut-and-paste induced errors. The type of the constructors dictates the structure of the implementation of all the functions. The KURE user has the option of being supported in this task with a Template Haskell library, called “KURE your boilerplate” (KYB) which generates these congruence functions automatically from the data structures, performing the tedious task of writing these functions. There may also be good reasons why a library author might want to write these by hand, as we will discuss in section 9, but KYB certainly makes it easier to get KURE working for any specific grammar.

## 7.2 Supporting Generic Shallow Term Traversals

How do we support generic traversals over a syntax tree? In the previous section we implemented congruence, which was supported using multiple `Rewrite` or `Translate` arguments, of potentially different types, one per argument of the individual constructor. We want to build generic traversals that take a single `Rewrite` or `Translate`. We discussed `all` in section 5, where we gave it the type

$$\text{all} :: \forall t \langle \mathcal{G} t = t + t_1 + t_2 + \dots \rangle \Rightarrow \mathcal{R} (\mathcal{G} t) \rightarrow \mathcal{R} t$$

The traditional way of capturing the type function  $\mathcal{G}$  that is a mapping from  $t$  to the sum of possible sub-children type in Haskell is multi-parameter type classes and functional dependencies [20]. However, a more natural implementation route is now possible; we can directly use a type function, as implemented in experimental extension to Haskell called associated type synonyms [3]. We now introduce the relevant part of associated type synonyms, and use it to implement the generic rewrite scheme for KURE.

Type families allow the introduction of `data` and `type` declarations inside a `class` declaration.

```
class Term exp where
  type Generic *
  -- | 'select' selects into a 'Generic' exp,
  --   to get the exp inside, or fails with Nothing.
  select :: Generic exp -> Maybe exp

  -- | 'inject' injects an exp into a 'Generic' exp.
  inject :: exp -> Generic exp
```

This `class` declaration creates a type function, `Generic` which looks like a type synonym inside the class – it does not introduce any constructors or abstraction – but actually provides a configurable function from a type to another type. Unlike traditional type synonyms `Generic` is only defined for specific instances, specifically instances of our class `Term`. We have two utility functions. `select` takes one of our `Generic` type, and selects a specific type `exp`, or fails. `inject` creates something of the `Generic` type, and can not fail.

It is now easy to find the Generic type for a supported type, simply using `Generic`. For our example, we choose to create a new data-type `OurGeneric`, though we could have chosen to use the Haskell type `Either`.

```
data OurGeneric = GStmt Stmt
                 | GExpr Expr

instance Term Stmt where
  type Generic Stmt = OurGeneric
  inject = GStmt
  select (GStmt stmt) = Just stmt
  select _ = Nothing

instance Term Expr where
  type Generic Expr = OurGeneric
  inject = GExpr
  select (GExpr expr) = Just expr
  select _ = Nothing
```

This gives the type equalities

$$\text{Generic Stmt} = \text{Stmt} + \text{Expr} = \text{Generic Expr}$$

Following Strafunski, as well as providing an `all` combinator in KURE, we also provide a `unify` function, which requires all (interesting) children to be translatable to a single, unified, monoidal type. Following our naming conventions, we call these two variants `allR` and `crushU`. We can now give their interface using class overloading.

```
class (Monoid dec, Monad m, Term exp) => Walker m dec exp where
  allR :: Rewrite m dec (Generic exp)
        -> Rewrite m dec exp
  crushU :: (Monoid result)
        => Translate m dec (Generic exp) result
        -> Translate m dec exp result
```

`Walker` is a multi-parameter type class, which signifies the ability to walk over a specific type. It requires that the type be an instance of type `Term`, which will provide our generics machinery. `allR` applies the `Generic` rewrites to all the chosen children of this node. `crushU` applied a `Generic Translate` to a common, monoidal result, to all the interesting children of this node.

Table 5 lists the functions that use the `Term` typeclass to provide various promotion and extraction functions over the the `Translate` type, as well as `allR` and `crushU`. `Term` is implicit in the type of `allR` and `crushU`, because `Walker` is a subclass of `Term`.

If we have provided a complete set of congruence operators (section 7.1), then providing the implementation for `allR` and `crushU` becomes straightforward. For `Expr`, our instance can be written as

**Table 5.** Shallow Translate Combinators in KURE

Name	Type	Purpose
All functions in this table have context <code>(Monad m,Monoid dec,...) =&gt; ...</code>		
<code>allR :: (..., Walker m dec e)</code> <code>-&gt; Rewrite m dec (Generic e)</code> <code>-&gt; Rewrite m dec e</code>		Apply the argument to each immediate child.
<code>crushU :: (..., Walker m dec e, Monoid r)</code> <code>-&gt; Translate m dec (Generic e) r</code> <code>-&gt; Translate m dec e r</code>		Apply the argument to each immediate child, to build something of a unified type.
<code>extractR :: (..., Term e)</code> <code>-&gt; Rewrite m dec (Generic e)</code> <code>-&gt; Rewrite m dec e</code>		Extract a specific <code>Rewrite</code> from our generic <code>Rewrite</code> .
<code>extractU :: (..., Term e)</code> <code>-&gt; Translate m dec (Generic e) r</code> <code>-&gt; Translate m dec e r</code>		Extract a specific <code>Translate</code> from our generic <code>Translate</code> , where both share a common (unifying) target type.
<code>promoteR :: (..., Term e)</code> <code>-&gt; Rewrite m dec e</code> <code>-&gt; Rewrite m dec (Generic e)</code>		Create a generic <code>Rewrite</code> out of a specific <code>Rewrite</code> , where all other types fail.
<code>promoteU :: (..., Term e)</code> <code>-&gt; Translate m dec e r</code> <code>-&gt; Translate m dec (Generic e) r</code>		Create a generic <code>Translate</code> out of a specific <code>Translate</code> , where all other types fail.

```
instance (Monad m,Monoid dec) => Walker m dec Expr where
  allR rr = varR
    <+ litR
    <+ addR (extractR rr) (extractR rr)
    <+ eseqR (extractR rr) (extractR rr)
  crushU rr = varU
    <+ litU
    <+ addU (extractU rr) (extractU rr)
    <+ eseqU (extractU rr) (extractU rr)
```

One caveat is we do need to make sure we successfully accept all constructors, including the ones with no interesting arguments, otherwise `allR` and `crushU` will unexpectedly fail on these constructors.

## 8 Providing Deep Generic Traversal Combinators

We reach a significant issue when we try to implement deep traversals. Our shallow depth rewrite combinators, like `allR` act on *specific* types, but we want to rewrite anything in our `Generic` type sibling set. If we consider writing a

topdown rewriter that applies a rewrite at every node in a top-down manner, we appear to want a function of type

```
topdownR :: (Walker m dec e)           -- INCORRECT TYPE, ATTEMPT 1
  => Rewrite m dec (Generic e)
  -> Rewrite m dec e
topdownR rr = extractR rr >-> allR (promoteR (topdownR rr))
```

We are confident that `Generic e` contains all the possible interesting sub-children. However, attempts to compile this function lead to type failures, even though this appears to be a reasonable implementation. We extract the rewrite at the current type, apply it, then rewrite all the children, using `topdownR` recursively. However, this function is unresolvably ambiguous. The `promote` allows a *single* rewrite type to be promoted, and critically, this type is not determinable at compile time for associated type synonyms. This definition actually type-checks for associated *data-types*, but the use of associated type synonyms is central to our lightweight generic mechanism.

We need to take a different approach. We could take the type from `topdown` (and `all`) in Strafunski, which performs the whole traversal at the universal type. In KURE, using our typed ‘universal’ type, `topdown` would transliterate into

```
topdownR :: (Walker m dec e)           -- INCORRECT TYPE, ATTEMPT 2
  => Rewrite m dec (Generic e)
  -> Rewrite m dec (Generic e)
topdownR rr = rr >-> allR (topdownR rr)
```

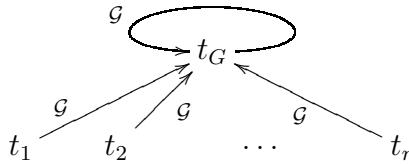
We are now operating at the generic type, hiding any extraction inside `rr`. However, the type of `allR` takes a rewrite of type `Generic e` and returns a rewrite over `e`. This fails because `Generic e` does not have the ability to invoke `allR` directly.

The key to getting `topdownR` working is empowering `Generic e` with the same traversal abilities as `e`, specifically the ability to use `Generic` to find the generic type. This means that for this function to work, `Generic e` and `e` must be of same type. We can achieve this by using the `~` operator provided by associated types, and augmenting the context of deep traversal with the aesthetically pleasing

```
(..., Generic e ~ e) =>
```

which means literally that `Generic e` and `e` must unify.

Diagrammatically, this means that we now have the following coercion relationship provided by our generic type function,  $\mathcal{G}$ .



Retrospectively this result is not surprising, in the sense that `Generic` is being used as a type coercion, and in a custom DSL implementation, the identity coercion would be trivial. Hosting KURE in Haskell has forced the creative use of the type system. Taking this into consideration, we can now give our type and implementation of `topdownR`, which is

```
topdownR :: (Generic e ~ e, Walker m dec e)
          => Rewrite m dec (Generic e)
          -> Rewrite m dec (Generic e)
topdownR rr = rr >-> allR (topdownR rr)
```

Like in Strafunski, we still return a `Rewrite` over our universal type, and we can safely use `extract` when invoking `topdownR` to build a topdown rewrite strategy at the required type. We can also write the type of `topdownR` as

```
topdownR :: (Generic e ~ e, Walker m dec e)
          => Rewrite m dec e
          -> Rewrite m dec e
topdownR rr = rr >-> allR (topdownR rr)
```

but prefer the former because it expresses explicitly that we are operating over our `Generic` type.

Table 6 gives a list of the provided deep traversal combinators, all of which use the `~` trick, and have straightforward implementations.

**Table 6.** Deep Translate Combinators in KURE

Name	Purpose
<code>topdownR</code>	Apply a rewrite in a top down manner.
<code>bottomupR</code>	Apply a rewrite in a bottom up manner.
<code>alltdR</code>	Apply a rewrite in a top down manner, pruning at successful rewrites.
<code>downupR</code>	Apply a rewrite twice, in a top down and bottom up way, using one single tree traversal.
<code>innermostR</code>	A fixed point traversal, starting with the innermost term.
All these functions have type	
<code>(Generic e ~ e, Walker m dec e)</code> <code>=&gt; Rewrite m dec (Generic e)</code> <code>-&gt; Rewrite m dec (Generic e)</code>	

In order to provide this reflective `Generic` ability, we need to provide an instance for `Generic e`, for each `Generic e`. To continue our running example, we can have to add an `OurGeneric` instance for both `Term` and `Walker`.

```
instance Term OurGeneric where
  -- OurGeneric is its own Generic root.
  type Generic OurGeneric = OurGeneric
  inject    = id
  select e  = Just e

instance (Walker m dec Stmt, Walker m dec Expr, Monad m, Monoid dec)
  => Walker m dec OurGeneric where
  allR rr = transparently $ rewrite $ \ e -> case e of
    GStmt s -> liftM GStmt $ apply (allR rr) s
    GExpr s -> liftM GExpr $ apply (allR rr) s
  crushU rr = translate $ \ e -> case e of
    GStmt s -> apply (crushU rr) s
    GExpr s -> apply (crushU rr) s
```

Again, this construction is tedious, but performed only once for each new type of syntax tree. KYB also generates the instance of `Walker` automatically for the `Generic` data-type, literally curing more of this boilerplate painfulness. Unfortunately, KYB does not generate the instance for `Term` because of limitations of the current implementation of Template Haskell.

Though this generic system have proven both useful and sufficient in practice, there are significant technical limitations with our generic mechanism. Specifically, we assume a completely monomorphic syntax tree without any parameterization, and KYB enforces this limitation when generating boilerplate code.

## 9 Using KURE Extensions

KURE is intended to help build rewrite engines, based on strategic programming technologies. KURE threads a user-defined local environment though all its transformations and rewrites. This environment must be a monoid. Often, this environment would be  $\langle \rangle$ , meaning that the rules are being evaluated in context free manner. Consider an operational semantics where the rules have the form

$$C \vdash E \rightarrow C \vdash E'$$

$C$  itself does not change, and is read-only. In KURE, we can model  $C$  using our environment. For example,  $C$  might be the path through the rewrite tree.

$$C ::= \text{root} \mid \langle n \rangle C$$

We can construct  $C$  using the following Haskell code, and define a version of `apply` that records what edge number we are traversing when applying a `Translate`.

```

data C = C [Int] deriving (Monoid)

applyN :: (Monad m)
        => Int -> Translate m C e1 e2 -> e1 -> RewriteM m C e2
applyN n rr e = mapDecsM (\ (C xs) -> C (n:xs)) $ apply rr e

```

Using `applyN`, we can rework our congruence methods to use our context, `C`. For example `eseqR` would be rewritten

```

eseqR :: (Monad m)
       => Rewrite m C Stmt
       -> Rewrite m C Expr
       -> Rewrite m C Expr
eseqR rr1 rr2 = transparently $ translate $ \ e -> case e of
    (ESeq s1 s2) -> liftM2 ESeq (applyN 0 rr1 s1)
                      (applyN 1 rr2 s2)
    _ -> fail "eseqR"

```

This `eseqR` performs all the services over `Expr` that the original `eseqR` did, but also updates the environment to record the path taken by any tree walker that uses `eseqR`. With this new location facility provided by the new `eseqR` (and siblings) we can write a combinator that rewrites at a specific, unique node, using the path through the term tree to find the node.

```

rewriteAtR :: (Walker m C e, Monad m, Generic e ~ e)
            => [Int] -> Rewrite m C (Generic e) -> Rewrite m C (Generic e)
rewriteAtR [] rr = rr
rewriteAtR (p:ps) rr = allR (getDecsT $ \ (C (x:xs)) ->
    if x == p then rewriteAtR ps rr else idR)

```

This performs a single rewrite, potentially deep inside the term tree, and gives a small flavor of the flexibility and power of the combinators provided by KURE.

## 10 Performance

In this section we perform some preliminary measurements to evaluate the cost of the flexibility that KURE provides. We know from experience that KURE is reasonably efficient on small and medium sized examples, but for a simple evaluation we implemented Fibonacci using both top-down and bottom-up rewrites in five different systems over a common term representation. In KURE, we implemented Fibonacci using the rewrite `fibR`.

We also implemented basic arithmetic reductions, as a separate rewrite, and repeatedly use a deep rewrite combinator until no more rewrites can be performed. For bottom-up rewriting, we expressed this using

```

fibR :: Rewrite Id () Exp
fibR = rewrite $ \ e -> case e of
  Fib (Val 0) -> return $ Val 1
  Fib (Val 1) -> return $ Val 1
  Fib (Val n) -> return $ Add (Fib (Dec (Val n)))
                           (Fib (Dec (Dec (Val n))))
  _ -> fail "no match for fib"

```

```

evalFibR :: Rewrite Id () Exp
evalFibR = repeatR (bottomupR (tryR (fibR <+ arithR)) .+ failR "done")

```

Table 7 gives the preliminary results of our five implementations, measuring wall-clock time in seconds on a 2.5GHz Intel Mac laptop. Our first two implementations use strategies that can never fail, implemented as simple endomorphic functions. The final three implementations use strategies that can encode failure.

All the implementations use the same basic term rewriting algorithm. The first implementation (Tree Rewrite) uses a straightforward and explicit tree traversal directly coded in Haskell. The second implementation (SYB) used SYB to provide the deep tree traversal. Both of these implementations do not allow for the encoding of failure. The SYB-Maybe and the StrategyLib implementations are both transliterations of the KURE solution. SYB-Maybe encodes failure using the Maybe monad. StrategyLib is a recent version of Strafunski available on [hackage.haskell.org](http://hackage.haskell.org). All implementations except KURE compare the result terms after each deep traversal for equality to find a fix-point, while KURE uses the built-in equality rewrite checker, via the ‘.+’ combinator.

Clearly, KURE is the slowest implementation, and the DSL users pay a cost for the various extra capabilities and counters that KURE provides, up to an order of magnitude of wall clock time over Strafunski. Further investigation will reveal if the costs of KURE can be reduced by using new optimizations inside the Glasgow Haskell compiler. It would be nice if there was a way to completely eliminate the cost of the extra capabilities in KURE which are not being utilized. This brings KURE full circle; it was written to investigate exactly

**Table 7.** Fibonacci Implemented Using Rewrites

Family	Test	Top Down			Bottom Up				
		fib	20	25	30	fib	20	25	30
<b>No Failures</b>	Tree Rewrite		0.018	0.157	2.024		0.010	0.075	0.987
	SYB		0.074	0.844	9.452		0.044	0.377	4.298
<b>Can Fail</b>	KURE		0.363	4.040	45.531		0.434	5.731	66.451
	SYB+Maybe		0.054	0.559	6.256		0.050	0.513	5.764
	StrategyLib		0.119	1.376	15.385		0.060	0.546	6.080

such a transformation [21], inside a more controlled setting than the Glasgow Haskell compiler.

## 11 Related Work

We have already surveyed in some detail two strategic programming systems, Stratego and Strafunski. The widest used and most mature strategy based rewrite system is Stratego [4], which grew out of the work on building a strategy language to translate RML [22], and drew inspiration from ELAN [14], a specification formalism. Stratego is a untyped language for expressing both rewrites and rewrite systems. Strafunski [5] is an implementation of strategic program in Haskell which gives types to strategies, and uses “scrap your boilerplate” generics to provide a universal type, and thus shallow and deep rewrites. Visser [23] is a useful resource as an overview of the discipline.

There have been many, many syntax translators written in Haskell. The Glasgow Haskell compiler itself is an example of a large rewriting optimizer. Another general framework, also being developed at the University of Kansas, is InterpreterLib [24], which uses Modular monadic semantics [25] as its basis for building algebra combinators over co-algebras. There are also systems that express rewrites as extensions to mainstream languages, including Java [26].

## 12 Conclusion and Further Work

We have hosted a strategy based rewrite system in Haskell, using the well understood principles of thinking in terms of types and computations, before implementation. Haskell as a host language worked remarkably well, especially the recent extension for associated type synonyms which were invaluable for finding types for our deeper traversal combinators. KURE as a DSL is clearly aimed at existing Haskell programmers, and requires a significant level of comfort with Haskell before being able to be productive in KURE. This same issue exists with KURE-enabled DSLs that export specific functionality for rewriting a specific syntax. In summary this experiment supports the long held belief that hosted DSLs are great for the users of the host language, and for understanding the intrinsics of a specific DSL, but of questionable general purpose use, except to guide a future stand-alone language.

KURE as a system is in use at the University of Kansas, and forms a base tool for several exploration experiments into rewriting and optimizations. The new `Generic` mechanism works well in practice, and was a great application of associated type families, but has a number of significant limitations. We will look to lift these restrictions as issues arise in real grammars. We also want to explore the use of Haskell’s quasi-quoting facility [27] which will allow rewrite rules to be expressed directly in the target syntax as an alternative to working in abstract syntax.

## Acknowledgments

I would like to thank the members of the Computer Systems Design Laboratory at the Information and Telecommunication Technology Center at the University of Kansas for providing a creative research environment. Specifically, I would like to thank Perry Alexander, Prasad Kulkarni, Vijayanand Manickam, Garrin Kimmell, Nicolas Frisby, as well as Adam Proctor of the University of Missouri. I would also like to thank the referees, for their many detailed and useful suggestions.

## References

1. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge (2003)
2. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In: Chakravarty, M.M.T. (ed.) ACM SIGPLAN Haskell Workshop 2002, pp. 1–16. ACM Press, New York (2002)
3. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP 2005: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, pp. 241–253. ACM, New York (2005)
4. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
5. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 137–154. Springer, Heidelberg (2002)
6. Gill, A.: Introducing the Haskell Equational Reasoning Assistant. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, pp. 108–109. ACM Press, New York (2006)
7. Hutton, G., Meijer, E.: Monadic parsing in Haskell. Journal of Functional Programming 8(4), 437–444 (1998)
8. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices 38(3), 26–37 (2003); Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)
9. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae 69(1–2), 123–178 (2006)
10. Dolstra, E.: Functional stratego. In: Visser, E. (ed.) Proceedings of the Second Stratego Users Day (SUD 2001), pp. 10–17 (2001)
11. Dolstra, E.: First-class rules and generic traversal for program transformation languages. Master’s thesis, Utrecht University, Utrecht, The Netherlands, INF/SCR-2001-15 (August 2001)
12. Dolstra, E., Visser, E.: First-class rules and generic traversal. Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2001)
13. Hall, C., Hammond, K., Jones, S.P., Wadler, P.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems 18, 241–256 (1996)

14. Borovansky, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science* 1, 69–95 (2001)
15. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: 2nd USENIX Conference on Domain Specific Languages (DSL 1999), Austin, Texas, October 1999, pp. 109–122 (1999)
16. Lämmel, R., Visser, J.: Type-safe functional strategies. In: Scottish Functional Programming Workshop (2000)
17. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: ACM (ed.) Conference record of POPL 1995, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, pp. 333–343. ACM Press, New York (1995)
18. Hughes, J.: Generalising monads to arrows. *Science of Computer Programming* 37, 67–111 (2000)
19. Appel, A.W.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press, Cambridge (2002)
20. Jones, M.P., Diatchki, I.S.: Language and program design for functional dependencies. In: Haskell 2008: Proceedings of the first ACM SIGPLAN symposium on Haskell, pp. 87–98. ACM, New York (2008)
21. Gill, A., Hutton, G.: The worker/wrapper transformation. *Journal of Functional Programming* 19(2), 227–251 (2009)
22. Visser, E., Benaissa, Z., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), pp. 13–26. ACM Press, New York (1998)
23. Visser, E.: A survey of rewriting strategies in program transformation systems. In: Gramlich, B., Lucas, S. (eds.) Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001), Utrecht, The Netherlands. *Electronic Notes in Theoretical Computer Science*, vol. 57. Elsevier Science Publishers, Amsterdam (2001)
24. Weaver, P., Kimmell, G., Frisby, N., Alexander, P.: Constructing language processors with algebra combinators. In: GPCE 2007: Proceedings of the 6th international conference on Generative programming and component engineering, pp. 155–164. ACM, New York (2007)
25. Harrison, W.L., Kamin, S.N.: Metacomputation-based compiler architecture. In: *Mathematics of Program Construction*, pp. 213–229 (2000)
26. Balland, E., Moreau, P.E., Reilles, A.: Rewriting strategies in Java. *Electron. Notes Theor. Comput. Sci.* 219, 97–111 (2008)
27. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Haskell 2007: Proceedings of the ACM SIGPLAN workshop on Haskell, pp. 73–82. ACM, New York (2007)

# Varying Domain Representations in Hagl

## Extending the Expressiveness of a DSL for Experimental Game Theory

Eric Walkingshaw and Martin Erwig

School of Electrical Engineering and Computer Science,  
Oregon State University, Corvallis, OR 97331, USA

**Abstract.** Experimental game theory is an increasingly important research tool in many fields, providing insight into strategic behavior through simulation and experimentation on game theoretic models. Unfortunately, despite relying heavily on automation, this approach has not been well supported by tools. Here we present our continuing work on Hagl, a domain-specific language embedded in Haskell, intended to drastically reduce the development time of such experiments and support a highly explorative research style.

In this paper we present a fundamental redesign of the underlying game representation in Hagl. These changes allow us to better utilize domain knowledge by allowing different classes of games to be represented differently, exploiting existing domain representations and algorithms. In particular, we show how this supports analytical extensions to Hagl, and makes strategies for state-based games vastly simpler and more efficient.

## 1 Introduction

Game theory has traditionally been used as an analytical framework. A game is a formal model of a strategic situation in which players interact by making moves, eventually achieving a payoff in which each player is awarded a value based on the outcome of the game. Classical game theorists have derived many ways of solving such situations, by mathematically computing “optimal” strategies of play, usually centered around notions of stability or equilibrium [1].

The derivation of these optimal strategies, however, almost always assumes perfectly rational play by all players—an assumption which rarely holds in practice. As such, while these methods have many practical uses, they often fair poorly at predicting *actual* strategic behavior by humans and other organisms.

One striking example of sub-optimal strategic behavior is the performance of humans in a class of simple guessing games [2]. In one such game, a group of players must each guess a number between 0 and 100 (inclusive). The goal is to guess the number closest to  $1/2$  of the average of all players’ guesses. Traditional game theory tells us that there is a single equilibrium strategy for this game, which is to choose zero. Informally, the reasoning is that since each player is rational and assumes all other players are rational, any value considered above zero would lead the player to subsequently consider  $1/2$  of that value. That is, if

a player initially thinks the average of the group might be 50 (based on randomly distributed guesses), 25 would initially seem a rational guess. However, since all other players are assumed to be rational, the player would assume that the others came to the same conclusion and would also choose 25, thereby making 12.5 a better guess. But again, all players would come to the same conclusion, and after halving their guesses once again it quickly becomes clear that an assumption of rationality among all players leads each to recursively consider smaller and smaller values, eventually converging to zero, which all rational players would then play.

It has been demonstrated experimentally, however, that most human players choose values greater than zero [3]. Interestingly, a nonzero guess does not necessarily imply irrationality on the part of the player, who may be rational but assumes that others are not. Regardless, it is clear that the equilibrium strategy is sub-optimal when playing against real opponents.

Experimental game theory attempts to capture, understand, and predict strategic behavior where analytical game theory fails or is difficult to apply. The use of game theoretic models in experiments and simulations has become an important research tool for economists, biologists, political scientists, and many other researchers. This shift towards empirical methods is also supported by the fact that game theory's formalisms are particularly amenable to direct execution and automation [4]. Interactions, decisions, rewards, and the knowledge of players are all formally defined, allowing researchers to conduct concise and quantifiable experiments.

Perhaps surprisingly, despite the seemingly straightforward translation from formalism to computer automation, general-purpose tool support for experimental game theory is extremely low. A 2004 attempt to recreate and extend the famous Axelrod experiments on the evolution of cooperation [5] exemplifies the problem. Experimenters wrote a custom Java library (IPDLX) to conduct the experiments [6]. Like the original, the experiments were structured as tournaments between strategies submitted by players from around the world, competing in the iterated form of a game known as the *prisoner's dilemma*. Excluding user interface and example packages, the library used to run the tournament is thousands of lines of Java code. As domain-specific language designers, we knew we could do better.

Enter Hagl<sup>1</sup>, a domain-specific language embedded in Haskell designed to make defining games, strategies, and experiments as simple and fun as possible. In Hagl, the Axelrod experiments can be reproduced in just a few lines.

```
data Cooperate = C | D
dilemma = symmetric [C, D] [2, 0, 3, 1]
axelrod players = roundRobin dilemma players (times 100)
```

In our previous work on Hagl [7] we have provided a suite of operators and smart constructors for defining common types of games, a combinator library

---

<sup>1</sup> Short for “Haskell game language”.

Available for download at: <http://web.engr.oregonstate.edu/~walkiner/hagl/>

for defining strategies for playing games, and a set of functions for carrying out experiments using these objects. Continuing the example above, we provide a few example players defined in Hagl below, then run the Axelrod experiment on this small sample of players.

At each iteration of the iterated prisoner’s dilemma, each player can choose to either cooperate or defect. The first two very simple players just play the same move every time.

```
mum = "Mum" 'plays' pure C
fink = "Fink" 'plays' pure D
```

A somewhat more interesting player is one that plays the famous “Tit for Tat” strategy, winner of the original Axelrod tournament. Tit for Tat cooperates in the first iteration, then always plays the previous move played by its opponent.

```
tft = "Tit for Tat" 'plays' (C 'initiallyThen' his (prev move))
```

With these three players defined, we can carry out the experiment. The following, run from an interactive prompt (e.g. GHCi), plays each combination of players against each other 100 times, printing out the final scores.

```
> axelrod [mum,fink,tft]
Final Scores:
  Tit for Tat: 699.0
  Fink: 602.0
  Mum: 600.0
```

Since this original work we have set out to add various features to the language. In this paper we primarily discuss two such additions, the incorporation of operations from analytical game theory, and adding support for state-based games, which together revealed substantial *bias* in Hagl’s game representation.

## 1.1 Limitations of the Game Representation

The definition of the prisoner’s dilemma above is nice since it uses the terminology and structure of notations in game theory. Building on this, we would like to support other operations familiar to game theorists. In particular, we would like to provide analytical functions that return equilibrium solutions like those mentioned above. This is important since experimental game theorists will often want to use classically derived strategies as baselines in experiments. Unfortunately, algorithms that find equilibria of games like the prisoner’s dilemma rely on the game’s simple structure. This structure, captured in the concrete syntax of Hagl’s notation above, is immediately lost since all games are converted into a more general internal form, making it very difficult to provide these operations to users. This is an indication of bias in Hagl—the structure inherent in some game representations is lost by converting to another. Instead, we would like all game representations to be first-class citizens, allowing games of different types to be structured differently.

The second limitation is related to *state-based games*. A state-based game is a game that is most naturally described as a series of transformations over some state. As we've seen, defining strategies for games like the prisoner's dilemma is very easy in Hagl. Likewise for the extensive form games introduced in Section 2.2. Unfortunately, the problem of representational bias affects state-based games as well, making the definition of strategies for these kinds of games extremely difficult.

In Section 2.3 we present a means of supporting state-based games within the constraints of Hagl's original game representation. The match game is a simple example of a state-based game that can be defined in this way. In the match game  $n$  matches are placed on a table. Each player takes turns drawing some limited number of matches until the player who takes the last match loses. The function below generates a two-player match game with  $n$  matches in which each player may choose some number from  $ms$  matches each turn. For example, `matches 15 [1,2,3]` would generate a match game in which 15 matches are placed on the table, and each move consists of taking away 1, 2, or 3 matches.

```
matches :: Int -> [Int] -> Game Int
matches n ms = takeTurns 2 end moves exec pay n
  where end n _ = n <= 0
        moves n _ = [m | m <- ms, n-m >= 0]
        exec n _ m = n-m
        pay _ 1 = [1,-1]
        pay _ 2 = [-1,1]
```

The `takeTurns` function is described in Section 2.3. Here it is sufficient to understand that the state of the game is the number of matches left on the table and that the game is defined by the series of functions passed to `takeTurns`, which describe how that state is manipulated.

Defining the match game in this way works fairly well. The problem is encountered when we try to write strategies to play this game. In Section 3.2, we show how state-based games are transformed into Hagl's internal, stateless game representation. At the time a strategy is run, the game knows nothing about the state. This means that a strategy for the match game has no way of seeing how many matches remain on the table! In Section 5.2 we provide an alternative implementation of the match game using the improved model developed in Section 4. Using this model, which utilizes type classes to support games with diverse representations, we can write strategies for the match game, an example of which is also given in Section 5.2.

## 1.2 Outline of Paper

In the following section we provide an interleaved introduction to Hagl and game theory. This summarizes our previous work and also includes more recent additions such as preliminary support for state-based games, improved dimensioned list indexing operations, and support for symmetric games. Additionally, it provides a foundation of concepts and examples which is drawn upon in the rest of

the paper. In Section 3 we discuss the bias present in our original game representation and the limitations it imposes. In Section 4 we show how we can overcome this bias, generalizing our model to support many distinct domain representations. In Section 5 we utilize the new model to present solutions to the problems posed in Section 3.

## 2 A Language for Experimental Game Theory

In designing Hagl, we identified four primary domain objects that must be represented: games, strategies, players, and simulations/experiments. Since this paper is primarily concerned with game representations, this section is correspondingly heavily skewed towards that subset of Hagl. Sections 2.1, 2.2, and 2.3 all focus on the definition of different types of games in Hagl, while Section 2.4 provides an extremely brief introduction to other relevant aspects of the language. For a more thorough presentation of the rest of the language, please see our earlier work in [7].

Game theorists utilize many different representations for different types of games. Hagl attempts to tap into this domain knowledge by providing smart constructors and operators which mimic existing domain representations as closely as possible given the constraints of Haskell syntax. The next two subsections focus on these functions, while Section 2.3 introduces preliminary support for state-based games. Each of these subsections focuses almost entirely on concrete syntax, the interface provided to users of the system. The resulting type and internal representation of these games, `Game mv`, is left undefined throughout this section but will be defined and discussed in depth in Section 3.

### 2.1 Normal Form Game Definition

One common representation used in game theory is *normal form*. Games in normal form are represented as a matrix of payoffs indexed by each player's move. Fig. 1 shows two related games which are typically represented in normal form. The first value in each cell is the payoff for the player indexing the rows of the matrix, while the second value is the payoff for the player indexing the columns.

	C	D
C	2, 2	0, 3
D	3, 0	1, 1

	C	D
C	3, 3	0, 2
D	2, 0	1, 1

(a) Prisoner's dilemma

(b) Stag hunt

**Fig. 1.** Normal form representation of two related games

The first game in Fig. 1 is the prisoner’s dilemma, which was briefly introduced in Section 1; the other is called the *stag hunt*. These games will be referred to throughout the paper. In both games, each of two players can choose to either cooperate (**C**) or defect (**D**). While these games are interesting from a theoretical perspective and can be used to represent many real-world strategic situations, they each (as with many games in game theory) have a canonical story associated with them from which they derive their names.

In the prisoner’s dilemma the two players represent prisoners suspected of collaborating on a crime. The players are interrogated separately, and each can choose to cooperate with his criminal partner by sticking to their fabricated story (not to be confused with cooperating with the interrogator, who is not a player in the game), or they can choose to defect by telling the interrogator everything. If both players cooperate they will be convicted of only minor crimes—a pretty good outcome considering their predicament, and worth two points to each player as indicated by the payoff matrix. If both players defect, they will be convicted of more significant crimes, represented by scoring only one point each. Finally, if one player defects and the other cooperates, the defecting player will be pardoned, while the player who sticks to the fabricated story will be convicted of the more significant crime in addition to being penalized for lying to the investigators. This outcome is represented by a payoff of 3,0, getting pardoned having the best payoff, and being betrayed leading to the worst.

In the stag hunt, the players represent hunter-gathers. Each player can choose to either cooperate in the stag hunt, or defect by spending their time gathering food instead. Mutual cooperation leads to the best payoff for each player, as the players successfully hunt a stag and split the food. This is represented in the payoff matrix by awarding three points to each player. Alternatively, if the first player cooperates but the second defects, a payoff of 0,2 is awarded, indicating zero points for the first player, who was unsuccessful in the hunt, and two points for the second player, who gathered some food, but not as much as could have been acquired through a hunt. Finally, if both choose to gather food, they will have to split the food that is available to gather, earning a single point each.

Hagl provides the following smart constructor for defining normal form games.

```
normal :: Int -> [[mv]] -> [[Float]] -> Game mv
```

This function takes the number of players the game supports, a list of possible moves for each player, a matrix of payoffs, and returns a game. For example, the stag hunt could be defined as follows.

```
stag = normal 2 [[C,D],[C,D]] [[3,3],[0,2],
[2,0],[1,1]]
```

The syntax of this definition is intended to resemble the normal form notation in Fig. 1. By leveraging game theorists’ existing familiarity with domain representations, we hope to make Hagl accessible to those within the domain.

Since two-player games are especially common amongst normal form games, game theorists have special terms for describing different classes of two-player,

normal form games. The most general of which is a *bimatrix* game, which is simply an arbitrary two player, normal form game.

```
bimatrix = normal 2
```

Somewhat more specialized are *matrix* games and *symmetric* games. A matrix game is a two-player, zero-sum game. In these games, whenever one player wins, the other player loses a corresponding amount. Therefore, the payoff matrix for the *matrix* function is a simple list of floats, the payoffs for the first player, from which the payoffs for the second player can be derived.

```
matrix :: [mv] -> [mv] -> [Float] -> Game mv
```

The traditional game rock-paper-scissors, defined below, is an example of such a game. When one player wins (scoring +1) the other loses (scoring -1), or the two players can tie (each scoring 0) by playing the same move.

```
data RPS = R | P | S
rps = matrix [R,P,S] [R,P,S] [0,-1, 1,
                                1, 0,-1,
                                -1, 1, 0]
```

Symmetric games are similar in that the payoffs for the second player can be automatically derived from the payoffs for the first player. In a symmetric game, the game appears identical from the perspective of both players; each player has the same available moves and the payoff awarded to a player depends only on the combination of their move and their opponent's, not on whether they are playing along the rows or columns of the grid.

```
symmetric :: [mv] -> [Float] -> Game mv
```

Note that all of the games discussed so far have been symmetric. Below is an alternative, more concise definition of the stag hunt, using this new smart constructor.

```
stag = symmetric [C,D] [3, 0, 2, 1]
```

Recall that the definition of the prisoner's dilemma given in Section 1 utilized this function as well.

Although it is theoretically possible to represent any game in normal form, it is best suited for games in which each player plays simultaneously from a finite list of moves. For different, more complicated games, game theory relies on other representations. One of the most common and general of which is *extensive form*, also known as decision or game trees.

## 2.2 Extensive Form Game Definition

Extensive form games are represented in Hagl by the following data type, which makes use of the preceding set of type synonyms.

```

type PlayerIx = Int
type Payoff = ByPlayer Float
type Edge mv = (mv, GameTree mv)
type Dist a = [(Int, a)]

data GameTree mv = Decision PlayerIx [Edge mv]
  | Chance (Dist (Edge mv))
  | Payoff Payoff

```

**Payoff** nodes are the leaves of a game tree, containing the score awarded to each player for a particular outcome. The type of a payoff node's value, `ByPlayer Float`, will be explained in Section 2.4, but for now we'll consider it simply a type synonym for a list of `Float` values.

Internal nodes are either `Decision` or `Chance` nodes. `Decision` nodes represent locations in the game tree where a player must choose to make one of several moves. Each move corresponds to an `Edge` in the game tree, a mapping from a move to its resulting subtree. The player making the decision is indicated by a `PlayerIx`. The following example presents the first (and only) player of a game with a simple decision; if he chooses move `A`, he will receive zero points, but if he chooses move `B`, he will receive five points.

```
easyChoice = Decision 1 [(A, Payoff [0]), (B, Payoff [5])]
```

Finally, `Chance` nodes represent points where an external random force pushes the game along some path or another based on a distribution. Currently, that distribution is given by a list of edges prefixed with their relative likelihood; e.g. given `[(1,a),(3,b)]`, the edge `b` is three times as likely to be chosen as `a`. However, this definition of `Dist` could easily be replaced by a more sophisticated representation of probabilistic outcomes, such as that presented in [8]. A random die roll, while not technically a game from a game theoretic perspective, is illustrative and potentially useful as a component in a larger game. It can be represented as a single `Chance` node where each outcome is equally likely and the payoff of each is the number showing on the die.

```
die = Chance [(1, (n, Payoff [n])) | n <- [1..6]]
```

The function `extensive`, which has type `GameTree mv -> Game mv` is used to turn game trees into games which can be run in Hagl. Hagl also provides operators for incrementally building game trees, but those are not presented here.

### 2.3 State-Based Game Definition

One of the primary contributions of this work is the addition of support for state-based games in Hagl. While the normal and extensive forms are general in the sense that any game can be translated into either representation, games which can be most naturally defined as transitions between states seem to form a much broader and more diverse class. From auctions and bargaining games, where the state is the highest bid, to graph games, where the state is the location of players and elements in the graph, to board games like chess and tic-tac-toe,

where the state is the configuration of the board, many games seem to have natural notions of state and well-defined transitions between them.

The preliminary support provided for state-based games described here has been made obsolete by the redesign described in Section 4, but presenting our initial design is valuable as a point of reference for future discussion.

As with other game representations, support for state-based games is provided by a smart constructor, whose type is given below.

```
stateGame :: Int -> (s -> PlayerIx) -> (s -> PlayerIx -> Bool) ->
            (s -> PlayerIx -> [mv]) -> (s -> PlayerIx -> mv -> s) ->
            (s -> PlayerIx -> Payoff) -> s -> Game mv
```

The type of the state is determined by the type parameter `s`. The first argument to `stateGame` indicates the number of players the game supports, and the final argument provides an initial state. The functional arguments in between describe how players interact with the state over the course of the game. Each of these takes the current state and (except for the first) the current player, and returns some information about the game. In order, the functional arguments define:

- which player’s turn it is,
- whether or not the game is over,
- the available moves for a particular player,
- the state resulting from executing a move on the given state, and
- the payoffs for some final state.

The result type of this function is the same `Game mv` type as the other games described in previous sections. Notice that `s`, the type of the state, is not reflected in the type signature of the resulting game. This is significant, reflecting the fact that the notion of state is completely lost in the generated game representation.

Since players alternate in many state-based games, and tracking which player’s turn it is can be cumbersome, another smart constructor, `takeTurns`, is provided which manages this aspect of state games automatically. The `takeTurns` function has the same type as the `stateGame` function above minus the second argument, which is used to determine whose turn it is.

```
takeTurns :: Int -> (s -> PlayerIx -> Bool) -> (s -> PlayerIx -> [mv]) ->
            (s -> PlayerIx -> mv -> s) -> (s -> PlayerIx -> Payoff) ->
            s -> Game mv
```

The match game presented in Section 1.1 is an example of a state-based game defined using this function.

The definition of players, strategies, and experiments are less directly relevant to the contributions of this paper than the definition of games, but the brief introduction provided in the next subsection is necessary for understanding later examples. The definition of strategies for state-based games, in particular, are one of the primary motivations for supporting multiple domain representations.

## 2.4 Game Execution and Strategy Representation

All games in Hagl are inherently *iterated*. In game theory, the iterated form of a game is just the original game played repeatedly, with the payoffs of each

iteration accumulating. When playing iterated games, strategies often rely on the events of previous iterations. Tit for Tat, presented in Section 1.1, is one such example.

Strategies are built from a library of functions and combinators which, when assembled, resemble English sentences written from the perspective of the player playing the strategy (i.e. in Tit for Tat, `his` corresponds to the other player in a two player game). Understanding these combinators requires knowing a bit about how games are executed.

In Hagl, all games are executed within the following monad.

```
data ExecM mv a = ExecM (StateT (Exec mv) IO a)
```

This data type wraps a state transformer monad, and is itself an instance of the standard `Monad`, `MonadState` and `MonadIO` type classes, simply deferring to the monad it wraps in all cases. The inner `StateT` monad transforms the `IO` monad, which is needed for printing output and obtaining random numbers (e.g. for `Chance` nodes in extensive form games).

The state of the `ExecM` monad, a value of type `Exec mv`, contains all of the runtime information needed to play the game, including the game itself and the players of the game, as well as a complete history of all previous iterations. The exact representation of this data type is not given here, but the information is made available in various formats through a set of *accessor* functions, some of which will be shown shortly.

A player in Hagl is represented by the `Player` data type defined below. Values of this type contain the player's name (e.g. "Tit for Tat"), an arbitrary state value, and a strategy which may utilize that state.

```
data Player mv = forall s. Player Name s (Strategy mv s)
```

The type `Name` is simply a synonym for `String`. More interestingly, notice that the `s` type parameter is existentially quantified, allowing players with different state types to be stored and manipulated generically within the `ExecM` monad.

The definition of the `Strategy` type introduces one more monadic layer to Hagl. The `StratM` monad adds an additional layer of state management, allowing players to store and access their own personal state of type `s`.

```
data StratM mv s a = StratM (StateT s (ExecM mv) a)
```

The type `Strategy mv s` found in the definition of the `Player` data type, is simply a type synonym for `StratM mv s mv`, an execution in the `StratM` monad which returns a value of type `mv`, the move to be played.

Since many strategies do not require additional state, the following smart constructor is provided which circumvents this aspect of player definition.

```
plays :: Name -> Strategy mv () -> Player mv
plays n s = Player n () s
```

This function reads particularly well when used as an infix operator, as in the definition of Tit for Tat.

Hagl provides smart constructors for defining simple strategies in game theory, such as the pure and mixed strategies discussed in Section 3.1, but for defining more complicated strategies we return to the so-called accessor functions mentioned above. The accessors extract data from the execution state and transform it into some convenient form. Since they are accessing the state of the `ExecM` monad, we would expect their types to have the form `ExecM mv a`, where `a` is the type of the returned information. Instead, they have types of the form `GameM m mv => m a`, where `GameM` is a type class instantiated by both `ExecM` and `StratM`. This allows the accessors to be called from within strategies without needing to be “lifted” into the context of a `StratM` monad.

A few sample accessors are listed below, with a brief description of the information they return. The return types of each of these rely on two data types, `ByGame a` and `ByPlayer a`. Each is just a wrapper for a list of `as`, but are used to indicate how that list is indexed. A `ByPlayer` list indicates that each element corresponds to a particular player, while a `ByGame` list indicates that each element corresponds to a particular game iteration.

- `move :: GameM m mv => m (ByGame (ByPlayer mv))`  
A doubly nested list of the last move played by each player in each game.
- `payoff :: GameM m mv => m (ByGame Payoff)`  
A doubly nested list of the payoff received by each player in each game.
- `score :: GameM m mv => m (ByPlayer Float)`  
The current cumulative scores, indexed by player.

The benefits of the `ByGame` and `ByPlayer` types become apparent when we start thinking about how to process these lists. First, they help us keep indexing straight. The two different types of lists are organized differently, and the two indexing functions, `forGame` and `forPlayer`, perform all of the appropriate conversions and abstract the complexities away. Second, the data types provide additional type safety by ensuring that we never index into a `ByPlayer` list thinking that it is `ByGame`, or vice versa. This is especially valuable when we consider our other class of combinators, called *selectors*, which are used to process the information returned by the accessors.

While the accessors above provide data, the selector functions constrain data. Two features distinguish Hagl selectors from generic list operators. First, they provide the increased type safety already mentioned. Second, they utilize information from the current execution state to make different selections depending on the context in which they are run. Each `ByPlayer` selector corresponds to a first-person, possessive pronoun in an effort to maximize readability. An example is the `his` selector used in the definition of Tit for Tat above.

```
his :: GameM m mv => m (ByPlayer a) -> m a
```

This function takes a monadic computation that returns a `ByPlayer` list and produces a computation which returns only the element corresponding to the opposing player (in a two player game). Other selectors include `her`, a synonym for `his`, `my`, which returns the element corresponding to the current player, and `our`, which selects the elements corresponds to all players.

Similar selectors exist for processing `ByGame` lists, except these have names like `prev`, for selecting the element corresponding to the previous iteration, and `every`, for selecting the elements corresponding to every iteration.

While this method of defining strategies with a library of accessor and selector functions and other combinators has proven to be very general and extensible, we have run into problems with our much more rigid game representation.

### 3 A Biased Domain Representation

In Sections 2.1, 2.2, and 2.3 we introduced a suite of functions and operators for defining games in Hagl. However, we left the subsequent type of games, `Game mv`, cryptically undefined. It turns out that this data type is nothing more than an extensive form `GameTree` value, plus a little extra information.

```
data Game mv = Game { numPlayers :: Int,
                      info          :: GameTree mv -> InfoGroup mv,
                      tree          :: GameTree mv }
```

Thus, all games in Hagl are translated into extensive form. This is nice since it provides a relatively simple and general internal representation for Hagl to work with. As we'll see, however, it also introduces a substantial amount of *representational bias*, limiting what we can do with games later.

In addition to the game tree, a Hagl `Game` value contains an `Int` indicating the number of players that play the game, and a function from nodes in the game tree to *information groups*. In game theory, information groups are an extension to the simple model of extensive form games. The need for this extension can be seen by considering the translation of a simple normal form game like the stag hunt into extensive form. A straightforward translation would result in something like the following.

```
Decision 1 [(C, Decision 2 [(C, Payoff (ByPlayer [3,3])),
                           (D, Payoff (ByPlayer [0,2]))]),
            [(D, Decision 2 [(C, Payoff (ByPlayer [2,0])),
                           (D, Payoff (ByPlayer [1,1]))])]]
```

Player 1 is presented with a decision at the root of the tree; each move leads to a decision by player 2, and player 2's move leads to the corresponding payoff, determined by the combination of both players' moves.

The problem is that we've taken two implicitly simultaneous decisions and sequentialized them in the game tree. If player 2 examines her options, the reachable payoffs will reveal player 1's move. Information groups provide a solution by associating with each other a set of decision nodes for the same player, from within which a player knows only the group she is in, not the specific node.

An information group of size one implies *perfect information*, while an information group of size greater than one implies *imperfect information*. In Hagl, information groups are represented straightforwardly.

```
data InfoGroup mv = Perfect (GameTree mv)
                  | Imperfect [GameTree mv]
```

Therefore, the definition of a new game type in Hagl must not only be translated into a Hagl `GameTree` but must also provide a function for returning the information group corresponding to each `Decision` node. As we can see from the definition of the smart constructor for normal form games introduced earlier, this process is non-trivial.

```
normal :: Int -> [[mv]] -> [[Float]] -> Game mv
normal np mss vs = Game np group (head (level 1))
  where level n | n > np = [Payoff (ByPlayer v) | v <- vs]
    | otherwise = let ms = mss !! (n-1)
                  bs = chunk (length ms) (level (n+1))
                  in map (Decision n . zip ms) bs
group (Decision n _) = Imperfect (level n)
group t = Perfect t
```

Herein lies the first drawback of this approach—defining new kinds of games is difficult due to the often complicated translation process. Until this point in Hagl’s history, we have considered the definition of new kinds of games to be part of our role as language designers. This view is limiting for Hagl’s use, however, given the incredible diversity of games in game theory, and that the design of new games is often important to a game theorist’s research.

A more fundamental drawback of converting everything into a decision tree, however, is that it introduces representational bias. By converting from one representation to another we at best obscure, and at worst completely lose important information inherent in the original representation. The next two subsections demonstrate both ends of this spectrum.

### 3.1 Solutions of Normal Form Games

As mentioned in the introduction, classical game theory is often concerned with computing optimal strategies for playing games. There are many different definitions of optimality, but two particularly important definitions involve computing *Nash equilibria* and finding *Pareto optimal solutions* [1].

A Nash equilibrium is defined as a set of strategies for each player, where each player, knowing the others’ strategies, would have nothing to gain by unilaterally changing his or her own strategy. More colloquially, Nash equilibria describe stable combinations of strategies—even if a player knows the strategies of the other players, he or she would still not be willing to change.

Nash equilibria for normal form games can be *pure* or *mixed*. A pure equilibrium is one where each player plays a specific move. In a mixed equilibrium, players may play moves based on some probability. A good example of a game with an intuitive mixed equilibrium is rock-paper-scissors. If both players randomly play each move with equal probability, no player stands to gain by changing strategies. Going back to our examples from Section 2.1, the stag hunt has two pure Nash equilibria,  $(\mathbf{C}, \mathbf{C})$  and  $(\mathbf{D}, \mathbf{D})$ . If both players are cooperating, they are each earning 3 points; a player switching to defection would earn only 2 points. Similarly, if both players are defecting, they are each earning 1 point;

switching to cooperation would cause the switching player to earn 0 points. The prisoner's dilemma has only one Nash equilibrium, however, which is  $(\mathbf{D}, \mathbf{D})$ . In this case, mutual cooperation is not stable because a player unilaterally switching to defection will earn 3 points instead of 2.

While the focus of Nash equilibria are on ensuring stability, Pareto optimality is concerned with maximizing the benefit to as many players as possible. A *Pareto improvement* is a change from one solution (i.e. set of strategies) to another that causes at least one player's payoff to increase, while causing no players' payoffs to decrease. A solution from which no Pareto improvement is possible is said to be Pareto optimal. The only Pareto optimal solution of the stag hunt is mutual cooperation, since any other solution can be improved by switching to pure cooperation. In the prisoner's dilemma  $(\mathbf{C}, \mathbf{D})$  and  $(\mathbf{D}, \mathbf{C})$  are also both Pareto optimal, since any change would cause the defecting player's payoff to decrease.

Nash equilibria and Pareto optimal solutions are guaranteed to exist for every game, but they are hard to compute, in general. Finding a Nash equilibrium for an arbitrary game is known to be PPAD-complete (computationally intractable) [9], and even the much more constrained problem of deciding whether a game has a pure Nash equilibrium is NP-hard [10]. There do exist, however, simpler algorithms for certain variations of these problems on highly constrained games. Finding pure Nash equilibria and Pareto optimal solutions on the kinds of games typically represented in normal form is comparatively easy [11].

While Hagl's primary focus is experimental game theory, we would like to begin providing support for these kinds of fundamental, analytical operations. Unfortunately, the bias in Hagl's game representation makes this very difficult. We would like to add functions for finding pure Nash equilibria and Pareto optimal solutions, and have them only apply to simple normal form games, but this is impossible since all games have the same type. Additionally, although the structure of a game's payoff matrix is available at definition time, it is instantly lost in the translation to a decision tree.

While this subsection demonstrated how representational bias can obscure the original structure of a game, the next provides an example where the structure is completely and irrecoverably lost.

### 3.2 Loss of State in State-Based Games

Section 2.3 introduced preliminary support for games defined as transformations of some state. The smart constructor `stateGame` takes an initial state and a series of functions describing how to manipulate that state, and produces a standard Hagl game tree representation. Below is the implementation of this function.

```
stateGame np who end moves exec pay init = Game np Perfect (tree init)
  where tree s | end s p  = Payoff (pay s p)
        | otherwise = Decision p [(m, tree (exec s p m))
                                    | m <- moves s p]
  where p = who s
```

The most interesting piece here is the `tree` function, which threads the state through the provided functions to generate a standard, stateless decision tree.

Haskell's laziness makes this viable for even the very large decision trees often generated by state-based games. But there is still a fundamental problem with this solution: we've lost the state.

At first this way of folding the state away into a game tree was very appealing. The solution is reasonably elegant, and seemed to emphasize that even the most complex games could be represented in our general form. The problem is first obvious when one goes to write a strategy for a state-based game. Because the state is folded away, it is forever lost immediately upon game definition; strategies do not and cannot have access to the state during game execution. Imagine trying to write a strategy for a chess player without being able to see the board!

Of course, there is a workaround. Since players are afforded their own personal state, as described in Section 2.4, each player of a state-based game could personally maintain his or her own copy of the game state, modifying it as other players make their moves. In addition to being wildly inefficient, this makes defining strategies for state-based games much more difficult, especially considering strategies may have their own additional states to maintain as well.

This limitation of strategies for state-based games is perhaps the most glaring example of representational bias. The translation from a state-based representation to a stateless extensive form representation renders the game nearly unusable in practice.

## 4 Generalizing Game Representations

In setting about revising Hagl's game representation, we established four primary goals, listed here roughly in order of significance, from highest to lowest.

1. Better accommodate state in games. Many kinds of games are very naturally represented as transitions between states. The inability of strategies to directly access these states, as described in Section 3.2, was the most egregious instance of representational bias in the language.
2. Lessen representational bias in general by allowing for multiple different game representations. Although dealing with the state issue was a pressing need, we sought a more general solution to the underlying problem. This would also help us overcome the problems described in Section 3.1.
3. Lower the barrier to entry for defining new classes of games. Having one general game representation is nice, but requiring a translation from one representation to another is difficult, likely preventing users of the language from defining their own game types.
4. Minimize impact on the rest of the language. In particular, high-level game and strategy definitions should continue to work, with little to no change.

In the rest of this section we will present a design which attempts to realize these goals, touching briefly on the motivations behind important design decisions.

From the goals stated above, and the second goal in particular, it seems clear that we need to introduce a type class for representing games. Recall the game

representation given in Section 3: A game is represented by the `Game` data type which contains the number of players that play the game, an explicit game tree as a value of the `GameTree` data type, and a function for getting the information group associated with a particular node. One approach would be to simply redefine the `Game` data type as a type class as shown below.

```
class Game g where
  type Move g
  numPlayers :: g -> Int
  gameTree   :: g -> GameTree (Move g)
  info       :: g -> GameTree (Move g) -> Info (Move g)
```

Note that this definition uses associated types [12], an extension to Haskell 98 [13] available in GHC [14], which allow us to use type classes to overload types in the same way that we overload functions. `Move g` indicates the move type associated with the game type `g` and is specified in class instances, as we'll see later.

In effect, the use of a type class delays the translation process, maintaining diverse game representations and converting them into game trees immediately before execution. However, this solution still does not support state-based games, nor does it make defining new types of games any easier since we must still provide a translation to a game tree. Enabling state-based games would be fairly straightforward; we could simply add a state value to each node in the game tree, making it trivial to retain the state that was previously folded away, and to provide easy access to it by strategies. Making game definition easier, on the other hand, requires a more radical departure from our original design.

## 4.1 Final Game Representation

Ultimately, we decided to abandon the game tree representation altogether and define games as arbitrary computations within the game execution monad. The final definition of the `Game` type class is given below.

```
class Game g where
  type Move g
  type State g
  initState :: g -> State g
  runGame   :: ExecM g Payoff
```

Note that we provide explicit support for state-based games by adding a new associated type `State` and a function `initState` for getting the initial state of the game. This state is then stored with the rest of the game execution state and can be accessed and modified from within the monadic `runGame` function, which describes the execution of the game.

Also note that we have removed the `numPlayers` method from the previous type class. This change eliminates another small source of bias, reflecting the fact that many games support a variable number of players, violating the functional relationship between a game and a constant number of players that was

previously implied. Game instances should now handle this aspect of game definition on their own, as needed.

To ease the definition of new types of games, we also provide a library of game definition combinators. These combinators provide a high-level interface for describing the execution of games, while hiding all of the considerable minutiae of game execution briefly discussed in Section 2.4. Some of these combinators will be introduced in the next subsection, but first we discuss some of the trade-offs involved in this design change.

Perhaps the biggest risk in moving from an *explicit* representation (game trees) to an *implicit* representation (monadic computations) is the possibility of overgeneralization. While abstraction helps in removing bias from a representation, one has to be careful not to abstract so far away from the domain that it is no longer relevant. After all, we initially chose game trees since they were a very general representation of *games*. Does a monadic computation really capture what it means to be a game?

Another subtle drawback is that, with the loss of an explicit game representation, it is no longer possible to write some generic functions which relied on this explicitness. For example, we can no longer provide a function which returns the available moves for an arbitrary game. Such functions would have to be provided per game type, as needed. For games where an explicit representation is more suitable, another type class is provided, similar to the initial type class suggested in Section 4 that defines a game in terms of a game tree, as before.

These risks and minor inconveniences are outweighed by many substantial benefits. First and foremost, the monadic representation is much more flexible and extensible than the tree-based approach. Because games were previously defined in terms of a rigid data type, they were limited to only three types of basic actions, corresponding to the three constructors in the `GameTree` data type. Adding a fundamentally new game construct required modifying this data type, a substantial undertaking involving the modification of lots of existing code. Adding a new construct in the new design, however, is as easy as adding a new function.

Another benefit of the new design is that games can now vary depending on their execution context. One could imagine a game which changes during execution to handicap the player with highest score, or a game where the payoff of a certain outcome depends on past events, perhaps to simulate diminishing returns. These types of games were not possible before. Since games now define their execution directly and have complete access to the execution context through the `ExecM` monad, we can define games which change as they play.

Finally, as the examples given in Section 5 demonstrate, defining game execution in terms of the provided combinators is substantially easier than translating game representations into extensive form. This lowers the barrier to entry for game definition in Hagl, making it much more reasonable to expect users of the system to be able to define their own game types as needed. In the next subsection we provide an introduction to some of the combinators in the game definition library.

## 4.2 Game Definition Combinators

Game execution involves a lot of bookkeeping. Various data structures are maintained to keep track of past moves and payoffs, the game's current state, which players are involved, etc. The combinators introduced in this section abstract away all of these details, providing high-level building blocks for describing how games are executed.

The first combinator is one of the most basic and will be used in most game definitions. This function executes a decision by the indicated player.

```
decide :: Game g => PlayerIx -> ExecM g (Move g)
```

This function retrieves the indicated player, executes his or her strategy, updates the historical information in the execution state accordingly, and returns the move that was played.

The `allPlayers` combinator is used to carry out some action for all players simultaneously, returning a list of the accumulated results.

```
allPlayers :: Game g => (PlayerIx -> ExecM g a) -> ExecM g (ByPlayer a)
```

Combining these first two combinators as `allPlayers decide`, we define a computation in which all players make a simultaneous (from the perspective of the players) decision. This is used, for example, in the definition of normal form games in Section 5.1.

While the `allPlayers` combinator is used for carrying out simultaneous actions, the `takeTurns` combinator is used to perform sequential actions. It cycles through the list of players, executing the provided computation for each player, until the terminating condition (second argument) is reached.

```
takeTurns :: Game g => (PlayerIx -> ExecM g a) -> ExecM g Bool -> ExecM g a
```

This combinator is used in the revised definition of the match game given in Section 5.2.

There are also a small number of functions for constructing common payoff values. One example is the function `winner`, whose type is given below.

```
winner :: Int -> PlayerIx -> Payoff
```

When applied as `winner n p`, this function constructs a payoff value (recall that `Payoff` is a synonym for `ByPlayer Float`) where the winning player `p` receives 1 point, and all other players, out of `n`, receive `-1` point. There is a similar function `loser`, which gives a single player `-1` point and all other players 1 point, and a function `tie` which takes a single `Int` and awards all players 0 points.

## 5 Flexible Representation of Games

In this section we demonstrate the improved flexibility of the game representation by solving some of the problems posed earlier in the paper. Section 5.1 demonstrates the new representation of normal form games and provides functions for computing analytical solutions. Section 5.2 provides a new implementation of the match game and a strategy that can access its state in the course of play.

### 5.1 Representing and Solving Normal Form Games

The fundamental shortcoming of the original game representation with regard to normal form games was that the structure of the games was lost. Now, we can capture this structure explicitly in the following data type.

```
data Normal mv = Normal Int (ByPlayer [mv]) [Payoff]
```

Note that the arguments to this data constructor are almost exactly identical to the normal form smart constructor introduced in Section 2.1. A second data type resembles another smart constructor from the same section, for constructing two-player, zero-sum games.

```
data Matrix mv = Matrix [mv] [mv] [Float]
```

Even though these games are closely related, we represent them with separate data types because some algorithms, such as the one for computing saddle point equilibria described below, apply only to the more constrained matrix games. To tie these two data types together, we introduce a type class that represents all normal form games, which both data types implement.

```
class Game g => Norm g where
  numPlayers :: g -> Int
  payoffFor :: g -> Profile (Move g) -> Payoff
  moves :: g -> PlayerIx -> [Move g]
```

This type class allows us to write most of our functions for normal form games in a way that applies to both data types. The `Profile` type here refers to a *strategy profile*, a list of moves corresponding to each player. This is reflected in the definition of `Profile mv`, which is just a type synonym for `ByPlayer mv`. The `payoffFor` method returns the payoff associated with a particular strategy profile by looking it up in the payoff matrix. The `moves` method returns the moves available to a particular player.

Using the type class defined above, and the combinators from Section 4.2, we can define the execution of normal form games as follows.

```
runNormal :: Norm g => ExecM g Payoff
runNormal = do g <- game
               ms <- allPlayers decide
               return (g `payoffFor` ms)
```

Contrast this definition with the smart constructor in Section 3 that translated normal form to extensive form. We think that this dichotomy is extremely compelling motivation for the generalized representation. Defining new types of games is now a much simpler process.

Finally, in order to use normal form games within Hagl, we must instantiate the `Game` type class for both data types. This is very straightforward using the `runNormal` function above, and we show only one instance here since the other is nearly identical.

```
instance Eq mv => Game (Normal mv) where
  type Move (Normal mv) = mv
  type State (Normal mv) = ()
  initState _ = ()
  runGame = runNormal
```

Since normal form games do not require state, the associated state type of is just the unit type `()`. Similarly, for any normal form game, the initial state value is the unit value.

Now we can run experiments on, and write strategies for, normal form games just as before. And with the addition of some very straightforward smart constructors, we can directly reuse our earlier definitions of the prisoner’s dilemma and stag hunt. We can also, however, write the analytical functions which we previously could not. Below are type definitions for functions which return the pure Nash equilibria and Pareto optimal solutions of normal form games.

```
nash    :: (Norm g, Eq (Move g)) => g -> [Profile (Move g)]
pareto :: (Norm g, Eq (Move g)) => g -> [Profile (Move g)]
```

Using these we can define a function to find Pareto-Nash equilibria, solutions which are both Pareto optimal and a Nash equilibrium, and which represent especially desirable strategies to play [15].

```
paretoNash g = pareto g `intersect` nash g
```

Applying this new analytical function to the stag and hunt and prisoner’s dilemma provides some interesting results.

```
> paretoNash stag
[ByPlayer [C,C]]
> paretoNash pd
[]
```

This likely demonstrates why prisoner’s dilemma is a far more widely studied game than the stag hunt. The stag hunt is essentially solved—a strategy which is both Pareto optimal and a Nash equilibria is a truly dominant strategy. Compare this to the following analysis of the prisoner’s dilemma.

```
> nash pd
[ByPlayer [D,D]]
> pareto pd
[ByPlayer [C,C], ByPlayer [C,D], ByPlayer [D,C]]
```

In the prisoner’s dilemma, the only Nash equilibrium is not Pareto optimal, while all other solutions are. This makes for a much more subtle and complex game.

Saddle points represent yet another type of equilibrium solution, but one that only applies to matrix games. Essentially, a saddle point of a matrix game is strategy profile which corresponds to a value which is both the smallest value in its row and the largest value in its column [11]. Such a value is ideal for both players, and thus we would expect two rational players to always play a strategy corresponding to a saddle point, if one exists. Hagl provides the following function for finding saddle points in matrix games.

```
saddle :: Eq mv => Matrix mv -> [Profile mv]
```

A key feature of this function is that the type system prevents us from applying it to a normal form game which is not of the right form. Being able to better utilize the type system is another advantage of the more flexible representation enabled by type classes.

## 5.2 Representing and Playing the Match Game

In Section 1.1 we presented a definition of the match game only to discover that we could not write a strategy for it. In this subsection we present a redefinition of the game with the improved representation, and an unbeatable strategy for playing it.

First, we create a data type to represent an instance of the match game. As with our normal form game definition in Section 5.1, this data type resembles the original smart constructor for building match games.

```
data Matches = Matches Int [Int]
```

Here, the first argument indicates the number of matches to set on the table, while the second argument defines the moves available to the players (i.e. the number of matches that can be drawn on a turn).

Instantiating the `Game` type class for the match game is mostly straightforward, with only the implementation of `runGame` being non-trivial.

```
instance Game Matches where
  type Move Matches = Int
  type State Matches = Int
  initState (Matches n _) = n
  runGame = ...
```

The associated `Move` type of the match game is `Int`, the number of matches to draw; the `State` type is also `Int`, the number of matches remaining on the table; and the initial state of the match game is just extracted from the data type. To define the execution of the match game, we build up a series of helper functions.

First we define two simple functions which make manipulating the state of the match game a little nicer.

```
matches = gameState
draw n = updateGameState (subtract n)
```

Both the `gameState` and `updateGameState` functions are part of the game definition combinator library. `gameState` is used to return the current game state, while `updateGameState` updates the current game state by applying a provided function. These functions have the following types.

```
gameState :: (Game g, GameM m g) => m (State g)
updateGameState :: Game g => (State g -> State g) -> ExecM g (State g)
```

Note that, the type of `gameState` is somewhat more general than the type of `updateGameState`. The type variable `m` in the type of `gameState` ranges over the monadic type constructors `ExecM` and `StratM`, which are both instances of `GameM`, while `updateGameState` applies only to computations in the `ExecM` monad. This means that access to the game's state is available from within both game definitions (`ExecM`) and strategy definitions (`StratM`), whereas modifying the game state is only possible from within game definitions—strategies may only manipulate the state indirectly, by playing moves.

Thus, the `matches` function returns the current number of matches on the table, while the `draw` function updates the state by removing the given number of matches from the table.

From here we can define a computation which determines when the game is over, that is, when there are no matches remaining on the table.

```
end = do n <- matches
         return (n <= 0)
```

And a function which executes a turn for a given player.

```
turn p = decide p >>= draw >> return p
```

On a player's turn, the player makes a decision and the move indicates how many matches to draw from the table. The reason this function returns the player's index will be seen in a moment. First, we define a computation which returns the payoff for the game, given the player who draws the last match.

```
payoff p = do n <- numPlayers
              return (loser n p)
```

The player who draws the last match loses, earning  $-1$  point, while other players win  $1$  point.

We now have everything we need to define the execution of the match game. Recall the `takeTurns` function introduced in Section 4.2 which takes two arguments, the first of which represents a player's turn, and the second is a function indicating when to stop looping through the players, applying the turn functions. We have just shown the definition of both of these functions for the match game, `turn` and `end`.

```
instance Game Matches where
  ...
  runGame = takeTurns turn end >>= payoff
```

The result of the `takeTurns` function is the value produced on the last player's turn, in this case, the player's index, which is passed to the `payoff` function indicating that the player taking the last match lost.

With the match game now defined, and with a representation that allows easy access to the state of state-based games, we can look towards defining strategies. First, we define two helper functions. The first returns the moves that are available to player.

```

moves = do n <- matches
  (Matches _ ms) <- game
  return [m | m <- ms, n-m >= 0]

```

This function extracts the available moves from the game representation, and filters out the moves that would result in a negative number of matches. Second, we define a function which returns a move randomly.

```
randomly = moves >>= randomlyFrom
```

The `randomlyFrom` function is part of Hagl's strategy combinator library, and returns a random element from a list.

Finally, a strategy for playing the match game is given below. The two-player match game is solvable for the first player, which means that the first player to move can always win if he or she plays correctly. The following player demonstrates one such solution.

```

matchy = "Matchy" `plays`
  do n <- matches
    ms <- moves
    let winning m = mod (n-1) (maximum ms + 1) == m
    in maybe randomly return (find winning ms)

```

The `winning` function, defined in this strategy, takes a move and returns true if it is a move that leads to an eventual win. The strategy plays a winning move if it can find one (which it always will as the first player), and plays randomly otherwise. While a proof that this player always wins when playing first is beyond the scope of this paper, we can provide support for this argument by demonstrating it in action against another player. The following player simply plays randomly.

```
randy = "Randy" `plays` randomly
```

We now run an experiment with these two players playing the match game against each other. The following command runs the game 1000 times consecutively, then printing the accumulated score.

```
> execGame (Matches 15 [1,2,3]) [matchy,randy] (times 1000 >> printScore)
Score: Matchy: 1000.0
      Randy: -1000.0
```

This shows that, when playing from the first position, the optimal strategy won every game. Even from the second position Matchy is dominant, since it only takes one poor play by Randy for Matchy to regain the upper hand.

```
> execGame (Matches 15 [1,2,3]) [randy,matchy] (times 1000 >> printScore)
Score: Randy: -972.0
      Matchy: 972.0
```

These types of simple experiments demonstrate the potential of Hagl both as a simulation tool, and as a platform for exploring and playing with problems in experimental game theory. By providing support for the large class of state-based games, and making it easier for users to define their own types of games, we greatly increase its utility.

## 6 Conclusions and Future Work

Hagl provides much needed language support to experimental game theory. As we extended the language, however, we discovered many problems related to bias in Hagl’s game representation. In this work we fundamentally redefine the concept of a Hagl game in a way that facilitates multiple underlying representations, eliminating this bias. In addition, we provide a combinator library which drastically simplifies the process of defining new classes of games, enabling users of the language to define new games and new game constructs as needed. Finally, we added explicit support for the broad class of games defined as transitions between states, resolving one of the primary weaknesses of earlier versions of the language.

In Section 4.1 we briefly introduced the idea of games which vary depending on their execution context. This represents a subset of a larger class of games which vary depending on their environment. Examples include games where the payoffs change based on things like the weather or a stock market. Since we have access to the `IO` monad from with execution monad, such games would be possible in Hagl. Other games, however, change depending on the *players* that are playing them. Auctions are a common example where, since each player may value the property up for auction differently, the payoff values for winning or losing the game will depend on the players involved. While it would be possible to simply parameterize a game definition with valuation functions corresponding to each player, it would be nice if we could capture this variation in the representation of the players themselves, where it seems to belong. Since auctions are an important part of game theory, finding an elegant solution to this problem represents a potentially useful area for future work.

Another potential extension to Hagl is support for human-controlled players. This would have many possible benefits. First, it would support more direct exploration of games; often the best way to understand a game is to simply play it yourself. Second, it would allow Hagl to be used as a platform for experiments *on* humans. Experimenters could define games which humans would play while the experimenters collect the results. Understanding how people actually play games is an important aspect of experimental game theory, and a significant niche which Hagl could fill.

This project is part of a larger effort to apply language design concepts to game theory. In our previous work we have designed a visual language for defining strategies for normal form games, which focused on the explainability of strategies and on the traceability of game executions [16]. In future work we hope to utilize ideas from both of these projects to make game theory accessible to a broader audience. One of the current limitations of Hagl, common in DSELs in general, is a presupposition of knowledge of the host language, in this case, Haskell. Our visual language is targeted at a much broader audience, but has a correspondingly smaller scope than Hagl. Somewhere in between there is a sweet spot. One possibility is using Hagl as a back-end for an integrated game theory tool which incorporates our visual language as part of an interface for defining, executing and explaining concepts in game theory.

## References

1. Fudenberg, D., Tirole, J.: *Game Theory*, xvii–xx, pp. 11–23. MIT Press, Cambridge (1991)
2. Nagel, R.: Unraveling in Guessing Games: An Experimental Study. *American Economic Review* 85, 1313–1326 (1995)
3. Ho, T., Camerer, C., Weigelt, K.: Iterated Dominance and Iterated Best-response in p-Beauty Contests. *American Economic Review* 88(4), 947–969 (1998)
4. Crawford, V.: Introduction to Experimental Game Theory. *Journal of Economic Theory* 104(1), 1–15 (2002)
5. Axelrod, R.: *The Evolution of Cooperation*. Basic Books, New York (1984)
6. Kendall, G., Darwen, P., Yao, X.: *The Prisoner’s Dilemma Competition* (2005), <http://www.prisoners-dilemma.com>
7. Walkingshaw, E., Erwig, M.: A Domain-Specific Language for Experimental Game Theory. Under consideration for publication in the *Journal of Functional Programming* (2008)
8. Erwig, M., Kollmansberger, S.: Functional Pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming* 16(01), 21–34 (2005)
9. Papadimitriou, C.: The Complexity of Finding Nash Equilibria. *Algorithmic Game Theory*, 29–52 (2007)
10. Gottlob, G., Greco, G., Scarcello, F.: Pure Nash Equilibria: Hard and Easy Games. In: *Proceedings of the 9th conference on Theoretical aspects of rationality and knowledge*, pp. 215–230. ACM, New York (2003)
11. Straffin, P.: *Game Theory and Strategy*, pp. 7–12, 65–80. The Mathematical Association of America, Washington (1993)
12. Chakravarty, M., Keller, G., Jones, S., Marlow, S.: Associated types with class. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 40, pp. 1–13. ACM, New York (2005)
13. Peyton Jones, S.L.: *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge (2003)
14. GHC: The Glasgow Haskell Compiler (2004), <http://haskell.org/ghc>
15. Groves, T., Ledyard, J.: Optimal Allocation of Public Goods: A Solution to the Free Rider Problem. *Econometrica* 45(4), 783–809 (1977)
16. Erwig, M., Walkingshaw, E.: A Visual Language for Representing and Explaining Strategies in Game Theory. In: *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pp. 101–108 (2008)

# A DSL for Explaining Probabilistic Reasoning

Martin Erwig and Eric Walkingshaw

School of EECS,  
Oregon State University

**Abstract.** We propose a new focus in language design where languages provide constructs that not only describe the computation of results, but also produce explanations of how and why those results were obtained. We posit that if users are to understand computations produced by a language, that language should provide explanations to the user.

As an example of such an explanation-oriented language we present a domain-specific language for explaining probabilistic reasoning, a domain that is not well understood by non-experts. We show the design of the DSL in several steps. Based on a story-telling metaphor of explanations, we identify generic constructs for building stories out of events, and obtaining explanations by applying stories to specific examples. These generic constructs are then adapted to the particular explanation domain of probabilistic reasoning. Finally, we develop a visual notation for explaining probabilistic reasoning.

## 1 Introduction

In this paper we introduce a domain-specific language for creating explanations of probabilistic reasoning together with a visual notation for the explanations. This DSL is an example of an *explanation-oriented language*, that is, a language whose primary goal is not to describe the computation of values, but rather the construction of explanations of how and why values are obtained.

In this introduction we will first introduce the idea of explanation-oriented programming in Section 1.1 and then motivate the application domain of probabilistic reasoning in Section 1.2. The specific components required for an explanation DSL for probabilistic reasoning will be outlined in Section 1.3.

### 1.1 Toward Explanation-Oriented Programming

A program is a description of a computation, and a computation can be broadly viewed as a transformation of some input to some output. The produced output typically is a result sought for some problem. In this sense, programming is about producing results. This view is appropriate as long as the produced results are obviously correct or can be trusted for other reasons, which is not always the case.

There are many instances in which programs produce unexpected results. These results are not necessarily incorrect, but a user might not understand how the result was produced and/or cannot judge whether or not the result is correct.

In situations like these an explanation of how the result was obtained or why it is correct would be very helpful. Unfortunately, however, those explanations are not easy to come by. In many cases, one has to go through a long and exhausting debugging process to understand how a particular result was produced.

One reason for this situation is that explanations are not the objects of concern of programming languages. Therefore, explanation tools, such as debuggers, have to be designed as an add-on to programming languages, mostly as an afterthought. Since an explanation concept is missing in the design of programming languages, explanation tools are difficult to integrate, and this mismatch forces debuggers to reflect the notion of computation realized in the design of the programming language, which in many cases leads to “explanations” that are difficult to produce and have low explanatory value. In particular, traces generated through debugging are difficult to reuse and often impossible to combine to produce other explanations.

The idea of *explanation-oriented programming* is to shift the focus toward explanations of the described computations, so that in devising abstractions and constructs, language design does not only center on producing final values, but also on explanations of how those values are obtained and why they are correct.

This idea has a huge potential to innovate language design. In particular, with a large and growing group of end-user programmers (creating, reusing, and modifying spreadsheets, web macros, email filters, etc.) [28] there is a growing need to provide explanations in addition to effective computations. Besides the obvious application of supplementing or replacing current debugging approaches, there is also a huge potential to define domain-specific languages that can be employed to create explanations in specific application areas that can be customized and explored by users, ranging from the explanation of mechanical devices to medical procedures, or defining explanations for all kinds of scientific phenomena (in natural and social sciences).

One such application area is probabilistic reasoning, which is generally not well understood and can thus benefit greatly from a language to create corresponding explanations. A domain-specific language for creating explanations of probabilistic reasoning is thus one example of an explanation-oriented language.

## 1.2 Understanding Probabilistic Reasoning

Probabilistic reasoning is often difficult to understand for people that have little or no corresponding educational background. Even rather simple questions about conditional probabilities can sometimes cause confusion among lay people, and disbelief about answers to probability questions in many cases remains despite elaborate justifications.

Consider, for example, the following question: “Given that a family with two children has a boy, what is the probability that the other child is a girl?” Many people respond that the probability is 50%, whereas it is, in fact,  $66\frac{2}{3}\%$ .

Given the importance of probabilistic and statistical reasoning in all areas of science and in many practical questions concerning modern societies (insurances, effectiveness of medical procedures, etc.), the question is what can be done to

help this situation. In addition to trying to improve education in general and the teaching of basic probability theory in schools in particular, a more immediate approach is to provide concrete help with specific probability problems.

Currently, somebody who wants to understand the answer to the above question has several options: First, they could ask somebody for an explanation. Second, they could try to find explanatory material on their own, for example, by searching the web or maybe even by looking into a textbook. Third, they could give up and not understand the problem, which is likely to happen if no one is around who could take the time to provide a personal explanation, or if the explanation found on a web site (or in a textbook) does not provide the right amount of information on the right level.

In general, a personal explanation can be expected to be the best option in most cases since it allows the explainer to rephrase explanations, to answer potential questions, make clear the underlying assumptions, and also to use different examples to provide further illustration. Unfortunately, personal explanations are a comparatively scarce resource. Not only are they not always available, but they also have a very low degree of shareability and reusability. In contrast, an explanation provided on a web site, say, can be accessed almost at any time, by almost anyone from anywhere, and as often as needed, that is, it has very high availability, sharability, and reusability. On the other hand, explanations on web sites tend to be rather static and lack the adaptive features that a personal explanation provides. In many cases, one has to read a whole text, which might for a particular reader be too elaborate in some parts, which can potentially cause exhaustion, and too short in other aspects, which can cause frustration.

The goal of our DSL is to combine the positive access aspects of electronically available explanations with the flexibility that personal explanations can offer and to provide a language to create widely accessible and flexible explanations. Our goal is *not*, however, to replace personal or web-based explanations, but to complement them. For example, a teacher or a web page can provide an explanation in the traditional sense and then offer a structured, explorable explanation object as an additional resource. Another combined use is that different views of an explanation object can be employed as illustrations as part of a personal or written explanation.

### 1.3 This Paper

The design of a successful explanation DSL ultimately requires several different components.

- Language constructs to build explanations
- A visual representation of probability distributions and transitions
- Interaction concepts to support the dynamic exploration of explanations
- A visual explanation programming language to allow domain experts who are not programmers to develop explanations

In this paper we will focus on the first two aspects. First, we investigate the notion of an explanation in Section 2 and try to identify an explanation concept that is most suitable for our task of defining explanation DSLs. Based on those insights we then design in Section 3 a domain-specific embedded language in Haskell [14] for building explanations. This DSL can be employed to create explanations in a variety of domains. We then specialize some parts of the DSL to the domain of probabilistic reasoning in Section 4. The visual representation of those explanations is introduced in Section 5. After the discussion of related work in Section 6 we present some conclusions and goals for future work in Section 7.

The contributions of this paper are these.

- A DSL for building (generic) explanations
- A DSL for building explanations of probabilistic reasonings
- A prototypical implementation of both DSLs as an embedding into Haskell
- A visual notation for probabilistic reasoning explanations

## 2 Theories of Explanation

The question of what an explanation is has been discussed by philosophers of all times [26], dating back to Aristotle. Despite all these effort, the concept of explanations remains elusive, and an all-encompassing theory that could be agreed upon does not yet exist.

One of the most influential approaches in the 20th century was the so-called “deductive nomological” theory of explanation by Carl Hempel [12]. According to that theory an explanation is essentially an inference rule that has laws and facts as premises and the explanandum<sup>1</sup> as a conclusion. This view is very attractive and has initiated renewed interest and research in the subject. However, Hempel’s theory has been criticized, in particular, for being too permissive and thus failing to characterize the essential aspects of explanations.

One problem is the *overdetermination* of inference rules, that is, by adding irrelevant facts or laws to an inference rule, the inference rule is still considered to be an explanation even though the added facts and laws are not explanatory and, in fact, weaken the explanation (regardless of their truth). For example, a law that says that men taking birth control pills do not get pregnant does not explain the fact that a specific man did not become pregnant as well as the simpler, more general law that men do not become pregnant. The other problem with Hempel’s theory is that it fails to capture *asymmetry* in explanations. For example, an inference rule that can explain the length of a shadow by the height of a flagpole and the position of the sun can also be used to “explain” the height of the flagpole from the length of the shadow even though this would not be accepted as an explanation.

In search for a unifying theory of explanation, the problem has been approached not only from philosophy, but also from other related areas, including cognitive science [13] and linguistics [2].

---

<sup>1</sup> The thing that is to be explained.

## 2.1 Criteria for an Explanation Model

An initial survey of the literature quickly reveals that a comprehensive overview over the field is well beyond the scope of this paper. Therefore, in order to have an effective approach to finding an appropriate explanation theory, we have devised some criteria that an explanation theory should fulfill. Traditionally, the search for an explanation model has been driven by the goal to achieve generality. However, an explanation model that is adequate in most cases is sufficient for our purpose, in particular, if it supports the definition of a DSL for building explanations. The main criteria for selecting a working model of explanation are the following. An explanation model should be:

- *Simple and intuitive.* The model should be understandable by ordinary people, not just philosophers and scientists, because explanations expressed in this model are to be consumed by ordinary people.
- *Constructive.* The model should identify the components of an explanation and how they are connected or assembled to support the definition of a DSL.

The need for simplicity rules out some advanced statistical models [25,21], models based on physical laws [27] or process theory [4]. Unificationist models [18] do not provide a constructive approach, and they are quite complicated too.

## 2.2 Causal Explanations

We believe the most promising explanation theory is based on the notion of *causation*, that is, to explain a phenomenon  $A$  means to identify what caused  $A$ . The idea that explanations reveal *causes* for *why* things have happened goes back to Plato [24].

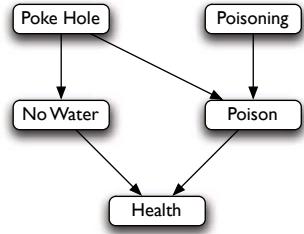
The theory put forward by Woodward [30] ties the notion of a cause to the concept of *manipulability*, which, simply said, allows one to ask the question “What would happen to the explanandum if things had been different?”. In this way, manipulating a cause shows effects on the explanandum and thereby places it into a context of alternative outcomes. Similar emphasis on the importance of an explanations to reveal opportunities to intervene to change the outcome can be found in the work of Humphreys and von Wright [15,29].

An argument for the importance of the manipulability aspect of explanations is that the motivation to understanding something comes from the promise of using the knowledge to manipulate nature for one’s own advantage. This has been nicely demonstrated by Dummett’s “intelligent tree” thought experiment [5]: Suppose we humans were intelligent trees, capable of only passive observations. Would we have developed the concepts of causation and explanation as we have them now?

In Woodwards model, an explanation is represented by a directed graph with variables as nodes and directed edges between nodes  $X$  and  $Y$  if  $X$  has a direct effect on  $Y$ . A similar representation is used by Pearl [21]. Essentially, each variable in such a graph corresponds to a type of possible values (typically boolean), and a directed edge from  $X$  to  $Y$  corresponds to a function of type  $X \rightarrow Y$ .

Therefore, an explanation graph represents a set of interconnected functional relationships. This is essentially the same view as structural equation modeling, which dates back almost 90 years [31].

As an example, consider the desert traveler problem, in which a traveler takes a bottle of water on a trip through the desert. Two people try to kill the traveler: One pokes a hole in the bottle, the other one poisons the water. When the traveler later gets thirsty, he will find the bottle empty and dies of dehydration. A graph representation of this story is shown in Figure 1 on the right. This example is used to test theories about causation and whether they can properly attribute the blame for the traveler's death.



**Fig. 1.** Causal graph for the desert traveler problem

### 3 The Story-Telling Model of Explanations

It seems that we could use the graph representation suggested by Woodward and Pearl [30,21] directly as a representation for explanations. However, a potential problem with the graph representation is that it puts an additional navigational burden of the user. In particular, the user has to decide which nodes, edges, or paths to look at in which order, which can be a frustrating experience.

Therefore, we have aimed for a more linear representation. The idea is that an explanation consists of a sequence of steps that guide the user from some initial state to the explanandum using a series of well-understood operations to transform the explanation state. Of course, the linearity requirement might mean more work for the explanation programmer who has to define the state in a way that is general enough to include all aspects of a potentially heavily branched graph representation. On the other hand, if done well, this effort pays off since it makes the life for the explanation consumer much easier.

This task of creating an explanation is similar to writing a story in which events happen that transform the state of the story to a final outcome, the explanandum.<sup>2</sup> There is empirical evidence that presenting facts and explanations in the form of a story makes them more convincing and understandable [22]. We will therefore employ the story-telling metaphor for explanations in the design of our explanation DSL.

Each event denotes a change and is represented by a function that is annotated by a textual comment to be used in printing explanations. So a simplified view of an event is given by the following Haskell type definition.

```
type Event a = Annotate (Chg a) -- preliminary definition
```

The type constructors `Annotate` and `AnnotateP` are used to attach a `Note` to a single or a pair of values, respectively.

<sup>2</sup> Friedman explicitly calls explanations “little stories” [9].

```
type Note = String
type Annotate a = (Note,a)
type AnnotateP a = (Note,a,a)
```

The type constructor `Chg` simply represents a function type. It is one of several type constructors to be used (and explained) in the following.

```
data Id a = I a
data Chg a = C (a -> a)
data Agg a = A ([a] -> a)
data Sum a = S [a] a
```

The definition of `Event` can be generalized in two ways. First, in addition to a linear sequence of events we also consider stories that can branch into separate substories and add two constructors for branching and joining story lines. Second, a story or explanation can be viewed on two different, but closely related levels. On the one hand, we can talk about the sequence of events. On the other hand, we can look at the sequence of results that are produced by the events when they happen to a particular (initial) state. Therefore, we define a general type `Step`, of which the types for events and states are instances.

These requirements are captured by the following type definition. The parameter type `s`, which is a type constructor, defines what is represented in each step of a story, and `j` defines how results from multiple stories can be combined. In an `Event` we store a function in each step and an aggregation function for joins. In contrast, in a state we just store a value for each step and a summary of the values of the branches and the computed aggregate value for joins.

```
data Step s j a = Linear (Annotate (s a))
                 | Branch (AnnotateP [Step s j a])
                 | Join (Annotate (j a))

type Event a = Step Chg Agg a
type State a = Step Id Sum a
```

A story is a list of events, and an explanation is given by a list of states. The construction of an explanation happens by first building a story and then instantiating it to an explanation.

```
type Story a = [Event a]
type States a = [State a]

newtype Explanation a = E (States a)
```

We can illustrate the operations for building stories with the desert traveler example. We represent two aspects of the traveler, his health and his water bottle.

```
type Traveler = (Health,Bottle)
```

This definition shows how a fan-out graph explanation can be linearized by defining a richer story state. The traveler's health is given by two states, dead or alive, where we are interested in the case of death, how he died. The water bottle can be empty, full, or leaking, and we have to indicate in the non-empty cases whether the bottle contains water or poison.

```
data Health = Alive | Dead Note
data Fluid = Water | Poison
data Bottle = Full Fluid | Leak Fluid | Empty
```

To define a story that explains the traveler's death, we have to define four events: poisoning the water, poking a hole in the bottle, traveling through the desert, and (attempted) drinking.

In many cases, an aggregated state definition, such as `Traveler`, requires operations that allow the application of events to parts of the state. In the example, poking a hole or poisoning are events that actually apply to a bottle, but that will need to be defined to transform the whole traveler state. This state access is provided by the following function.

```
inBottle :: (Bottle -> Bottle) -> Traveler -> Traveler
inBottle f (h, b) = (h, f b)
```

Now we can define the poisoning event as a function `putPoison` that has an effect on a bottle, but that is lifted to transform a traveler state.

```
poison :: Event Traveler
poison = "Poison the water" ## inBottle putPoison
        where putPoison (Leak _) = Leak Poison
              putPoison _           = Full Poison
```

The annotation of the lifted function is performed with the smart constructor `##`, which is defined as follows.

```
(##) :: Note -> (a -> a) -> Event a
n ## f = Linear (n, C f)
```

The events of poking a hole in the bottle and the potential draining of the bottle's content during the travel are defined in a completely analogous ways.

```
poke :: Event Traveler
poke = "Poke hole in bottle" ## inBottle pokeHole
      where pokeHole (Full f) = Leak f
            pokeHole b        = b

travel :: Event Traveler
travel = "Travel through desert" ## inBottle drain
        where drain (Leak _) = Empty
              drain b        = b
```

The final event of the story is when the traveler gets thirsty and wants to drink.

```
quench :: Event Traveler
quench = "Traveler gets thirsty and tries to drink" ## drink
        where drink (h,Full Water) = (h,Empty)
              drink (h,Leak Water) = (h,Empty)
              drink (h,Empty)      = (Dead "thirst",Empty)
              drink (h,_)          = (Dead "poison",Empty)
```

A story about the desert traveler is given by a list of events. For the original story as told in Section 2.2 we have:

```
story :: Story Traveler
story = [poison,poke,travel,quench]
```

We can generate an explanation from the story by successively applying the story events to an initial state value.

```
desertTravel :: Explanation Traveler
desertTravel = explain story 'with' (Alive,Full Water)
```

The function `explain` is actually just a synonym for the identity function and is used only as syntactic sugar. The function `with` threads a state value through a list of events and builds a corresponding trace of resulting state values that constitute the explanation.

```
with :: Story a -> a -> Explanation a
```

If we evaluate `desertTravel`, we obtain a description of the events and how they lead to the death of the traveler.

```
--{Start}-->
(Alive,Full Water)
--{Poison the water}-->
(Alive,Full Poison)
--{Poke hole in bottle}-->
(Alive,Leak Poison)
--{Travel through desert}-->
(Alive,Empty)
--{Traveler gets thirsty and tries to drink}-->
(Died of thirst,Empty)
```

This result is obtained by simply pretty printing the states and interjecting the notes associated with the events.

Arguably, the trace “explains” how the traveler died, but it is not necessarily obvious that it is the poking event that is responsible for the death. To strengthen the explanation in this regard we could add a branch to the story that shows that the poisoning has no effect on the outcome. Adding a version of the story to the explanation that omits the poisoning event achieves this because the outcome stays the same.

```

poisonEffect :: Explanation Traveler
poisonEffect =
  explain [Branch ("Effect of poisoning",story,tail story)]
  'with' (Alive,Full Water)

```

The evaluation of this explanation will produce a pair of traces that both end with the same result (`Died of thirst,Empty`), which thus illustrates that poisoning has no causal effect on the death. We will present an output that includes an explanation with a `Branch` constructor later in Section 4.3.

The `Branch` construct has much the same purpose as the `do` operation presented in [21], which permits the representation of interventions in causal networks. The difference is that the `do` operation changes a probability network, whereas `Branch` combines two explanations representing both situations before and after the intervention, similar to the twin network representation, also described in [21].

## 4 Explanations of Probabilistic Reasoning

Explanations for probabilistic reasoning are built using, in principle, the same operations as introduced in Section 3. However, the domain of probability distributions presents specific challenges, and opportunities, to adapt the story-telling model to this particular domain. By specializing the generic explanation DSL with respect to another domain, probabilistic reasoning, we obtain a “domain-specific domain-specific language”.

This section introduces the components of our DSL in several steps. In Section 4.1 we review an approach to computing with probabilistic values that forms the basis for the representation of probability distributions. In Section 4.2 we adapt the explanation DSL to the specific domain of probabilistic values, and we show example explanations in Section 4.3.

### 4.1 Computing with Probabilistic Values

In previous work we have presented an approach to represent probabilistic values explicitly as probability distributions [7]. Any such probability distribution is basically a list of values paired with associated probabilities.

Probability distributions can be constructed using a variety of functions. For example, the function `uniform` takes a list of values and produces a distribution of the values with equal probabilities.

```
uniform :: [a] -> Dist a
```

Coin flips or die rolls can be conveniently expressed using `uniform`. Or, to use the example we started in Section 1.2, we can represent the birth of a child as a probability distribution.

```
data Child = Boy | Girl

birth :: Dist Child
birth = uniform [Boy,Girl]
```

The evaluation of a probabilistic value will show the elements of the distributions together with their probabilities.

```
> birth
  Boy 50%
  Girl 50%
```

It turns out that the `Dist` type constructor is actually a monad [10,23]. The return function produces a distribution of a single value (with 100% probability). The bind operation takes a probability distribution `d` of type `Dist a` and function `f` of type `a -> Dist b` and applies `f` to all elements of `d`. The list of distributions obtained in this way will be combined into one distribution, and the probabilities of the elements will be adjusted accordingly. The meaning of the bind operation `>>=` can probably be best explained by showing an example. To compute the probability distribution of families with two children we can use bind to extend the distribution `birth` by another birth. However, we have to represent the second birth as a function that transforms each child value from the first distribution into a pair of children in the produced distribution.

```
secondChild :: Child -> Dist (Child,Child)
secondChild c = uniform [(c,Boy),(c,Girl)]

twoKids : Dist (Child,Child)
twoKids = birth >>= secondChild
```

Evaluation of `twoKids` results in the following distribution.

```
(Boy,Boy) 25%
(Boy,Girl) 25%
(Girl,Boy) 25%
(Girl,Girl) 25%
```

Another operation on probability distributions that will be employed in explanations of probabilistic reasoning is the filter operation `|||` that restricts a probability distribution by a predicate.

```
(|||) :: Dist a -> (a -> Bool) -> Dist a
```

The definition of `|||` selects all the elements that pass the filter predicate and then scales the probabilities of these elements equally so that they sum up to 100%. The Haskell definition requires several helper function and is not so important; it can be found in the distribution of the PFP library.<sup>3</sup> The operation

---

<sup>3</sup> See [eecs.oregonstate.edu/~erwig/pfp/](http://eecs.oregonstate.edu/~erwig/pfp/)

`|||` computes a distribution of all *conditional probabilities* with respect to the filter predicate.

For example, the riddle from Section 1.2 asks us to consider only families with boys. This constraint can be expressed by filtering `twoKids` with the predicate that one kid must be a boy.

```
oneIsA :: Child -> (Child,Child) -> Bool
oneIsA c (c1,c2) = c1==c || c2==c
```

The filtered distribution of conditional probabilities looks then as follows.

```
> twoKids ||| oneIsA Boy
(Boy,Boy) 33%
(Boy,Girl) 33%
(Girl,Boy) 33%
```

This distribution almost tells the answer to the riddle. Formally, the result can be obtained by using a function to compute the total probability of an event that is given by a predicate.

```
(??) :: (a -> Bool) -> Dist a -> Float
```

The probability that in a family that has one boy the other child is a girl is thus obtain as follows.

```
> oneIsA Girl ?? twoKids ||| oneIsA Boy
67%
```

We will present a generalization of `??` in the next subsection.

## 4.2 Adapting Explanations to Probabilistic Values

The explanation of the riddle that was provided in Section 4.1 by a sequence of function definitions intertwined with textual comments is, so we hope, effective and has high explanatory value. The reason may be that it follows to some degree the story-telling schema propagated earlier. In any case, it is, however, not represented in a formal language and thus is not well suited for reuse, manipulation, or exploration.

When we try to express the boy/girl riddle as a story we encounter the difficulty that the types of events, stories, and explanations have only one type parameter, which means, for example, that we cannot directly reuse the function `secondChild` to represent an event because it changes the type of the distribution.

There are two ways to deal with this problem. One approach is to generalize the explanation types to two type parameters to allow for type changes in explanations. The alternative is to define a union type, say `Family`, that accounts for all possible types encountered during the course of the explanation.

It seems that the two-type-parameter approach is preferable since it is less constraining and promises more modularity and a higher degree of reuse. However, the problem with this approach is that it requires the use of existential types, which would make our approach to grouping probability distributions (to be described below) impossible. We have therefore adopted the simpler solution and require a single type for the explanandum.

For the riddle example the definition of such a union type is rather obvious.

```
data Family = NoKids | One Child | Two Child Child
```

Now before we go on to define family events, we extend the representation of probability distributions by an optional function to group values in a distribution into different categories. We call the new type `VDist` since the grouping is an example of a view on the distribution.

```
data VDist a = Group (Dist a) (Maybe (a -> Int))
```

At this point we only consider grouping views, but there are other kinds of views that we might want to add in the future to the type definition.

We also specialize the event, story, and explanation types to `VDist` distributions by reusing the definitions given in Section 3 (assuming they are defined in a module with name `E`).

```
type Event a = E.Event (VDist a)
type Story a = E.Story (VDist a)
type Explanation a = E.Explanation (VDist a)
```

We also define the following type synonyms for events that emphasize special functions of probabilistic events.

```
type Generate a = Event a
type Filter a = Event a
type Group a = Event a
```

The presence of a grouping function does not change the distribution of values, it just provides a particular presentation of the distribution, which is reflected in the fact that the effect of the grouping is noticeable when distributions are printed.

```
instance Show a => Show (VDist a) where
  show (Group d Nothing) = show d
  show (Group d (Just f)) = show (partition f d)
```

Without a grouping function, distributions are shown as normal. However, when a grouping function is present, values of the distribution are grouped into lists by the following function.<sup>4</sup>

<sup>4</sup> The actual argument type of the result distribution is not `[a]` but a specialized list type `ZoomList a` that allows a more flexible and condensed printing of large lists, which occur frequently in probability distributions.

```
partition :: (a -> Int) -> Dist a -> Dist [a]
```

Since the distributions in explanations are all wrapped by a `Group` constructor, we need auxiliary functions to lift functions defined on distributions to work on the new type and to embed distributions in the new type.

```
vmap :: (Dist a -> Dist a) -> VDist a -> VDist a
vmap f (Group d g) = Group (f d) g
```

```
toV :: Dist a -> VDist a
toV d = Group d Nothing
```

With the help of `vmap` we can easily adapt the annotation function `##` introduced in Section 3. Since we have different kinds of transitions between probability distributions, we introduce several new versions of annotations. In particular, we have distribution generator events that are defined using the monadic bind operation and distribution filter events that are directly based on the probabilistic filter operation.

```
(##>) :: Note -> (a -> Dist a) -> Generate a
n ##> f = n ## vmap (>>= f)
```

```
(##|) :: Note -> (a -> Bool) -> Filter a
n ##| p = n ## vmap (||| p)
```

We also specialize the `with` function that instantiates stories with examples.

```
with :: Story a -> Dist a -> Explanation a
s 'with' d = s 'E.with' (toV d)
```

### 4.3 Example Explanations

The explanation for the boy/girl riddle is now straightforward. First, we define the individual events, then we combine the events into a story, and finally we instantiate the story with an example to obtain an explanation.

The two generators are variations of the `birth` distribution, rewritten as argument functions for the bind operation.

```
firstChild :: Generate Family
firstChild = "First child" ##> \_>uniform [One Boy,One Girl]

secondChild :: Generate Family
secondChild = "Second Child" ##>
\(One c)->uniform [Two c Boy,Two c Girl]
```

With these two events we can define a story and explanation of a two-children family. Note that `certainly` is simply a synonym for `return`.

```

twoKids :: Story Family
twoKids = [firstChild,secondChild]

family :: Explanation Family
family = explain twoKids 'with' certainly NoKids

```

The other two elements of the riddle are the restriction to families that have a boy and the focus on the families that also have a girl. These two events represent, respectively, a filter and a grouping operation on distributions.

```

hasA :: Child -> Family -> Bool
hasA c (Two c1 c2) = c==c1 || c==c2

oneIsBoy :: Filter Family
oneIsBoy = "Only families that have a Boy" ##| hasA Boy

otherIsGirl :: Group Family
otherIsGirl = "Other child is a girl" ##@ hasA Girl

```

The special annotation function `##@` defines the grouping function for a distribution. In contrast to filtering, the argument function (which is here a predicate) does not change the probability distribution, but rather creates groups of values together with their accumulated probabilities.

```

(##@) :: Grouper b => Note -> (a -> b) -> Group a
n ##@ f = n ## groupBy f

groupBy :: Grouper b => (a -> b) -> VDist a -> VDist a
groupBy f (Group d _) = Group d (Just (toInt . f))

```

The type class `Grouper` ensures the availability of the function `toInt`, which turns the grouping function into one of type `a -> Int`, the type expected by `partition`. Finally, we can express the whole riddle story by the events defining the `twoKids` family followed by the boy filtering and then by focusing on the group containing family with girls. The solution to the riddle can be found in the resulting distribution of the explanation, which is obtained as an instance using the `with` function.

```

riddle :: Story Family
riddle = twoKids ++ [oneIsBoy,otherIsGirl]

solution :: Explanation Family
solution = explain riddle 'with' certainly NoKids

```

The explanation, shown in Figure 2, illustrates how the result is obtained from an initial state through the successive application of simple events.

Once the structure of the explanation has been identified, it is easy to write transformations that change the examples, that change generators (for example, replace children with coin flips), or that provide more or less detailed sequences of steps. Such a collection of explanations constitutes an explorable explanation object that, made accessible through a corresponding user interface, provides a flexible, yet structured explanation of the particular problem at hand.

As another example we consider an explanation of the so-called “Monty Hall problem” in which a game show contestant is presented with three doors, one of which hides a prize. The player chooses one of the doors. After that the host opens another door that does not have the prize behind it. The player then has the option of staying with the door they have chosen or switching to the other closed door. This problem is also discussed in [7,11,20]. Most people believe that switching doors makes no difference. However, switching doors *does* make a difference—it doubles the chances of winning from  $33\frac{1}{3}\%$  to  $66\frac{2}{3}\%$ .

To build an explanation we first define a data type to represent the different states of the story. Each door means a potential win or loss for the candidate unless it is opened, in which case it is out of the game.

```
data Door = Win | Loss | Open
type Doors = [Door]
```

Since the opening of doors is captured by the `Door` type, the game needs to distinguish only two states, the initial situation of the prize being hidden, and then the selection of a door through the contestant.

```
data Game = Hide Doors | Select Int Doors
```

The story starts with a situation that can be represented by three different door lists with `Win` in one place. Then the first event, selecting the door is represented by a generator that creates for each door list three possible selections, represented by an integer between 0 and 2. (Starting with 0 accommodates standard list indexing convention.)

```
> solution
--{Start}-->
No children 100%
--{First child}-->
Boy 50%
Girl 50%
--{Second Child}-->
Boy Boy 25%
Boy Girl 25%
Girl Boy 25%
Girl Girl 25%
--{Only families with a boy}-->
Boy Boy 33%
Boy Girl 33%
Girl Boy 33%
--{Other child is a girl}-->
[Boy Girl,Girl Boy] 67%
[Boy Boy] 33%
```

**Fig. 2.** Textual explanation of the boys and girls riddle

```

hiddenPrize :: Dist Game
hiddenPrize = uniform $ map Hide
    [[Win, Loss, Loss], [Loss, Win, Loss], [Loss, Loss, Win]]]

selectDoor :: Generate Game
selectDoor = "Candidate selects closed door" ##>
    \ (Hide ds) -> uniform [Select n ds | n <- doors]

doors :: [Int]
doors = [0..2]

```

When opening a door, the game master has to avoid the door hiding the prize and the door selected by the contestant (represented by the first parameter to `openBut`). The function `avoid` generates all possible doors for the game master.

```

openDoor :: Generate Game
openDoor = "Game master opens a randomly chosen no-win door" ##>
    \ (Select n ds) -> uniform $ map (Select n) (openBut n ds)

openBut :: Int -> Doors -> [Doors]
openBut n ds = [upd i Open ds | i <- avoid Win n ds]

avoid :: Door -> Int -> Doors -> [Int]
avoid d n ds = [i | i <- doors, i /= n, ds !! i /= d]

```

The grouping of situations into wins and losses can be expressed by a simple predicate on door lists.

```

winning :: Group Game
winning = "Group outcomes by winning and losing" ##@>
    \ (Select n ds) -> ds !! n == Win

```

Finally, we define two events for switching and non-switching and place them in a branching event.

Not switching is an annotated identity function, whereas switching means to select the door that has not been opened. This event is implemented by a function that simply transforms one game into another and does not generate multiple possibilities. This means that when we reuse the `avoid` function, we have to extract an element from the produced list. Since we know that the list can have only one element in this case, we can obtain it using the function `the`. Moreover, we need a new annotation function that lifts into the `Event` type, for which we also use the synonym `Map`.

```

stay :: Map Game
stay = "Don't switch door" ##* id

```

```

switch :: Map Game
switch = "Switch door" #**
  \Select n ds->Select (the (avoid Open n ds)) ds
  where the [x] = x

(##*) :: Note -> (a -> a) -> Map a
n ##* f = n ## vmap (fmap f)

```

The definition of `##*` exploits the fact that probability distributions are also defined as functor instances.

The contemplation of both alternative story lines can be expressed as a branching event as follows.

```

contemplate :: Event Game
contemplate = Branch("Consider switching doors", [stay], [switch])

```

We can now create an explanation from all these events. However, we can observe (either by inspecting the produced explanation or by simply thinking about the story) that it does not really matter which door hides the prize. We might want to exploit this knowledge and integrate it into the explanation since it helps to reduce the number of possibilities to be inspected. This idea can be realized very easily by a function that picks a particular element of a probability distribution and sets its probability to 100%.

```
forExample :: Int -> Event a
```

It is up to the explanation programmer to ensure that the explanation for the picked example is isomorphic to the other elements in the distribution. In future work we plan to extend the function `forExample` by requiring evidence that can be effectively checked against the other elements of the probability distribution.

Finally, the explanation for the Monty Hall problem can be put together as follows.

```

montyHall :: Explanation Game
montyHall = explain [forExample 1,selectDoor,openDoor,
                     winning,contemplate] 'with' hiddenPrize

```

The explanation that is produced is shown in Figure 3. In the textual representation \$ means win, x means loss, and a blank denotes an open door. Underlining indicates the door selected by the contestant.

Even though the textual representation is quite frugal, it still illustrates the progression of the probability distribution along the story. In particular, the grouping step shows that before the question of switching, the candidate's chances of losing are about 67%, and, of course, this probability doesn't change if the candidate doesn't switch. In contrast, the switching event in the second branch illustrates that switching will always move from winning to losing and from losing to winning, that is, the 67% losing probability turns into a 67%

```

> montyHall
  --Start-->
$xx 33%
x$x 33%
xx$ 33%
  --Select 1st value as representative-->
$xx 100%
  --Candidate selects closed door-->
$xx 33%
$xx 33%
$xx 33%
  --Game master opens non-chosen no-win door-->
$ x 17%
$x 17%
$ x 33%
$ x 33%
  --Group outcomes by winning and losing-->
[$ x,$x ] 33%
[$x ,$ x] 67%
  BRANCH Consider switching doors
<<<
  --Don't switch door-->
[$ x,$x ] 33%
[$x ,$ x] 67%
=====
  --Switch door-->
[$ x,$x ] 33%
[$x ,$ x] 67%
>>>

```

**Fig. 3.** Textual explanation of the Monty Hall problem

winning probability. The representation also shows why this is the case, namely, because the underscore has only one other case to switch to.

Laws that relate different stories with respect to their produced results can be employed to turn one explanation into several explanations to be explored by the user. We briefly illustrate this aspect with one example.

For example, given a function `result :: Explanator a -> [a]` that yields the result(s) of an explanation<sup>5</sup>, we can observe the following invariant. If `f` is a total function on `T` and `s :: Story T` is well formed (that is, any event directly following a branch event is a join), then for any story decomposition `s1 ++ s2 == s` we have:

```
result (s 'with' x) == result ((s1++["" ##@ f]++s2) 'with' x)
```

In other words, we can insert a grouping anywhere into an explanation without changing the resulting probability distribution.

---

<sup>5</sup> The list type is needed to aggregate results from branches.

For explanations this means that we can move groupings around to improve the explanatory value of explanations. (But we have to be careful to move groupings into the branches of a `Branch` constructor.)

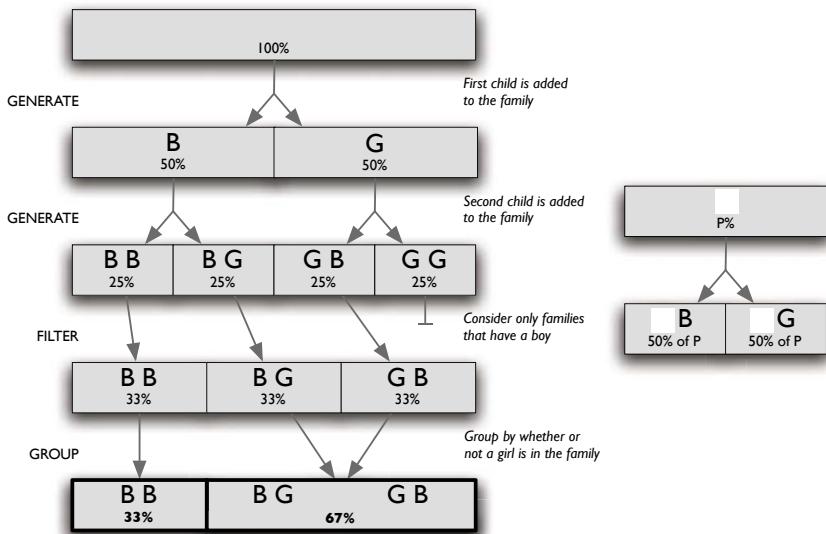
In the Monty Hall example this result means that we can safely move the grouping before the door opening or into the branch after the switching/staying, which yields alternative perspectives for the explanation that could be offered for exploration by the user.

## 5 Visual Explanations of Probabilistic Reasoning

The visual notation for explanations of probabilistic reasoning was *not* simply designed as a replacement for the clunky textual representation. In fact, we started the design of the DSL with the visual notation. The two major motivations were the following. (The first aspect is part of future work and will therefore not be discussed in this paper.)

- Providing a basis for interactive explanation exploration.
- Adding explanatory power by employing the two metaphors of *spatial partitions* and *story graphs*.

The explanatory value of spatial partitions and story graphs can be best explained by an example. Figure 4 shows on the left the visual representation of the explanation for the boy/girl riddle from Section 4.3.



**Fig. 4.** Visual explanation of the boys and girls riddle

A probability distribution is represented as a partition of a horizontal area in which the area of each block is proportional to the probability of the represented value. In this way spatial partitions capture the notion that a probability distribution is, in fact, a partitioning of the “space” of all possibilities. It treats the probability space as a resource that is split by generators and merged by groupings. Moreover, it illustrates indirectly how space that is filtered out is redistributed to the remaining blocks of the partition.

Each partition of an explanation is linked to its successor by basically two kinds of directed edges. First, filter and group edges link blocks of two partitions that either have identical values (in the case of filter) or that are considered equivalent (by the grouping function in the case of grouping). Second, generator edges are, from a graph-theoretic point of view, sets of edges that lead from one block to a set of blocks that are created from the value in the source block by a generator. They are represented visually with one shared tail and as many heads as there are generated values/blocks. Similarly, “for example” edges are sets of edges that map a set of blocks that are equivalent with respect to the continued probabilistic reasoning to one block chosen as an example. The chosen example is indicated by a fat tail (see Figure 5 for an example).

Since generators are more complicated events than filters or groupings, generator edges are necessarily more abstract and less informative about what exactly happens in that generating step. We have therefore also a notation that shows the effect of a generator operation in more detail. One example is shown in Figure 4 on the right. Showing this expanded notation everywhere would make the visual notation too noisy, so we envision it as part of the interaction capabilities of these explanations to expand individual operations into such a more detailed representation. We have employed a very similar strategy successfully in another visual language [8]. The notation can also be the basis for a visual explanation *programming* language in which explanations can be assembled on the fly using distributions and generators and then modified further through filters and groupings.

The chosen graph representation for linking the spatial partitions provides another opportunity for explanations. A user could mark a particular value anywhere in an explanation, and the system can trace the evolution of that value forward and backward, also possibly producing animations of how such values flow through the graph, changing the probabilities on their way.

In Figure 5 we show the story graph representation of the explanation for the Monty Hall problem. In anticipation of a visual user interface, we assume that only one branch is “active” or “selected” at a time so that there is only one set of edges that connect blocks into one branch to avoid confusion. We have added another `Map` event at the end, to illustrate the switch of winning and losing more clearly.

In future work, we will define the formal syntax and semantics of story graphs [6] and explore more of their potential usages, in particular, possible interactions for story graphs that support explanation exploration. We will also extend story graphs to a visual *programming* language for explanations. Moreover, we plan a

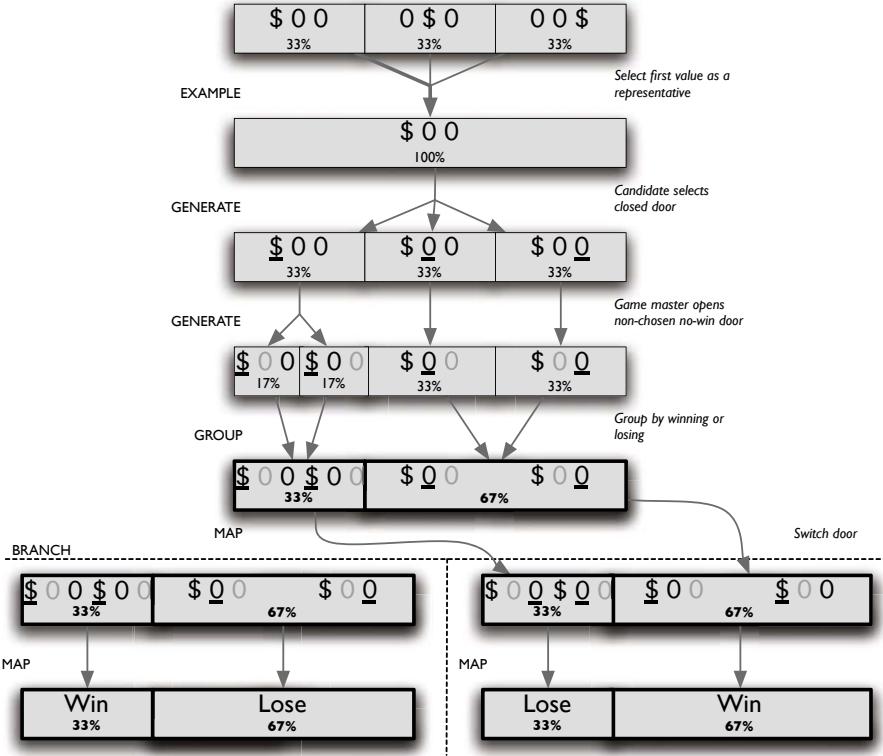


Fig. 5. Visual explanation of the Monty Hall problem

detailed comparison with Bayesian networks. In particular, we will devise methods for translating Bayesian networks into story graphs.

## 6 Related Work

In Section 2 we have already discussed work in the philosophy of science regarding theories of explanation. One important idea that has emanated from that work is the importance of causality for explanations [5,29,15,27,4,21,30]. Some notational support for expressing explanations and reasoning about causality has developed in the form of causal graphs [21,30]. Pearl provides a precise definition of causal graphs, and he demonstrates how they support the reasoning about intervention [21]. Our explanation language is based on a story-telling model that is a deliberate restriction of causal graphs to a mostly linear form. Since our language is embedded into Haskell it encourages the use of types to structure the domain of the explanandum. The existence of laws for the explanation operators (briefly pointed out in Section 4.3) allows the systematic generation of alternative explanations and thus provides support for explanation exploration.

It is not clear whether similar results hold in less constrained general graph models. Another difference between causal graphs and our explanation model is that intervention is represented in causal graphs by an operation (`do`) that transforms one causal graph into another, that is, there is no explicit representation of intervention in the graph representation itself. In contrast, we employ a `Branch` construct that allows the integration of different story lines into one explanation, which means that intervention can be represented explicitly.<sup>6</sup> This difference might be important when we consider exploration operations for explanations since the explicit representation supports the exploration of intervention, which could be a huge advantage given the importance that interventions in causal graphs have for explanations.

When we look for explanation support for specific domains, we find in the area of algorithm animation [17] many ideas and approaches to illustrate the working of algorithms dynamically through custom-made or (semi-)automatically generated animations [16]. The story-telling metaphor has been employed in that area as well [3], but only in the sense of adding textual explanations and not as a structural guide as in our case.

Related to algorithm animation is the more general idea of obtaining explanations of programs or program behavior, which, as discussed in Section 1.1, is not well supported by existing tools. Most debuggers operate on a very low level and often require much time and effort by the user to deliver meaningful information. The approach taken by the WHYLINE system [19] improves this situation by inverting the direction of debugging. This system allows users to ask “Why...?” and “Why didn’t...?” questions about particular program behaviors, and the system responds by pointing to parts of the code responsible for the outcomes in question. Even though this system improves the debugging process significantly, it can only point to places in a program, which limits its explanatory value. In the domain of spreadsheets we have extended this idea so that users can express expectations about outcomes of cells, and the system then generates change suggestions for cells in the spreadsheet that would produce the desired results [1]. From the point of view of causal explanations, the produced change suggestions play the role of counterfactuals.

In the context of probabilistic reasoning, although there are languages that support the *computation* with probabilities, such as IBAL [23] or PFP [7], there is, as far as we know, no language support for generating *explanations* of probabilistic computations.

Finally, the idea of elevating explainability as a design criterion for languages was first proposed in [8] where we have presented a visual language for expressing game strategies. A major guiding principle for the design of the visual notation was the *traceability* of game results, that is, how well the strategies could be explained by relating them to actual game traces. In this sense that visual language is an example of an explanation-oriented language.

---

<sup>6</sup> Pearl [21] briefly describes the concept of twin networks that is similar in this respect.

## 7 Conclusions and Future Work

Let us reconsider the boy/girl riddle and compare the program for solving it given in Section 4.1 with the explanation program presented in Section 4.2. Both programs can be used to obtain a solution to the riddle, but only the latter provides an explanation of *how and why* the solution is obtained.

An explanation is given by a linear sequence of values that describe the evolution from some start situation to the value describing the situation to be explained. Such an explanation trace is obtained by threading a value through a list of function applications. Each function is a causal link in the explanation and represents a part of a “story” that constitutes the explanation.

The visual notation for probability explanations combines the concepts of spatial partitions and story graphs and suggests the view of an explanation as a set of values that flow along edges and that show changes in their probabilities as growing or shrinking areas.

The idea of *explanation-oriented programming* is to shift the focus from producing values and results to producing explanations for how they are obtained. This shift changes the objective of programming and has implications for language design on multiple levels.

In future work we will investigate laws that allow semantics-preserving transformations of explanation to automatically generate a collection of related explanations that can be navigated by the user. We will also investigate the user interface aspect of explanation navigation and identify operations (and potential extensions of the representation, such as new kinds of views on probability distributions) that can help formally capture and support the exploration and navigation of explanations. We will also continue to explore opportunities for explanation-oriented languages in other domains.

## References

1. Abraham, R., Erwig, M.: GoalDebug: A Spreadsheet Debugger for End Users. In: 29th IEEE Int. Conf. on Software Engineering, pp. 251–260 (2007)
2. Achinstein, P.: The Nature of Explanation. Oxford University Press, New York (1983)
3. Blumenkrants, M., Starovisky, H., Shamir, A.: Narrative Algorithm Visualization. In: ACM Symp. on Software visualization, pp. 17–26 (2006)
4. Dowe, P.: Physical Causation. Cambridge University Press, Cambridge (2000)
5. Dummett, M.: Bringing About the Past. *Philosophical Review* 73, 338–359 (1964)
6. Erwig, M.: Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing* 9(5), 461–483 (1998)
7. Erwig, M., Kollmansberger, S.: Probabilistic Functional Programming in Haskell. *Journal of Functional Programming* 16(1), 21–34 (2006)
8. Erwig, M., Walkingshaw, E.: A Visual Language for Representing and Explaining Strategies in Game Theory. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 101–108 (2008)
9. Friedman, M.: Explanation and Scientific Understanding. *The Journal of Philosophy* 71(1), 5–19 (1974)

10. Giry, M.: A Categorical Approach to Probability Theory. In: Banaschewski, B. (ed.) *Categorical Aspects of Topology and Analysis*, pp. 68–85 (1981)
11. Hehner, E.C.R.: Probabilistic Predicative Programming. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 169–185. Springer, Heidelberg (2004)
12. Hempel, C.: *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*. Free Press, New York (1965)
13. Holland, J., Holyoak, K., Nisbett, R., Thagard, P.: *Induction: Processes of Inference, Learning and Discovery*. MIT Press, Cambridge (1986)
14. Hudak, P.: Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es), 196–196 (1996)
15. Humphreys, P.: *The Chances of Explanation*. Princeton University Press, Princeton (1989)
16. Karavirta, V., Korhonen, A., Malmi, L.: Taxonomy of Algorithm Animation Languages. In: *ACM Symp. on Software visualization*, pp. 77–85 (2006)
17. Kerren, J.T., Stasko, A.: Algorithm Animation – Introduction. In: Diehl, S. (ed.) *Dagstuhl Seminar 2001*. LNCS, vol. 2269, pp. 1–15. Springer, Heidelberg (2002)
18. Kitcher, P.: Explanatory Unification and the Causal Structure of the World. In: Kitcher, P., Salmon, W. (eds.) *Scientific Explanation*, pp. 410–505. University of Minnesota Press, Minneapolis (1989)
19. Ko, A.J., Myers, B.A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In: *IEEE Int. Conf. on Software Engineering*, pp. 301–310 (2008)
20. Morgan, C., McIver, A., Seidel, K.: Probabilistic Predicate Transformers. *ACM Transactions on Programming Languages and Systems* 18(3), 325–353 (1996)
21. Pearl, J.: *Causality: Models, Reasoning and Inference*. Cambridge University Press, Cambridge (2000)
22. Pennington, N., Hastie, R.: Reasoning in Explanation-Based Decision Making. *Cognition* 49, 123–163 (1993)
23. Ramsey, N., Pfeffer, A.: Stochastic Lambda Calculus and Monads of Probability Distributions. In: *29th Symp. on Principles of Programming Languages*, January 2002, pp. 154–165 (2002)
24. Ruben, D.: *Explaining Explanation*. Routledge, London (1990)
25. Salmon, W.: *Scientific Explanation and the Causal Structure of the World*. Princeton University Press, Princeton (1984)
26. Salmon, W.: *Four Decades of Scientific Explanation*. University of Minnesota Press, Minneapolis (1989)
27. Salmon, W.: Causality without Counterfactuals. *Philosophy of Science* 61, 297–312 (1994)
28. Scaffidi, C., Shaw, M., Myers, B.: Estimating the Numbers of End Users and End User Programmers. In: *IEEE Symp. on Visual Languages and Human-Centric Computing*, pp. 207–214 (2005)
29. von Wright, G.: *Explanation and Understanding*. Cornell University Press, Ithaca (1971)
30. Woodward, J.: *Making Things Happen*. Oxford University Press, New York (2003)
31. Wright, S.: Correlation and Causation. *Journal of Agricultural Research* 20, 557–585 (1921)

# Embedded Probabilistic Programming\*

Oleg Kiselyov<sup>1</sup> and Chung-chieh Shan<sup>2</sup>

<sup>1</sup> FNMOC

[oleg@pobox.com](mailto:oleg@pobox.com)

<sup>2</sup> Rutgers University

[ccshan@rutgers.edu](mailto:ccshan@rutgers.edu)

**Abstract.** Two general techniques for implementing a domain-specific language (DSL) with less overhead are the *finally-tagless* embedding of object programs and the *direct-style* representation of side effects. We use these techniques to build a DSL for *probabilistic programming*, for expressing countable probabilistic models and performing exact inference and importance sampling on them. Our language is embedded as an ordinary OCaml library and represents probability distributions as ordinary OCaml programs. We use delimited continuations to reify probabilistic programs as lazy search trees, which inference algorithms may traverse without imposing any interpretive overhead on deterministic parts of a model. We thus take advantage of the existing OCaml implementation to achieve competitive performance and ease of use. Inference algorithms can easily be embedded in probabilistic programs themselves.

## 1 Introduction

In many fields of science and engineering, including artificial intelligence (AI), cryptography, economics, and biology, *probabilistic* (or *stochastic*) models are a popular way to account for and deal with uncertainty. The uncertainty may be inherent in the domain being modeled, induced by our imperfect knowledge and observation of the domain, or introduced to simplify a problem that is in principle deterministic. For example, all three kinds of uncertainty enter models in our area of interest, natural-language dialogue: many words and phrases are ambiguous, so interlocutors do not know exactly each other's communicative intentions, and it may be expedient to lump possible intentions into a finite number of equivalence classes.

Whichever kind of uncertainty they are used to model, probabilities are real numbers that can be regarded as weights on nondeterministic choices. A canonical example of a probabilistic model is that of a lawn whose grass may be wet because it rained, because the sprinkler was on, or for some other reason. Mathematically speaking, we have three Boolean variables `rain`, `sprinkler`, and `grass_is_wet`, and a probability distribution defined by

---

\* Thanks to Olivier Danvy, Avi Pfeffer, and the anonymous reviewers. Thanks also to Aarhus University for hosting the second author in the fall of 2008.

$$\begin{aligned}\Pr(\text{rain}) &= 0.3, \\ \Pr(\text{sprinkler}) &= 0.5, \\ \Pr(\text{grass\_is\_wet} \mid \text{rain} = r \wedge \text{sprinkler} = s) &= 1 - n(0.1, r) \cdot n(0.2, s) \cdot 0.9.\end{aligned}$$

We use the ‘noisy not’ function  $n$  above to express possibilities such as a light rain that dries immediately and a sprinkler with low water pressure:

$$n(p, \text{true}) = p, \quad n(p, \text{false}) = 1.$$

By the definition of conditional probability ( $\Pr(A|B) = \Pr(A \wedge B) / \Pr(B)$ ), the product of these distributions is the *joint* distribution of all three variables. Unlike this simple example, many probabilistic models in production use contain deterministic parts (laws of motion, for example) alongside random parts [14, 32].

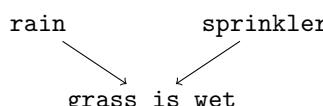
Given a probability distribution such as the lawn model just defined, we want to perform *inference*, which means to compute the probability distribution (called a *marginal* distribution) over some random variables in the model that we care about. For example, we might want to compute  $\Pr(\text{grass\_is\_wet})$ . We might also want to *sample* from the distribution, which means to generate a possible assignment of values to random variables such that each assignment’s probability of being generated is equal to its probability in the model. Sampling from a distribution is a popular way to conduct approximate inference: we can approximate  $\Pr(\text{grass\_is\_wet})$  by sampling the random variable `grass_is_wet` many times and calculating the proportion of times that it comes out true.

We often want to perform inference or sampling not on our probabilistic model as is but conditional on some observed evidence. For example, having observed that the grass is wet, we may want to compute  $\Pr(\text{rain} \mid \text{grass\_is\_wet})$ , either exactly or approximately. Such inference lets us *explain* observed evidence and *learn* from them, especially if, as Bayesians, we treat probabilities themselves (such as  $\Pr(\text{rain})$ ) as random variables [37].

## 1.1 Languages for Probabilistic Models

Like many AI researchers, we aim to represent probability distributions so that humans can develop and understand them easily and computers can perform inference and sampling on them efficiently. In other words, we aim to build a domain-specific language for probability distributions. Many such languages have been described in the literature [18, 36]. They are instances of declarative programming, in that they separate the description of models from the computations on them.

A fundamental and popular way to represent a probability distribution is as a *graphical model* (specifically a *Bayes net*), a directed acyclic graph whose nodes represent random variables and edges indicate (typically causal) dependencies.



Each node  $X$  is annotated with the conditional probability distribution of its value given the values of its parents  $\vec{Y}$ . The notion of dependencies is such that  $\Pr(X|\vec{Z}) = \Pr(X|\vec{Y})$  if  $Z$  is a superset of  $Y$ ; informally, each node depends directly on its parents only.

Humans find graphical models intuitive because they describe sampling procedures: to pick a value for a node  $X$ , first pick values for its parents  $\vec{Y}$ , then refer to the conditional probability distribution  $\Pr(X|\vec{Y})$  for the node  $X$ . The dependency structure that graphical models express also gives rise to efficient inference algorithms. However, as graphical models scale up to real problems involving thousands or more of random variables and dependencies, they become unwieldy in the raw.

It helps to *embed* [25, 31] a language of probability distributions in a host language such as MATLAB [35] or Haskell [8]. The embedded language is more commonly called a *toolkit* for probabilistic reasoning, because it is structured as a library of data types of graphical models and random variables along with inference procedures that operate on them. Similarly, distributions can be represented and reasoned about in a logic using formulas that talk about probabilities [22]. In the embedded setting, graphical models can be constructed using programs written in the host language, so repeated patterns (such as random variables that recur for each time step) can be factored out and represented compactly. However, a toolkit defines its own data types; for example, random integers are distinct from regular integers and cannot be added using the addition operation of the host language. This linguistic mismatch typically degrades the efficiency, concision, and maintainability of deterministic parts of a model. In particular, it is difficult to include the toolkit itself in a model, as is useful for agents to reason about each other.

The notational overhead caused by linguistic mismatch can be eliminated by building a standalone language for probability distributions. Such a language typically adds constructs for probabilistic choice and evidence observation to a general-purpose language, be it of a functional [20, 30, 38, 40, 42], logical [2, 7, 28, 34, 41, 43], or relational [16, 17, 32, 45, 46] flavor. For example, the conditional distribution  $\Pr(\text{rain} \mid \text{grass\_is\_wet})$  can be expressed as follows for inference in IBAL [38].

```

rain      = dist [ 0.3 : true, 0.7 : false ]
sprinkler = dist [ 0.5 : true, 0.5 : false ]
grass_is_wet = dist [ 0.9 : true, 0.1 : false ] & rain
           | dist [ 0.8 : true, 0.2 : false ] & sprinkler
           | dist [ 0.1 : true, 0.9 : false ]
observe grass_is_wet in rain

```

The construct `dist` expresses probabilistic choice, and the construct `observe` expresses evidence observation (here, that `grass_is_wet` is true). Because the entire language is probabilistic, the same operation can be used to, for example, add random integers as to add regular integers. It is easy to express deterministic parts of a model, simply by not using probabilistic constructs in part of the

code. These notational conveniences make programs in a standalone probabilistic language easier for domain experts to develop and understand.

The main drawback of a standalone probabilistic language is that it cannot rely on the infrastructure of an existing host language. To start with, the vast majority of standalone probabilistic languages are implemented as interpreters rather than compilers (with the notable exceptions of AutoBayes [13] and HBC [5]). In contrast, an embedded probabilistic language can piggyback on its host language’s compilers to remove some interpretive overhead. We are also not aware of any probabilistic language implemented in itself, so again the language cannot be included in a model for multi-agent reasoning. In general, a standalone language takes more effort to implement than an embedded one, if only due to the need to implement ‘boring bits’ such as string processing, I/O, timing facilities, and database interfaces. A foreign function interface to a mainstream language can help fill the void of libraries, but glue code imposes its own overhead.

## 1.2 Our Approach: A Very Shallow Embedding

We combine the advantages of embedded and standalone probabilistic languages by embedding a language of probabilistic distributions into a general-purpose language in a *very shallow* way. That is, probabilistic functions are embedded as host-language functions, calls to them are embedded as host function calls, and random integers can be added using the addition operation of the host language. This approach lets us take advantage of the existing infrastructure of the host language while also reducing notational and interpretive overhead.

For example, the conditional distribution  $\Pr(\text{rain} \mid \text{grass\_is\_wet})$  can be expressed as the following host-language program. We use the host language’s abstraction facility to factor out the notion of a coin **flip** that yields **true** with probability  $p$ .

```
let flip p = dist [(p, true); (1.-p, false)]
let grass_model () =
  let rain = flip 0.3 and sprinkler = flip 0.5 in
  let grass_is_wet = flip 0.9 && rain
    || flip 0.8 && sprinkler
    || flip 0.1 in
  if grass_is_wet then rain else fail ()
```

Our host language is OCaml: the code above is just an OCaml program that defines two ordinary functions **flip** (of type `float -> bool`) and **grass\_model** (of type `unit -> bool`). The definition of **grass\_model** binds the ordinary Boolean variables **rain**, **sprinkler**, and **grass\_is\_wet** using the functions **dist** (for making a probabilistic choice) and **fail** (for observing impossible evidence). These functions are provided by a library we built without modifying or duplicating the implementation of OCaml itself.<sup>1</sup> Accordingly, we can express probabilistic models using OCaml’s built-in operations (such as `&&`), control constructs

<sup>1</sup> In contrast, IBAL happens to be written in OCaml, but it cannot perform inference on OCaml programs such as this one or itself.

(such as `if`), data structures, module language, and wealth of libraries, including our implementations of probabilistic inference, which are useful for multi-agent reasoning. We can use OCaml’s type system to discover mistakes earlier, and OCaml’s bytecode compiler to perform inference faster.

On one hand, because our approach represents probability distributions as ordinary programs, it is easier for programmers to learn, especially those already familiar with the host language. On the other hand, there is some linguistic mismatch between evaluating a probability distribution and evaluating a call-by-value program, so some thunks are necessary. These occurrences of ‘`fun () ->`’ in the code are only slightly bothersome, and we could easily write a Camlp4 preprocessor to eliminate them. They would be unnecessary if we were to express distributions in Haskell rather than OCaml, as monadic computations.

To continue the example above briefly, we can use two other library functions `reify` and `normalize` to evaluate the model at OCaml’s top-level prompt ‘#’:

```
# normalize (reify None grass_model);;
reify: done; 14 accepted 9 rejected 0 left
bool pV = PV [(0.468471442720369724, V true);
               (0.53152855727963022, V false)]
```

Whereas `dist` takes a list of weighted possibilities as input and makes a random choice, `reify` takes a probabilistic program (a thunk) as input and computes its list of weighted possibilities (return values). The `normalize` function then rescales the weights to sum up to 1. The result above says that the conditional probability that it rained, given that the grass is wet, is 47%. If we do not like this result, we can define and evaluate a different model interactively. Alternatively, we can compile the model by placing the code above for `grass_model` in a file along with the following driver.

```
let PV choices = normalize (reify None grass_model) in
let [(p, _)] = List.filter (function (_,V x) -> x) choices in
Printf.printf "The probability it rained is %g" p
```

### 1.3 Contributions

This paper describes our design of a practical DSL for probabilistic programming. We achieve a very shallow embedding of the language, by a novel combination of two existing techniques for eliminating notational and interpretive overhead in embedded languages.

1. The *direct-style* representation of side effects [11, 12] using *delimited control operators* [4, 10] lets us reify a probabilistic program as a weighted search tree without the run-time overhead of stepping through deterministic code.
2. The *finally-tagless* embedding of object programs [1] using *combinator functions* [33, 47] allows multiple inference algorithms to operate on the same model without the run-time overhead of tagging values with their types.

We apply these general techniques to probabilistic programming and demonstrate how they make embedded DSL programs faster to run and easier to write. We then show how to implement interpreters for our language that perform exact and approximate inference efficiently. More specifically:

1. We implement *variable elimination* [6], an existing algorithm for exact inference.
2. We develop *importance sampling* [15, 44] with look-ahead, a new, general algorithm for approximate inference.
3. We support *lazy evaluation* in probabilistic programs.
4. Users of our library are free to implement their own inference algorithms.

Our implementation, with many examples, benchmarks, and tests, is available online at <http://okmij.org/ftp/kakuritu/README.dr>. It is only a few hundred lines long, yet its speed and accuracy on available benchmarks [27] are competitive with other, state-of-the-art implementations of probabilistic languages such as IBAL [39]. Our work thus establishes that these general techniques for very shallow embeddings are effective in the case of probabilistic programming and compatible with inference algorithms as advanced as importance sampling. The latter compatibility is surprising because a very shallow embedding prevents inference procedures from inspecting the source code of probabilistic programs.

Having reviewed related work above, we structure the rest of this paper as follows. Section 2 develops our very shallow embedding of a probabilistic DSL in OCaml in several steps. We begin with a finally tagless embedding, which is statically typed without tagging overhead, but whose straightforward implementation is verbose to use and does not run deterministic code at full speed. Following Filinski’s [11] representation of monads, we address these shortcomings by converting to CPS and then to direct style.

The remainder of the paper shows how to implement efficient inference algorithms by reifying a stochastic computation as a lazy search tree. Section 3 describes the use of reification for exact inference and for stepping through a stochastic model. Section 4 describes our algorithm for importance sampling with look-ahead. Section 5 introduces lazy evaluation in the embedded program. Finally, we describe our performance testing and conclude.

## 2 Embedding Stochastic Programs

In this section, we develop our very shallow embedding of a probabilistic DSL in OCaml in three steps. First, in §2.1, we define the probabilistic DSL in *tagless final* form: as a signature that lists the operations and their types. Second, in §2.2, we describe a basic implementation of the signature, using the probabilistic monad realized as a lazy search tree. The third step is to eliminate the notational overhead of writing deterministic code and the interpretive overhead of running it. In §2.3, we show how to implement the same signature in continuation-passing style (CPS). We then introduce the delimited control operators *shift* and *reset* in §2.4, so as to present the ultimate embedding of our language in §2.5.

## 2.1 Tagless Final Probabilistic Language

To begin, we define the language, that is, introduce its operations and their types. We define the syntax of the language as the following OCaml signature `ProbSig`, so that each implementation of the language is a module that matches the signature [1]. The implementations below make the language call-by-value.

```

type prob = float

module type ProbSig = sig
  type 'a pm
  type ('a,'b) arr

  val b      : bool -> bool pm
  val dist  : (prob * 'a) list -> 'a pm
  val neg   : bool pm -> bool pm
  val con   : bool pm -> bool pm -> bool pm
  val dis   : bool pm -> bool pm -> bool pm
  val if_   : bool pm -> (unit -> 'a pm) -> (unit -> 'a pm) -> 'a pm
  val lam  : ('a pm -> 'b pm) -> ('a,'b) arr pm
  val app  : ('a,'b) arr pm -> ('a pm -> 'b pm)
end

```

Our ultimate embedding in §2.5 is very shallow in the sense that it identifies our probabilistic DSL with the host language OCaml. For exposition, however, we start with Erwig and Kollmansberger’s toolkit approach [8] and keep the DSL distinct from OCaml.

For simplicity, the signature `ProbSig` includes only the base type of booleans and the operations necessary for the lawn example. Following the tagless-final approach [1], a boolean expression in the DSL, which denotes a distribution over booleans, is represented as an OCaml expression of the type `bool pm`. Here `pm` is an abstract type constructor, whose name is short for ‘probability monad’ [8, 19, 42]. Different implementations of `ProbSig` define `pm` in their own ways, giving rise to different interpreters of the same DSL. As an example, the ‘true’ literal in the DSL is represented in OCaml as `b true`, of the abstract type `bool pm`. The expression denotes the distribution over ‘true’ with probability 1.

Besides booleans, our language has functions. A function expression in the DSL, say of type `bool -> bool`, is represented as an OCaml expression of the type `(bool, bool) arr pm`. Here `arr` is a binary type constructor, again abstract. We use higher-order abstract syntax to represent binding. For example, we represent the curried binary function on booleans  $\lambda x. \lambda y. x \wedge \neg y$  in the DSL as the OCaml expression `lam (fun x -> lam (fun y -> con x (neg y)))`, whose inferred type is `(bool, (bool, bool) arr) arr pm`. To apply this function to the literal ‘false’, we write `app (lam (fun x -> ...)) (b false)`.

This DSL is probabilistic only due to the presence of the `dist` construct, which takes a list of values and their probabilities. The `dist` expression then denotes a stochastic computation with the given discrete distribution.

A probabilistic program is a functor that takes a `ProbSig` module as argument. For example, we can write the lawn example as follows:

```

module Lawn(S: ProbSig) = struct
  open S

  let flip p = dist [(p, true); (1.-p, false)]

  let let_ e f = app (lam f) e

  let grass_model () =
    let_ (flip 0.3) (fun rain ->
      let_ (flip 0.5) (fun sprinkler ->
        let_ (dis (con (flip 0.9) rain)
          (dis (con (flip 0.8) sprinkler)
            (flip 0.1))) (fun grass_is_wet ->
              if_ grass_is_wet (fun () -> rain) (fun () -> dist []))))
    end

```

For convenience, we introduce the `let_` ‘operator’ above.

This example shows a significant notational overhead. For example, boolean stochastic variables have the type `bool pm` rather than just `bool`, so we cannot pass a random boolean to OCaml functions accepting ordinary booleans, and we need to use `con` rather than the built-in conjunction operator `&&`. Also, we cannot use OCaml’s `let` and have to emulate it by `let_`. The notational overhead is as severe as if our embedded DSL were just a toolkit. By the end of the section, we shall eliminate most of the overhead except a few stray `fun () ->`.

Despite its verbosity, our embedding already takes advantage of the type system of the host language: our DSL is typed, and we do not need our own type checker—the OCaml type checker infers the types of stochastic expressions just as it does for all other expressions. For example, the expression `app (b true) (b false)` is flagged as a type error because `bool` is not an arrow type.

## 2.2 Search-Tree Monad

The first implementation of our language uses the probability monad that represents a stochastic computation as a lazy search tree. That is, our first `ProbSig` module instantiates `pm` by the type constructor `pV` defined below.

```

type 'a vc = V of 'a | C of (unit -> 'a pV)
and 'a pV = (prob * 'a vc) list

```

Each node in a tree is a weighted list of branches. The empty list denotes failure, and a singleton list `[(p, V v)]` denotes a deterministic successful outcome `v` with the probability mass `p`. A branch of the form `V v` is a leaf node that describes a possible successful outcome, whereas a branch of the form `C thunk` is not yet explored.

The intended meaning of a search tree of type `'a pV` is a discrete probability distribution over values of type `'a`. Of course, several distinct trees may represent the same distribution; for example, the following trees all represent the same distribution.

```

[(0.6, V true); (0.4, V false)]
[(0.4, V true); (0.4, V false); (0.2, V true)]
[(0.4, V true); (1.0, C (fun () ->
                           [(0.4, V false); (0.2, V true)]))]
[(0.4, V true); (0.8, C (fun () ->
                           [(0.5, V false); (0.25, V true)]))]
[(0.4, V true); (0.8, C (fun () ->
                           [(0.5, V false); (0.25, V true);
                            (0.25, C (fun () -> []))]))]

```

The first tree is just a probability table; it is the smallest, consisting only of a root node. We call such trees flat or fully explored, because they represent a distribution that has been evaluated or inferred completely. The last three trees contain unexplored branches, signified by the `C` variant. In particular, the last example includes a failure node `[]`, which becomes manifest when the branch `C (fun () -> [])` is explored.

We consider trees that represent the same distribution equivalent. Constructively, we introduce the following normalization, or, *exploration* procedure for search trees. The procedure traverses the tree depth-first, forces thunks of branches not yet explored to reveal their content, and accumulates all found values along with their weights in the probability table. While exploring a branch `(p, C t)`, we scale the probabilities of the found values by the factor `p`. The resulting probability table is returned as a flat search tree. The procedure uses a map `PMap` from values to probabilities to collate the values encountered during exploration. The procedure below is more general in that it explores the tree up to the depth specified by the first argument; giving the value `None` as the first argument requests complete normalization. Applying `explore None` to all these trees returns the same result, so they are all equivalent.

```

let explore (maxdepth : int option) (choices : 'a pV) : 'a pV =
  let rec loop p depth down choices ((ans,susp) as answers) =
    match choices with
    | [] -> answers
    | (pt, V v)::rest ->
        loop p depth down rest
        (PMap.insert_with (+.) v (pt *. p) ans, susp)
    | (pt, C t)::rest when down ->
        let down' =
          match maxdepth with Some x -> depth < x | None -> true
        in loop p depth down rest
          (loop (pt *. p) (depth+1) down' (t ()) answers)
    | (pt, c)::rest ->
        loop p depth down rest (ans, (pt *. p,c)::susp) in
  let (ans,susp) = loop 1.0 0 true choices (PMap.empty,[])
  in PMap.foldi (fun v p a -> (p, V v)::a) ans susp

```

The last two lines prepend the collated values to the list of unexplored branches.

The type constructor `pV`, equipped with the operations `pv_unit` and `pv_bind` below, forms a monad [48]:

```
let pv_unit (x : 'a) : 'a pV = [(1.0, V x)]
let rec pv_bind (m : 'a pV) (f : 'a -> 'b pV) : 'b pV =
  List.map (function
    | (p, V x) -> (p, C (fun () -> f x))
    | (p, C t) -> (p, C (fun () -> pv_bind (t ()) f)))
  m
```

Monad laws are satisfied only modulo tree equivalence. For example, for values `x` of type `'a` and `f` of type `'a -> 'b pV`, the tree resulting from OCaml evaluating `pv_bind (pv_unit x) f` is equivalent but not identical to the result of `f x`. More precisely, if the trees are finite, then exploring them fully (by applying `explore None`) yields identical results. For infinite trees, which too can be represented by the type `'a pV`, characterizing equivalence is trickier [49].

We use this search-tree monad to define the `SearchTree` module below, our first implementation of the `ProbSig` signature.

```
module SearchTree = struct
  type 'a pm = 'a pV
  type ('a, 'b) arr = 'a -> 'b pV

  let b = pv_unit
  let dist ch = List.map (fun (p, v) -> (p, V v)) ch
  let neg e = pv_bind e (fun x -> pv_unit (not x))
  let con e1 e2 = pv_bind e1 (fun v1 ->
    if v1 then e2 else (pv_unit false))
  let dis e1 e2 = pv_bind e1 (fun v1 ->
    if v1 then (pv_unit true) else e2)
  let if_ et e1 e2 = pv_bind et (fun t ->
    if t then e1 () else e2 ())
  let lam e = pv_unit (fun x -> e (pv_unit x))
  let app e1 e2 = pv_bind e1 (pv_bind e2)
end
```

This implementation is the standard monadic embedding of a call-by-value language. For example, `neg` is the straightforward ‘lifting’ of boolean negation so to accept and produce values of the monadic type `bool pV`. The implementation of `con` shows short-cut evaluation: if the first argument evaluates to false, we produce the result ‘false’ without evaluating the second argument. The implementations of `dis` and `if_` follow a similar pattern. The only interesting part is the implementation of `dist`, which merely converts its argument (specifying the distribution) to a flat search tree. We can use this implementation to run the `lawn` example, by instantiating the `Lawn` functor with the module `SearchTree`:

```
module SST = Lawn(SearchTree)
```

The resulting module `SST` contains a field `grass_model`, of the inferred type `bool SearchTree.pv`, or `bool pV`:

```
let sst1 = SST.grass_model ()
↪ val sst1 : bool SearchTree.pv = [(1., C <fun>)]
```

That tree is obviously unexplored. Exploring it to depths 3, 9, and 11 gives

```
let sste3 = explore (Some 3) sst1
↪ [(0.15, C <fun>); (0.15, C <fun>);
  (0.35, C <fun>); (0.35, C <fun>)]

let sste9 = explore (Some 9) sst1
↪ [(0.27, V true); (0.012, C <fun>);
  (0.0003, C <fun>); (0.0027, C <fun>);
  (0.0012, C <fun>); (0.0108, C <fun>); ...]

let sste11 = explore (Some 11) sst1
↪ [(0.2838, V true); (0.322, V false)]
```

Only when we explore the tree to depth 11 do we finally obtain the exact distribution denoted by the lawn model: an unnormalized distribution of `rain`. After normalization, it becomes what we have seen in §1.2.

The depth of this search tree reveals the run-time overhead imposed by this implementation. Because booleans and functions in our DSL do not share the types of ordinary booleans and functions in OCaml, we cannot invoke a stochastic function as an ordinary function, and we cannot pass stochastic booleans to OCaml boolean functions. We cannot even use OCaml’s boolean constants `true` and `false` as they are; rather, we have to lift them to `b true` and `b false`, search trees of type `bool pV`. Computing with such lifted values is not only more verbose but also less efficient: in the `SearchTree` module, all operations on trees involve `pv_bind`, which always increases the depth of the tree by one.

This example thus illustrates that the notational overhead in our representation of stochastic programs corresponds to an interpretive overhead in the search-tree implementation. When a sequence of deterministic operations is applied to a stochastic expression, as in `neg (neg rain)`, each step incurs its own overhead of `pv_bind` and `pv_unit` traversing and constructing lists and variant data structures. We next obviate the pattern matching performed by `pv_bind`.

### 2.3 Embedding in Continuation-Passing Style

We eliminate the run-time and notational overhead in this and the next two sections.

We still use search trees to represent stochastic expressions, but we no longer implement the stochastic language using the operations `pv_unit` and `pv_bind` in the search-tree monad. We use the continuation monad instead: we represent a stochastic expression of type `'a` not as a tree of type `'a pV` but in CPS, as a function from the continuation type `'a -> bool pV` to the *answer type*

`bool pV`. This move is same as Hughes's [26] and Hinze's [24] move from term implementations to context-passing implementations to reduce pattern-matching and improve efficiency. For this section, it is enough to fix the answer type at `bool pV` rather than make it '`w pV` for some generic '`w`.

Our new implementation of `ProbSig` is as follows. Most of this code is trivial; for example, the implementation of `app` is the standard CPS transform of call-by-value application.

```
module CPS = struct
  type 'a pm = ('a -> bool pV) -> bool pV
  type ('a, 'b) arr = 'a -> ('b -> bool pV) -> bool pV

  let b x = fun k -> k x
  let dist ch = fun k ->
    List.map (function (p, v) -> (p, C (fun () -> k v))) ch
  let neg e = fun k -> e (fun v -> k (not v))
  let con e1 e2 = fun k -> e1 (fun v1 ->
    if v1 then e2 k else b false k)
  let dis e1 e2 = fun k -> e1 (fun v1 ->
    if v1 then (b true k) else e2 k)
  let if_ et e1 e2 = fun k -> et (fun t ->
    if t then e1 () k else e2 () k)
  let lam e = fun k -> k (fun x -> e (fun k -> k x))
  let app e1 e2 = fun k -> e1 (fun f -> e2 (fun x -> f x k))

  let reify0 m = m pv_unit
end
```

The most interesting case is again `dist`: the expression `dist ch` (where `ch` is a list of choices) accepts a continuation `k` as usual but does not return the result of invoking `k`. Rather, `dist ch` expresses nondeterminism by immediately returning the search tree resulting from feeding each choice in `ch` to `k`. That is, `dist ch k` builds a collection of branches, each of which, upon exploration, returns the result of the rest of the computation using a particular choice in `ch`. This definition of `dist` is the same as applying `pv_bind` to the previous definition `SearchTree.dist`.

We can run our lawn example using this new implementation of `ProbSig`. We pass the module `CPS` to the `Lawn` functor.

```
module SCP = Lawn(CPS)
```

The resulting module `SCP` contains a field `grass_model` with the type `unit -> (bool -> bool pV) -> bool pV`. Forcing this thunk gives not a tree anymore but a higher-order function accepting continuations of type `bool -> bool pV`. In other words, `grass_model ()` is a CPS computation. To convert this result to a search tree so that we can compute the distribution, we have to *reify* it by feeding it the initial continuation `pv_unit`. We do so by applying `reify0`.

```
let scp1 = CPS.reify0 (SCP.grass_model ())
  ↵ [(0.3, C <fun>); (0.7, C <fun>)]
```

The result `scp1` is indeed a search tree. It still has a few unexplored branches; exploring the resulting tree to the depths 1 and 4 gives

```
let scpe1 = explore (Some 1) scp1
→ [(0.135, C <fun>); (0.015, C <fun>); (0.135, C <fun>);
  (0.015, C <fun>); (0.315, C <fun>); (0.035, C <fun>);
  (0.315, C <fun>); (0.035, C <fun>)]

let scpe4 = explore (Some 4) scpe1
→ [(0.2838, V true); (0.322, V false)]
```

Exploring this tree to depth 4 already flattens it, so it is not nearly as deep as the tree `sst1` computed in §2.2. This comparison hints that we have removed some overhead in the `SearchTree` implementation of the language.

We can see why the new implementation of `ProbSig` incurs less overhead by considering the example `neg (neg rain)` above. Although we still cannot negate the random boolean `rain` directly using OCaml’s built-in `not`, at least the definition of `neg` no longer uses `pv_unit` and `pv_bind`, so the inner call to `neg` no longer builds a tree only to be traversed and discarded right away by the outer call. Rather, each boolean is passed to the rest of the computation as is, without building any tree until `reify0` constructs a leaf node or `dist` constructs a choice node. This CPS implementation is thus more efficient than the search-tree implementation, yet equivalent according to the monad laws.

Some overhead remains. Stochastic boolean values remain distinct from ordinary boolean values, so we still have to write `b true` to use even a boolean constant in the language, and applying `b` at run time is not free. The per-application overhead, albeit reduced, is still present due to the need to explicitly invoke continuations. We next eliminate all this overhead, *mechanically*.

## 2.4 Delimited Control

To eliminate the notational and run-time overhead of the embedding of our stochastic DSL, we will use delimited control operators `shift` and `reset`. This section briefly introduces these operators.

The implementation of `CPS.dist` in §2.3 pointed out that a stochastic expression may be regarded as one that can return multiple times, like a `fork` expression in C. If `fork` were available in OCaml, we would use it to implement `dist`. Ordinary OCaml functions, which execute deterministically in a single ‘thread’, could then be used as they are within a stochastic computation.

The library of delimited continuations for OCaml [29] does provide the analogue of `fork`. To be more precise, the library provides two functions `reset` and `shift` [3, 4].<sup>2</sup> The function `reset`, of type `(unit -> 'a) -> 'a`, is analogous to `try` in OCaml. It takes a thunk, evaluates it, and returns its value—unless a

---

<sup>2</sup> The library actually implements so-called multi-prompt delimited continuations, similar to `cuputo` [21]. We embed stochastic computations using only a single prompt, which we store in a global variable and elide for clarity. The accompanying source file `probM.ml` shows the actual implementation.

control effect occurs during the evaluation. Thus, `reset (fun () -> 41 + 2)` returns 43. Control effects are raised by the function `shift`, which generalizes raising an exception. In particular, `reset (fun () -> 41 + shift (fun k -> 2))` returns the value 2: one may regard `shift (fun k -> 2)` to throw the integer 2 as an exception, caught by the closest dynamically enclosing `reset`.

In the example `reset (fun () -> 41 + shift (fun k -> 2))`, `shift` is applied in the context of the addition to 41 enclosed in `reset`. That context can be represented as a function `fun x -> reset (fun () -> 41 + x)`. Informally, the application of `shift` is replaced by `x` bound outside of `reset`. Therefore, `reset (fun () -> 41 + shift (fun k -> 2))` evaluates as

```
reset (fun () -> (fun k -> 2)
      (fun x -> reset (fun () -> 41 + x)))
```

and the result is 2, as we saw. The following examples proceed likewise.

```
reset (fun () -> 41 + shift (fun k -> k 2))
↪ reset (fun () -> (fun k -> k 2)
          (fun x -> reset (fun () -> 41 + x)))
↪ reset (fun () -> reset (fun () -> 41 + 2))
↪ 43
```

and

```
reset (fun () -> 41 + shift (fun k -> k (k 2)))
↪ reset (fun () -> (fun k -> k (k 2))
          (fun x -> reset (fun () -> 41 + x)))
↪ reset (fun () -> (fun x -> reset (fun () -> 41 + x))
          (reset (fun () -> 41 + 2)))
↪ 84
```

The last example shows that the context of `shift`, captured as a function, can be applied more than once. In other words, the `shift` expression can return more than once. In this regard, `shift` behaves like `fork`, splitting the computation into two threads and returning different values in them. That is exactly the behavior we need to implement `dist` with no overhead on deterministic code.

## 2.5 Direct Embedding of the Probabilistic DSL

We can finally implement our embedded DSL without overhead on deterministic code. Following Filinski [11], we do so by mechanically converting the CPS implementation to direct style.

```
module Direct = struct
  type 'a pm = 'a
  type ('a,'b) arr = 'a -> 'b

  let b x = x
  let dist ch = shift (fun k ->
```

```

List.map (function (p,v) -> (p, C (fun () -> k v))) ch)
let neg e = not e
let con e1 e2 = e1 && e2
let dis e1 e2 = e1 || e2
let if_ et e1 e2 = if et then e1 () else e2 ()
let lam e = e
let app e1 e2 = e1 e2

let reify0 m = reset (fun () -> pv_unit (m ()))
end

```

The transformation is straightforward, especially for expressions that use their continuations normally. The transformation makes the continuation argument `k` in the CPS implementation implicit, and so `neg` becomes just `not`. For the same reason, `app`, `lam`, and `b` become the identity function. The only interesting cases are `dist` and `reify0`. As explained in §2.3, the definition of `CPS.dist` is special: `CPS.dist ch` does not return the result of applying a continuation, but rather produces a search tree directly. The implementation of `dist` above expresses exactly the same operation in direct style, but the continuation argument `k` is implicit, so we access it using the control operator `shift`. Similarly, whereas `CPS.reify0` supplies the initial continuation `pv_unit` as an explicit argument, the implementation of `reify0` above uses `reset` to supply it implicitly.

We can run the lawn example as before, passing the `Direct` implementation to the functor `Lawn`. We obtain the same result as in §2.3:

```

module SDI = Lawn(Direct)
let sdi1 = Direct.reify0 (SDI.grass_model)
  ↵ [(0.3, C <fun>); (0.7, C <fun>)]

let sdie4 = explore (Some 4) sdi1
  ↵ [(0.2838, V true); (0.322, V false)]

```

The type of a stochastic computation that returns results of type `'a` is `'a pm`, which in the `Direct` implementation is just `'a`. Thus, a random boolean is just an ordinary boolean, which can be passed to ordinary boolean functions such as `not`. We can apply all ordinary OCaml functions, even compiled third-party libraries, to a stochastic value without lifting or changing them at all. Furthermore, since our direct embedding no longer distinguishes stochastic expressions from other (effectful) expressions in OCaml, we can dispense with the `ProbSig` structure. We can program probabilistic models directly in OCaml, using all of its facilities including `let` and `let rec` forms. The notational overhead has vanished (save a few stray thunks). This is how we wrote the lawn example in §1.2, directly in OCaml. The function `reify` mentioned there is the composition of `explore` and `reify0`. Applying `reify None` to the model performs exact inference, presenting the distribution of the model as a flat search tree.

One may remark that the thunk in the definition of `reify0` represents a certain overhead. Indeed, the result of an expression, even if it is completely deterministic, is represented as a search tree, and this boxing is overhead. However,

this overhead is incurred each time a complete probabilistic model is reified, not each time the model performs a deterministic operation. Unlike earlier implementations, we can apply deterministic functions to stochastic values with no overhead. We pay the price of embedding stochastic computations only when calling the two stochastic operators `dist` and `reify0`. The deterministic parts of a stochastic program run at full speed, as if no randomness were present and no inference were in progress. We have achieved our goal to eliminate run-time *and* notational overhead for deterministic parts of probabilistic models.

### 3 Accelerating Inference Using Reification and Reflection

We have seen that exact inference can be performed on a stochastic expression by reifying it into a lazy search tree (using `reify0`) then fully exploring the tree (using `explore None`). In this section, we show that `reify0` and `explore` are useful not only for producing the final inference result; these functions can help achieve the result faster. In other words, these functions let users of our library write optimized inference procedures. In particular, we implement the optimized exact-inference procedure known as *variable elimination* [6].

Throughout the rest of this paper, we use the very shallow embedding described in §2.5. We first introduce a function that generalizes `dist` there:

```
let reflect (choices : 'a pV) : 'a =
  let rec make_choices k pv =
    List.map (function
      | (p, V x) -> (p, fun () -> k x)
      | (p, C x) -> (p, fun () -> make_choices k (x ()))
    )
    pv
  in shift (fun k -> make_choices k choices)
```

Whereas `dist` takes as argument a weighted list of possibilities, `reflect` receives weighted possibilities in the form of a search tree. The only difference between a list and a tree is that a tree may not be flat and may even contain unexplored branches—hence the additional case alternative  $(p, C x) \rightarrow \dots$  above. Both `dist` and `reflect` express a random choice according to their argument.

We find `reflect` useful because it is an inverse of `reify0`, in the following sense. The type of `reflect` is  $'a pV \rightarrow 'a$ , and the type of `reify0` is  $(\text{unit} \rightarrow 'a) \rightarrow 'a pV$ . Informally, `reflect` turns a search tree into a stochastic expression, whereas `reify0` turns a stochastic expression (encapsulated in a thunk) into a search tree. For example, our lawn model in §1.2 defines a thunk `grass_model` of type `unit → bool`; the stochastic expression `grass_model ()` produces a random boolean. If we pass `grass_model` to the function `reify0`, we obtain

```
let gm_tree = reify0 grass_model
  ↪ val gm_tree : bool pV = [(0.3, C <fun>); (0.7, C <fun>)]
```

The result `gm_tree` reifies the stochastic expression `grass_model ()` in the form of a tree. We can `explore gm_tree` to get a shallower search tree describing the same distribution:

```
let gm_tree1 = explore (Some 4) gm_tree
→ val gm_tree1 : bool pV = [(0.2838, V true); (0.322, V false)]
```

Applying the function `reflect` to the trees `gm_tree` and `gm_tree1` produces two stochastic expressions that express the same distribution as `grass_model ()`, the stochastic expression we started with. Using `reify0`, we can turn these stochastic expressions into equivalent search trees.

These observations on `reify0` and `reflect` hold in general: for each search tree `pv` and stochastic expression `e`,

```
reify0 (fun () -> reflect pv) ≡ pv
reflect (reify0 (fun () -> e)) ~ e
```

where  $\equiv$  means that two search trees are equivalent and  $\sim$  means that two stochastic expressions are equivalent in that they express the same distribution. Thus, `reify0` and `reflect` form a *reification-reflection* pair [11].

As we just observed, any terminating stochastic expression is equivalent to reifying it then reflecting the resulting search tree. In this sense, reification-reflection is a sound program transformation, preserving program equivalence. It turns out that inference on the transformed program often takes less time and space. For example, if `e` is the expression `dist [(0.5, e1); (0.5, e2)]` where `e1` and `e2` are complex deterministic expressions, then reifying `e` gives the search tree `[(0.5, V v1); (0.5, V v2)]` where `v1` and `v2` are the results of `e1` and `e2`. Therefore, reflecting the search tree gives a stochastic expression that, albeit equivalent to `e`, can be evaluated faster because `e1` and `e2` no longer need to be evaluated. Reification followed by reflection can thus speed up inference—especially if the result of reification is (partially) flattened before reflection, because flattening reduces the tree by removing shallow failures and collating repeated values. For example, `reflect gm_tree1` uses the result of exact inference on the lawn model, so it runs faster than the original `grass_model ()`.

This optimization of reification followed by (partial) flattening and reflection gives rise to the inference technique known as variable elimination [6]. Its benefit is demonstrated by the following example, computing the XOR of  $n$  coin tosses.

```
let flips_xor n () =
  let rec loop n =
    if n = 1 then flip 0.5
    else flip 0.5 <> loop (n-1)
  in loop n
```

The search tree for `flips_xor 10` is a full binary tree of depth 10. The exploration of this tree for exact inference has to enumerate all of its 1024 leaves. Many parts of the computation are shared: each `flip 0.5` forks the search tree into two branches that make the same recursive call `loop (n-1)`. Variable elimination is thus much more efficient than brute-force exact inference:

```
let flips_xor' n () =
  let rec loop n =
```

```

if n = 1 then flip 0.5
else let r = reflect (explore None
                      (reify0 (fun () -> loop (n-1))))
      in flip 0.5 <> r
in loop n

```

Exact inference now explores 9 trees of depth 2, or a total of  $9 \times 4 + 2 = 38$  leaves. As is usual of dynamic programming, the optimization turns an exponential-time algorithm into a linear-time one, by sharing the computation for `loop (n-1)` across the two possible results of `flip 0.5` in `loop n`.

More generally, whenever we have a stochastic function `f`, we can replace it by the stochastic function

```

let bucket = memo (fun x -> explore None (reify0 (fun () -> f x)))
in fun x -> reflect (bucket x)

```

where `memo` memoizes a function. This latter stochastic function can then be invoked multiple times in different branches of the search tree, without repeatedly applying `f` to the same argument. This technique lets us handle benchmarks in Jaeger et al.'s collection [27] in at most a few seconds each. The memo table is also called a *bucket*, so variable elimination is also called *bucket elimination* [6]. It is similar to the nested-loop optimization of database join queries.

## 4 Importance Sampling with Look-Ahead

Because we can reify a probabilistic program as a lazy search tree, we (like any other user of our library) can implement inference algorithms as operations that explore the tree without slowing down the deterministic parts of the model. We have described exact inference algorithms (brute-force enumeration and variable elimination) above. In this section we implement approximate inference.

Recall that reifying a stochastic computation yields a lazy search tree, a data structure of the type `'a pV`. Exact inference traverses the whole tree. For many probabilistic models in the real world, however, the tree is too large to traverse completely, even infinite. Instead, we can sample from the distribution expressed by the probabilistic program, by tracing a path down the tree to a leaf. Whenever we arrive at a choice node, we select a branch at random, according to its weight. By repeating this process many times and computing the frequencies of those sampled values that are not failures, we can approximate the distribution.

This naive, or *rejection*, sampling method accurately approximates the probabilities of likely values. However, it poorly approximates the tail of the distribution, and it takes too long if most sampled paths end up in failure. Alas, a low success probability is quite common. An illustrative if contrived example is computing the `and` of  $n$  tosses of a fair coin by a drunk who often loses the coin.

```

let drunk_coin () =
  let toss = flip 0.5 in
  let lost = flip 0.9 in
  if lost then fail () else toss

```

```
let rec dcoin_and = function
| 1 -> drunk_coin ()
| n -> drunk_coin () && dcoin_and (n-1)
```

The function `drunk_coin` models the toss of a fair coin by a drunk; with the probability 0.9 the coin ends up in a gutter. The example is simple enough that the exact distribution of `and` of 10 tosses can be easily computed, either analytically or by complete exploration of the search tree:  $[(9.7656e-14, V \text{ true}) ; (0.05263, V \text{ false})]$ . Rejection sampling returns  $[(0.01, V \text{ false})]$  for 100 samples and  $[(0.052, V \text{ false})]$  for 10,000 samples. The probability for the drunk coin to come up heads ten times in a row is too low to be noticed by the rejection sampling. If we assert such an observation and wish to estimate its probability, the rejection sampling of `fun () -> dcoin_and 10 || fail ()` offers little help: even with 10,000 attempts, all samples are unsuccessful.

*Importance sampling* [15, 44] is an approximate inference strategy that improves upon rejection sampling by assigning weights to each sample. For example, because assigning `true` to the variable `lost` leads to failure, the sampler should not pick a value for `lost` randomly; instead, it should force `lost` to be `false`, but scale the weight of the resulting sample by 0.1 to compensate for this artificially induced success.

Pfeffer [39] introduced an importance-sampling algorithm to perform approximate inference for the probabilistic language IBAL. The main technique is called *evidence pushing* because it reduces nondeterminism in the search by pushing observed evidence towards their random cause in a stochastic program. Evidence pushing in the IBAL interpreter requires very sophisticated data-flow analysis on probabilistic programs, which our inference procedures cannot perform because they cannot inspect the source code of probabilistic programs.

Nevertheless, we can perform importance sampling on the lazy search tree that results from reifying a probabilistic program. The key is to explore the tree shallowly before committing to any random choice among its branches. Whenever this look-ahead comes across a branch that ends in failure, that branch is eliminated from the available choices. Similarly, whenever the look-ahead comes across a branch that ends in success, the successful result is registered as a sample, and that branch is again eliminated from the available choices. After the look-ahead, the sampler randomly selects among the remaining choices and repeats the process. The probability of choosing a branch is proportional to its probability mass. We also remember to scale the importance of all further samples by the total probability mass of the open branches among which we chose one. Since the look-ahead notices not only shallow failures but also shallow successes, one trace down the tree may yield more than one sample.

We have implemented importance sampling with look-ahead as described above. As one might expect, it is much more accurate than rejection sampling on programs like `dcoin_and`. For example, using our algorithm to sample 5000 times from `dcoin_and 10` estimates the distribution as  $[(8e-14, V \text{ true}) ; (0.0526, V \text{ false})]$ ; we now obtain quite an accurate estimate of the probability of the coin to come up heads ten times in a row. Recall that

rejection sampling failed to estimate this probability even with twice as many samples.

## 5 Lazy Evaluation

Because events in the world usually take place in a different order from when they are observed, values in a probabilistic program are usually produced in a different order from when they are observed. Just as in logic programming, this mismatch makes the search tree much larger, because the part of the search tree between making a random choice and observing its consequence is duplicated. To reduce this mismatch, we turn to lazy evaluation.

The following contrived example illustrates the mismatch that motivates lazy evaluation. We define the producer `flips p n` to be the random result of flipping `n` independent coins, each `true` with probability `p`. We also define the observer `trues n xs` to check that a list of `n` booleans consists entirely of `true`.

```
let rec flips p = function
  | 0 -> []
  | n -> let x = flip p in
    let xs = flips p (n - 1) in
    x :: xs

let rec trues n xs = match (n, xs) with
  | (0, [])      -> true
  | (n, (x::xs)) -> x && trues (n - 1) xs
```

The program `if trues 20 (flips 0.5 20) then () else fail()` expresses the observation that a sequence of 20 fair-coin flips comes out all `true`. It is easy to see mathematically that the probability of this observation is  $2^{-20}$ , but the inference algorithms described above need to explore millions of tree nodes to compute this probability as nonzero, because the entire list of 20 booleans is produced before any element of the list is observed. It is especially common for this kind of dissociation between production and observation to arise naturally in probabilistic models of perception; for example, to parse a given sentence is to observe that a randomly produced sentence is equal to the given sentence.

To reduce the delay between the production and observation of random values, we can revise the probabilistic program using lazy evaluation. We provide a library function `letlazy`, which adds memoization to a thunk. For example, `letlazy (fun () -> flip p)` below creates a thunk that evaluates `flip p` the first time it is invoked, then returns the same boolean thereafter. The data constructors `LNil` and `LCons` are lazy analogues of the list constructors `[]` and `::`.

```
let rec flips p = function
  | 0 -> LNil
  | n -> let x = letlazy (fun () -> flip p) in
    let xs = letlazy (fun () -> flips p (n - 1)) in
    LCons (x, xs)
```

```
let rec trues n xs = match (n, xs) with
  | (0, LNil)          -> true
  | (n, LCons (x, xs)) -> x () && trues (n - 1) (xs ())
```

To be sure, `letlazy` does not memoize the lazy search tree that results from reifying a probabilistic program. That would make the tree stay longer in memory but not have any fewer nodes. What `letlazy` memoizes is an object value within the probabilistic program. Importance sampling then requires only one sample to compute the probability  $2^{-20}$  exactly.

It is incorrect to implement `letlazy` using OCaml's built-in laziness, because the latter uses the mutable state of the underlying hardware, which does not 'snap back' when the inference procedure backtracks to an earlier point in the execution of the probabilistic program. Instead, we implement `letlazy` by using or emulating thread-local storage [23].

For equational reasoning, it is useful for the expression `letlazy t` (where `t` is a thunk) to denote the same distribution as `let v = t () in fun () -> v`. If the body of `t` involves observing evidence (for example, if `t` is `fail`), then this equivalence may not hold because, if the thunk returned by `letlazy t` is never invoked, then neither is `t`. To maintain the equivalence, we can change `letlazy` to always invoke `t` at least once. This variation on lazy evaluation is called *delayed evaluation* [39].

## 6 Performance Testing

The examples above are simple and contrived to explain inference algorithms. We have also expressed and performed inference on a model of realistic complexity, namely Pfeffer's [39] music model. This model describes the transformation of melodic motives in classical music as a combination of transposition, inversion, deletion and insertion operations. The number of operations and their sequence are random, as is the effect of transposition on each note. Given a transformed melody, we wish to obtain the conditional distribution of original motives that could have given rise to the transformed melody.

Exact inference is infeasible for this model because the search tree is very wide and deep: 40% of the nodes have 8 or more children, and the depth of the tree reaches 10. Rejection sampling is also useless because the overall probability of matching the given transformed melody is quite low—on the order of  $10^{-7}$ . The importance sampling procedure described in Section 4, with the crucial help of lazy evaluation described in Section 5, estimated the conditional probabilities with roughly the same accuracy as Pfeffer's state-of-the-art implementation of evidence pushing for his IBAL language.

For concreteness, we describe a typical instance of this test. The original melody (#1) and transformed melody (#1) are two different motives taken from a movement in an early piano sonata by Beethoven. On a 2GHz Pentium 4 computer, we ran 100 trials in which Pfeffer's importance sampler had 30 seconds per trial (following his original testing), 100 trials in which our sampler had 30

seconds per trial, and another 100 trials in which our sampler had 90 seconds per trial.

Given 30 seconds, Pfeffer’s sampler runs about 70,000 times; it estimates the evidence probability to be  $\exp(-14.6)$  on average, with the standard deviation  $\exp(-15.1)$ . Also given 30 seconds, our sampler runs about 10,000 times; it estimates the evidence probability to be  $\exp(-13.7)$  on average, with the standard deviation  $\exp(-13.8)$ . When given 90 seconds, our sampler runs about 30,000 times; it estimates the evidence probability to be  $\exp(-13.6)$  on average, with the standard deviation  $\exp(-14.4)$ . For both samplers, the histogram of estimated probabilities that result from repeated trials is skewed to the right; the skewness is roughly the same for the two samplers on each inference problem.

## 7 Conclusions

We have presented a *very shallow* embedding of a DSL for probabilistic modeling. This embedding exemplifies the following advantages of very shallow embeddings of DSLs in general.

**Negligible notational overhead.** We represent probabilistic programs as ordinary OCaml programs, so random integers can be added using ordinary `+`, random variables can be bound using ordinary `let`, and random processes can be defined using ordinary  $\lambda$ -abstraction.

**Type safety.** OCaml infers type signatures and detects type errors in stochastic expressions just as in all other OCaml expressions. Our embedding adds no type tags to run-time values.

**No run-time overhead for deterministic code.** Deterministic constants and operations used in a probabilistic model, including those in third-party compiled libraries, run at full speed as if our probabilistic library were not present. (This lack of overhead is similar to how non-privileged client code runs under VMWare as if the hypervisor were not present [9].)

**Ease of learning and maintenance.** We use the unmodified OCaml system as is. Probabilistic programs and inference procedures can use all facilities of the host language, including I/O, libraries, and a module system.

Our implementation is based on reifying probabilistic models as lazy search trees. This reification allows us to evaluate the same model using multiple inference algorithms, including algorithms such as variable elimination and importance sampling that were thought to require a more deeply embedded probabilistic language. We can also step through the random choices made by a model by tracing through its search tree.

The size and complexity of the models we have run are in the ballpark of state-of-the art systems such as IBAL [39]. We plan to exercise this system by implementing larger probabilistic models of real-world phenomena, in particular the nested inference that agents conduct about each other.

## References

- [1] Carette, J., Kiselyov, O., Shan, C.-c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* (2008) (in press)
- [2] Cussens, J.: Logic-based formalisms for statistical relational learning. In: [18], ch. 9, pp. 269–290 (2007)
- [3] Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark (1989), <http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz>
- [4] Danvy, O., Filinski, A.: Abstracting control. In: *Proceedings of the 1990 ACM conference on Lisp and functional programming*, pp. 151–160. ACM Press, New York (1990)
- [5] Daumé III, H.: Hierarchical Bayes compiler (2007), <http://www.cs.utah.edu/~hal/HBC/>
- [6] Dechter, R.: Bucket elimination: A unifying framework for probabilistic inference. In: Jordan, M.I. (ed.) *Learning and inference in graphical models*. Kluwer, Dordrecht (1998); Paperback: *Learning in Graphical Models*. MIT Press
- [7] Domingos, P., Richardson, M.: Markov logic: A unifying framework for statistical relational learning. In: [18], ch. 12, pp. 339–371 (2007)
- [8] Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. *Journal of Functional Programming* 16(1), 21–34 (2006)
- [9] Feigin, B., Mycroft, A.: Jones optimality and hardware virtualization: A report on work in progress. In: Glück, R., de Moor, O. (eds.) *Proceedings of the 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, pp. 169–175. ACM Press, New York (2008)
- [10] Felleisen, M., Friedman, D.P., Duba, B.F., Merrill, J.: Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University (1987)
- [11] Filinski, A.: Representing monads. In: *POPL 1994: Conference record of the annual ACM symposium on principles of programming languages*, pp. 446–457. ACM Press, New York (1994)
- [12] Filinski, A.: Representing layered monads. In: *POPL 1999: Conference record of the annual ACM symposium on principles of programming languages*, pp. 175–188. ACM Press, New York (1999)
- [13] Fischer, B., Schumann, J.: AutoBayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming* 13(3), 483–508 (2003)
- [14] Forbes, J., Huang, T., Kanazawa, K., Russell, S.J.: The BATmobile: Towards a Bayesian automated taxi. In: *Proceedings of the 14th international joint conference on artificial intelligence*, pp. 1878–1885. Morgan Kaufmann, San Francisco (1995)
- [15] Fung, R., Chang, K.-C.: Weighing and integrating evidence for stochastic simulation in Bayesian networks. In: Henrion, M., Shachter, R.D., Kanal, L.N., Lemmer, J.F. (eds.) *Proceedings of the 5th conference on uncertainty in artificial intelligence* (1989), pp. 209–220. North-Holland, Amsterdam (1990)
- [16] Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Džeroski, S., Lavrač, N. (eds.) *Relational data mining*, ch. 13, pp. 307–335. Springer, Berlin (2001)
- [17] Getoor, L., Friedman, N., Koller, D., Pfeffer, A., Taskar, B.: Probabilistic relational models. In: [18], ch. 5, pp. 129–174 (2007)

- [18] Getoor, L., Taskar, B. (eds.): *Introduction to statistical relational learning*. MIT Press, Cambridge (2007)
- [19] Giry, M.: A categorical approach to probability theory. In: Banaschewski, B. (ed.) *Categorical aspects of topology and analysis: Proceedings of an international conference held at Carleton University, Ottawa*. Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer, Berlin (1982)
- [20] Goodman, N.D., Mansinghka, V.K., Roy, D., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: McAllester, D.A., Myllymäki, P. (eds.) 2008: *Proceedings of the 24th conference in uncertainty in artificial intelligence*, pp. 220–229. AUAI Press (2008)
- [21] Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Peyton Jones, S.L. (ed.) *Functional programming languages and computer architecture: 7th conference*, pp. 12–23. ACM Press, New York (1995)
- [22] Halpern, J.Y.: An analysis of first-order logics of probability. *Artificial Intelligence* 46, 311–350 (1990)
- [23] Haynes, C.T.: Logic continuations. *Journal of Logic Programming* 4(2), 157–176 (1987)
- [24] Hinze, R.: Deriving backtracking monad transformers. In: ICFP 2000: *Proceedings of the ACM international conference on functional programming*. ACM SIGPLAN Notices, vol. 35(9), pp. 186–197. ACM Press, New York (2000)
- [25] Hudak, P.: Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es), 196 (1996)
- [26] Hughes, J.: The design of a pretty-printing library. In: Jeuring, J., Meijer, E. (eds.) *AFP 1995*. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)
- [27] Jaeger, M., Lidman, P., Mateo, J.L.: Comparative evaluation of probabilistic logic languages and systems: A work-in-progress report. In: *Proceedings of mining and learning with graphs* (2007), <http://www.cs.aau.dk/~jaeger/plsystems/>
- [28] Kersting, K., De Raedt, L.: Bayesian logic programming: Theory and tool. In: [18], ch. 10, pp. 291–321 (2007)
- [29] Kiselyov, O.: Native delimited continuations in (byte-code) OCaml (2006), <http://okmij.org/ftp/Computation/Continuations.html#caml-shift>
- [30] Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: AAAI 1997: *Proceedings of the 14th national conference on artificial intelligence*. The American Association for Artificial Intelligence, pp. 740–747. AAAI Press, Menlo Park (1997)
- [31] Landin, P.J.: The next 700 programming languages. *Communications of the ACM* 9(3), 157–166 (1966)
- [32] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: [18], ch. 13, pp. 373–398 (2007)
- [33] Mogensen, A.E.T.: Self-applicable online partial evaluation of the pure lambda calculus. In: *Proceedings of the 1995 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, pp. 39–44. ACM Press, New York (1995)
- [34] Muggleton, S., Pahlavi, N.: Stochastic logic programs: A tutorial. In: [18], ch. 11, pp. 323–338 (2007)
- [35] Murphy, K.: Bayes Net Toolbox for Matlab (2007), <http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html>
- [36] Murphy, K.: Software for graphical models: A review. *International Society for Bayesian Analysis Bulletin* 14(4), 13–15 (2007)

- [37] Pearl, J.: Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan Kaufmann, San Francisco (1988) (Revised 2nd printing, 1998)
- [38] Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: [18], ch. 14, pp. 399–432 (2007)
- [39] Pfeffer, A.: A general importance sampling algorithm for probabilistic programs. Tech. Rep. TR-12-07, Harvard University (2007)
- [40] Phillips, A., Cardelli, L.: Efficient, correct simulation of biological processes in the stochastic pi-calculus. In: Calder, M., Gilmore, S. (eds.) CMSB 2007. LNCS (LNBI), vol. 4695, pp. 184–199. Springer, Heidelberg (2007)
- [41] Poole, D., Mackworth, A.: Artificial intelligence: Foundations of computational agents. Cambridge University Press, Cambridge (2009)
- [42] Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL 2002: Conference record of the annual ACM symposium on principles of programming languages, pp. 154–165. ACM Press, New York (2002)
- [43] Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems 31(2), 161–176 (2008)
- [44] Shachter, R.D., Peot, M.A.: Simulation approaches to general probabilistic inference on belief networks. In: Henrion, M., Shachter, R.D., Kanal, L.N., Lemmer, J.F. (eds.) Proceedings of the 5th conference on uncertainty in artificial intelligence (1989), pp. 221–234. North-Holland, Amsterdam (1990)
- [45] Sutton, C., McCallum, A.: An introduction to conditional random fields for relational learning. In: [18], ch. 4, pp. 93–127 (2007)
- [46] Taskar, B., Abbeel, P., Wong, M.-F., Koller, D.: Relational Markov networks. In: [18], ch. 6, pp. 175–199 (2007)
- [47] Thiemann, P.: Combinators for program generation. Journal of Functional Programming 9(5), 483–525 (1999)
- [48] Wadler, P.L.: The essence of functional programming. In: POPL 1992: Conference record of the annual ACM symposium on principles of programming languages, pp. 1–14. ACM Press, New York (1992)
- [49] Wand, M., Vaillancourt, D.: Relating models of backtracking. In: ICFP 2004: Proceedings of the ACM international conference on functional programming. ACM Press, New York (2004)

# Operator Language: A Program Generation Framework for Fast Kernels\*

Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel

Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh PA 15213, USA

{franzf, fdemesma, dmcfarli, pueschel}@ece.cmu.edu

**Abstract.** We present the Operator Language (OL), a framework to automatically generate fast numerical kernels. OL provides the structure to extend the program generation system Spiral beyond the transform domain. Using OL, we show how to automatically generate library functionality for the fast Fourier transform and multiple non-transform kernels, including matrix-matrix multiplication, synthetic aperture radar (SAR), circular convolution, sorting networks, and Viterbi decoding. The control flow of the kernels is data-independent, which allows us to cast their algorithms as operator expressions. Using rewriting systems, a structural architecture model and empirical search, we automatically generate very fast C implementations for state-of-the-art multicore CPUs that rival hand-tuned implementations.

**Keywords:** Library generation, program generation, automatic performance tuning, high performance software, multicore CPU.

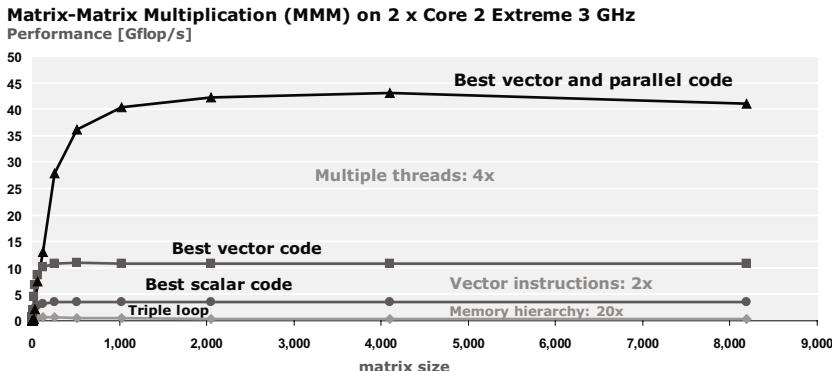
## 1 Introduction

In many software applications, runtime performance is crucial. This is particularly true for compute-intensive software that is needed in a wide range of application domains including scientific computing, image/video processing, communication and control. In many of these applications, the bulk of the work is performed by well-defined mathematical functions or *kernels* such as matrix-matrix multiplication (MMM), the discrete Fourier transform (DFT), convolution, or others. These functions are typically provided by high performance libraries developed by expert programmers. A good example is Intel's Integrated Performance Primitives (IPP) [1], which provides around 10,000 kernels that are used by commercial developers worldwide.

Unfortunately, the optimization of kernels has become extraordinarily difficult due to the complexity of current computing platforms. Specifically, to run fast on, say, an off-the-shelf Core 2 Duo, a kernel routine has to be optimized for the memory hierarchy, use

---

\* This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), the ARO grant W911NF0710416, and by Intel Corp. and Mercury Computer Systems, Inc.



**Fig. 1.** Performance of four double precision implementations of matrix-matrix multiplication. The operations count is exactly the same. The plot is taken from [2].

SSE vector instructions, and has to be multithreaded. Without these optimizations, the performance loss can be significant. To illustrate this, we show in Fig. 1 the performance of four implementations of MMM (all compiled with the latest Intel compiler) measured in giga-floating point operations per second (Gflop/s). At the bottom is a naive triple loop. Optimizing for the memory hierarchy yields about 20x improvement, explicit use of SSE instructions another 2x, and threading for 4 cores another 4x for a total of 160x.

In other words, the compiler cannot perform the necessary optimizations for two main reasons. First, many optimizations require high level algorithm or domain knowledge; second, there are simply too many optimization choices that a (deterministic) compiler cannot explore. So the optimization is left with the programmer and is often platform specific, which means it has to be repeated whenever a new platform is released.

We believe that the above poses an ideal scenario for the application of domain-specific languages (DSLs) and program generation techniques to automatically generate fast code for kernel functions directly from a specification. First, many kernel functions are fixed which helps with the design of a language describing their algorithms, and available algorithms have to be formalized only once. Second, since many optimizations require domain knowledge, there is arguably no way around a DSL if automation is desired. Third, the problem has real-world relevance and also has to address the very timely issue of parallelism.

**Contribution of this paper.** In this paper we present a program generation framework for kernel functions such as MMM, linear transforms (e.g. DFT), Viterbi decoding, and others. The generator produces code directly from a kernel specification and the performance of the code is, for the kernels considered, often competitive with the best available hand-written implementations. We briefly discuss the main challenges and how they are addressed by our approach:

- *Algorithm generation.* We express algorithms at a high abstraction level using a DSL called operator language (OL). Since our domain is mathematical, OL is derived from mathematics and it is declarative in that it describes the structure of

a computation in an implementation-independent form. Divide-and-conquer algorithms are described as OL breakdown rules. By recursively applying these rules a space of algorithms for a desired kernel can be generated. The OL compiler translates OL into actual C code.

- *Algorithm optimization.* We describe platforms using a very small set of parameters such as the vector datatype length or the number of processors. Then we identify OL optimization rules that restructure algorithms. Using these rules together with the breakdown rules yields optimized algorithms. It is an example of rule-based programming and effectively solves the problem of domain specific optimizations. Beyond that, we explore and time choices for further optimization.

Our approach has to date a number of limitations. First, we require that the kernels are for fixed input size. Second, the kernels have to be input data independent. Third, the algorithms for a kernel have to possess a suitable structure to be handled efficiently.

This paper builds on our prior work on Spiral, which targets program generation for linear transforms based on the language SPL [3,4,5,6]. Here we show for the first time how to go beyond this domain by extending SPL to OL using a few kernels as examples.

**Related work.** The area of program generation (also called generative programming) has gained considerable interest in recent years [7,8,9,10,11]. The basic goal is to reduce the development, maintenance, and analysis of software. Among the key tools for achieving these goals, domain-specific languages provide a compact representation that raises the level of abstraction for specific problems and hence enables the manipulation of programs [12,13,14,15,16]. However, this work has to date rarely considered numerical problems and focused on correctness rather than performance.

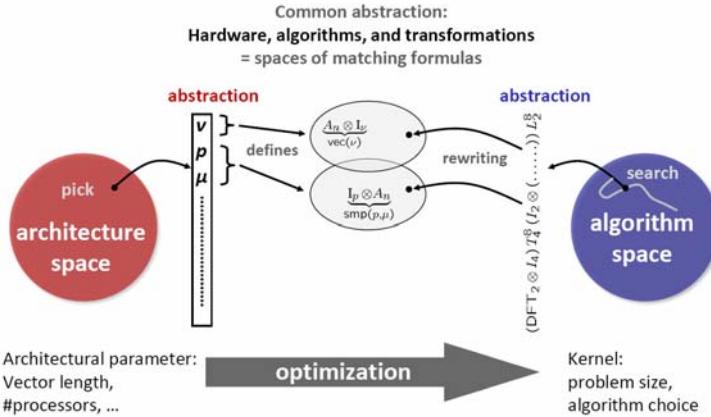
ATLAS [17] is an automatic performance tuning system that optimizes basic linear algebra functionality (BLAS) using a code generator to generate kernels that are used in a blocked parameterized library. OSKI (and its predecessor Sparsity) [18] is an automatic performance tuning system for matrix-vector multiplication for sparse matrices. The adaptive FFT library FFTW [19] implements an adaptive version of the Cooley-Tukey FFT algorithm. It contains automatically generated code blocks (codelets) [20]. The TCE [21] automates implementing tensor contraction equations used in quantum chemistry. FLAME [22] automates the derivation and implementation of dense linear algebra algorithms.

Our approach draws inspiration and concepts from symbolic computation and rule-based programming. Rewriting systems are reviewed in [23]. Logic programming is discussed in [24]. An overview of functional programming can be found in [25].

## 2 Program Generation Framework for Fast Kernels

The goal of this paper is to develop a program generator for high performance kernels. A kernel is a fixed function that is used as library routine in different important applications. We require that the input size for the kernel is fixed. Examples of kernels include a discrete Fourier transform for input size 64 or the multiplication of two  $8 \times 8$  matrices.

To achieve high performance, our generator produces programs that are optimized to the specifics of the targeted computing platform. This includes tuning to the memory



**Fig. 2.** Our approach to program generation: The spaces of architectures (left) and algorithms (right) are abstracted and joined using the operator language framework

hierarchy and the efficient use of vector instructions and threading (if available). To do this efficiently, a few platform parameters are provided to the generator and the platform is assumed to be available for benchmarking of the generated code, so an exploration of alternatives is possible.

In summary, the input to our generator is a kernel specification (e.g.,  $DFT_{64}$ ) and platform parameters (e.g., 2 cores); the output is a fast C function for the kernel. Performance is achieved through both structural optimization of available algorithms based on the platform parameters and search over alternatives for the fastest. Intuitively, the search optimizes for the memory hierarchy by considering different divide-and-conquer strategies.

Fig. 2 shows a very high level description of our approach, which consists of three key components: 1) A DSL called operator language (OL) to describe algorithms for kernels (right bubble); 2) the use of tags to describe architecture features (left bubble); 3) a common abstraction of architecture and algorithms using tagged OL. We briefly describe the three components and then give detailed explanations in separate subsections.

**Operator language (OL): Kernels and algorithms.** OL is a mathematical DSL to describe structured divide-and-conquer algorithms for data-independent kernels. These algorithms are encoded as breakdown rules and included into our generator. By recursively applying these rules, the generator can produce a large space of alternative algorithms for the desired kernel. OL is platform-independent and declarative in nature; it is an extension of SPL [3,4] that underlies Spiral.

**Hardware abstraction: Tags.** We divide hardware features into parameterized paradigms. In this paper we focus on two examples: vector processing abilities (e.g., SSE on Intel P4 and later) with vector length  $\nu$ , and shared memory parallelism with  $p$  cores.

**Common abstraction: Tagged operator language.** The key challenge in achieving performance is to optimize algorithms for a given paradigm. We do this in steps. First,

we introduce the hardware paradigms as tags in OL. Second, we identify basic OL expressions that can be efficiently implemented on that paradigm. These will span a sublanguage of OL that can be implemented efficiently. Finally, we add generic OL rewriting rules in addition to the breakdown rules describing algorithms. The goal is that the joint set of rules now produces structurally optimized algorithms for the desired kernel. If there are still choices, feedback driven search is used to find the fastest solution.

Besides the above, our generator requires a compiler that translates OL into actual C, performing additional low level optimizations (Section 3). However, the focus of this paper is the OL framework described next in detail.

## 2.1 Operator Language: Kernels and Algorithms

In this section we introduce the Operator Language (OL), a domain-specific language designed to describe structured algorithms for data-independent kernel functions. The language is declarative in that it describes the structure of the dataflow and the data layout of a computation, thus enabling algorithm manipulation and structural optimization at a high level of abstraction. OL is a generalization of SPL [3,4], which is designed for linear transforms.

The main building blocks of OL are *operators*, combined into *operator formulas* by *higher-order operators*. We use OL to describe recursive (divide-and-conquer) algorithms for important kernels as *breakdown rules*. The combination of these rules then produces a space of alternative algorithms for this kernel.

**Operators.** Operators are  $n$ -ary functions on vectors: an operator of arity  $(r, s)$  consumes  $r$  vectors and produces  $s$  vectors. An operator can be (multi)linear or not. Linear operators of arity  $(1, 1)$  are precisely linear transforms, i.e., mappings  $x \mapsto Mx$ , where  $M$  is a fixed matrix. We often refer to linear transforms as matrices. When necessary, we will denote  $A_{m \times n \rightarrow p}$  an operator  $A$  going from  $\mathbb{C}^m \times \mathbb{C}^n$  into  $\mathbb{C}^p$ .

Matrices are viewed as vectors stored linearized in memory in row major order. For example, the operator that transposes an  $m \times n$  matrix<sup>1</sup>, denoted by  $L_n^{mn}$ , is of arity  $(1, 1)$ . Table 1 defines a set of basic operators that we use.

**Kernels.** In this paper, a computational kernel is an operator for which we want to generate fast code. We will use matrix-matrix multiplication and the discrete Fourier transform as running examples to describe OL concepts. However, we also used OL to capture other kernels briefly introduced later, namely: circular convolution, sorting networks, Viterbi decoding, and synthetic aperture radar (SAR) image formation.

We define the *matrix-multiplication*  $\text{MMM}_{m,k,n}$  as an operator that consumes two matrices and produces one<sup>2</sup>:

$$\text{MMM}_{m,k,n} : \mathbb{R}^{mk} \times \mathbb{R}^{kn} \rightarrow \mathbb{R}^{mn}; (\mathbf{A}, \mathbf{B}) \mapsto \mathbf{AB}$$

<sup>1</sup> The transposition operator is often referred to as the *stride* permutation or *corner turn*.

<sup>2</sup> We use this definition for explanation purposes; the  $\text{MMM}$  required by BLAS [17] has a more general interface.

**Table 1.** Definition of basic operators. The operators are assumed to operate on complex numbers but other base sets are possible. Boldface fonts represent vectors or matrices linearized in memory. Superscripts  $U$  and  $L$  represent the upper and lower half of a vector. A vector is sometimes written as  $\mathbf{x} = (x_i)$  to identify the components.

name	definition
<i>Linear, arity (1,1)</i>	
identity	$I_n : \mathbb{C}^n \rightarrow \mathbb{C}^n; \mathbf{x} \mapsto \mathbf{x}$
vector flip	$J_n : \mathbb{C}^n \rightarrow \mathbb{C}^n; (x_i) \mapsto (x_{n-i})$
transposition of an $m \times n$ matrix	$L_m^{mn} : \mathbb{C}^{mn} \rightarrow \mathbb{C}^{mn}; \mathbf{A} \mapsto \mathbf{A}^T$
matrix $M \in \mathbb{C}^{m \times n}$	$M : \mathbb{C}^n \rightarrow \mathbb{C}^m; \mathbf{x} \mapsto M\mathbf{x}$
<i>Bilinear, arity (2,1)</i>	
Point-wise product	$P_n : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}^n; ((x_i), (y_i)) \mapsto (x_i y_i)$
Scalar product	$S_n : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}; ((x_i), (y_i)) \mapsto \Sigma(x_i y_i)$
Kronecker product	$K_{m \times n} : \mathbb{C}^m \times \mathbb{C}^n \rightarrow \mathbb{C}^{mn}; ((x_i), \mathbf{y}) \mapsto (x_i \mathbf{y})$
<i>Others</i>	
Fork	$\text{Fork}_n : \mathbb{C}^n \rightarrow \mathbb{C}^n \times \mathbb{C}^n; \mathbf{x} \mapsto (\mathbf{x}, \mathbf{x})$
Split	$\text{Split}_n : \mathbb{C}^n \rightarrow \mathbb{C}^{n/2} \times \mathbb{C}^{n/2}; \mathbf{x} \mapsto (\mathbf{x}^U, \mathbf{x}^L)$
Concatenate	$\oplus_n : \mathbb{C}^{n/2} \times \mathbb{C}^{n/2} \rightarrow \mathbb{C}^n; (\mathbf{x}^U, \mathbf{x}^L) \mapsto \mathbf{x}$
Duplication	$\text{dup}_n^m : \mathbb{C}^n \rightarrow \mathbb{C}^{nm}; \mathbf{x} \mapsto \mathbf{x} \otimes I_m$
Min	$\text{min}_n : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}^n; (\mathbf{x}, \mathbf{y}) \mapsto (\min(x_i, y_i))$
Max	$\text{max}_n : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}^n; (\mathbf{x}, \mathbf{y}) \mapsto (\max(x_i, y_i))$

The *discrete Fourier transform*  $\text{DFT}_n$  is a linear operator of arity (1, 1) that performs the following matrix-vector product:

$$\text{DFT}_n : \mathbb{C}^n \rightarrow \mathbb{C}^n; \mathbf{x} \mapsto [e^{-2\pi i k l / n}]_{0 \leq k, l < n} \mathbf{x}$$

**Higher-order operators.** Higher-order operators are functions on operators. A simple example is the *composition*, denoted in standard infix notation by  $\circ$ . For instance,

$$L_n^{mn} \circ P_{mn}$$

is the arity (2, 1) operator that first multiplies point-wise two matrices of size  $m \times n$ , and then transposes the result.

The *cross product* of two operators applies the first operator to the first input set and the second operator to the second input set, and then combines the outputs. For example,

$$L_n^{mn} \times P_{mn}$$

is the arity (3, 2) operator that transposes its first argument and multiplies the second and third argument pointwise, producing two output vectors.

The most important higher order operator in this paper is the *tensor product*. For linear operators  $A, B$  of arity (1,1) (i.e., matrices), the tensor product corresponds to the tensor or Kronecker product of matrices:

$$A \otimes B = [a_{k,l} B], \quad A = [a_{k,l}].$$

An important special case is the tensor product of an identity matrix and an arbitrary matrix,

$$I_n \otimes A = \begin{bmatrix} A \\ & \ddots \\ & & A \end{bmatrix}.$$

This can be interpreted as applying  $A$  to a list of  $n$  contiguous subvectors of the input vector. Conversely,  $A \otimes I_n$  applies  $A$  multiple times to subvectors extracted at stride  $n$ .

The Kronecker product is known to be useful for concisely describing DFT algorithms as fully developed by Van Loan [26] and is the key construct in the program generator Spiral for linear transforms [4]. Its usefulness is in the concise way that it captures loops, data independence, and parallelism.

We now formally extend the tensor product definition to more general operators, focusing on the case of two operators with arity (2,1); generalization is straightforward.

Let  $\mathcal{A} : \mathbb{C}^p \times \mathbb{C}^q \rightarrow \mathbb{C}^r$  be a *multi-linear* operator and let  $B : \mathbb{C}^m \times \mathbb{C}^n \rightarrow \mathbb{C}^k$  be any operator. We denote the  $i$ th canonical basis vector of  $\mathbb{C}^n$  with  $\mathbf{e}_i^n$ . Then

$$\begin{aligned} (\mathcal{A} \otimes B)(\mathbf{x}, \mathbf{y}) &= \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} \mathcal{A}(\mathbf{e}_i^p, \mathbf{e}_j^q) \otimes B((\mathbf{e}_i^p)^T \otimes I_m) \mathbf{x}, (\mathbf{e}_j^q)^T \otimes I_n) \mathbf{y} \\ (B \otimes \mathcal{A})(\mathbf{x}, \mathbf{y}) &= \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} B((I_m \otimes \mathbf{e}_i^p)^T) \mathbf{x}, (I_n \otimes \mathbf{e}_j^q)^T \mathbf{y}) \otimes \mathcal{A}(\mathbf{e}_i^p, \mathbf{e}_j^q) \end{aligned}$$

Intuitively,  $\mathcal{A}$  describes the coarse structure of the algorithm and captures how to operate on the chunks of data produced by  $B$ . Therefore, the structure of the operations  $\mathcal{A}$  and  $\mathcal{A} \otimes B$  is similar. For instance, consider the point-wise product  $P_2$  and the tensor product  $P_2 \otimes B$  (we denote with the superscripts  $U$  and  $L$  the upper and lower halves of a vector):

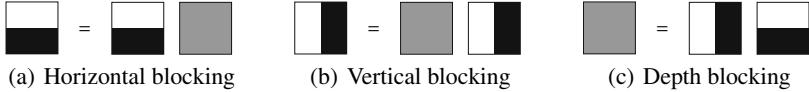
$$\left( \begin{array}{c} x_0 \\ x_1 \end{array} \right) \times \left( \begin{array}{c} y_0 \\ y_1 \end{array} \right) \xrightarrow{P_2} \left( \begin{array}{c} x_0 \cdot y_0 \\ x_1 \cdot y_1 \end{array} \right) \quad \left( \begin{array}{c} \mathbf{x}^U \\ \mathbf{x}^L \end{array} \right) \times \left( \begin{array}{c} \mathbf{y}^U \\ \mathbf{y}^L \end{array} \right) \xrightarrow{P_2 \otimes B} \left( \begin{array}{c} B(\mathbf{x}^U, \mathbf{y}^U) \\ B(\mathbf{x}^L, \mathbf{y}^L) \end{array} \right).$$

We show another example by selecting the Kronecker product  $K_{2 \times 2}$  (now viewed as operator of arity (2, 1) on vectors, see Table 1, not viewed as higher order operator). Again, the multilinear part of the tensor product describes how blocks are arranged and the non-linear part  $B$  prescribes what operations to perform on the blocks:

$$\left( \begin{array}{c} x_0 \\ x_1 \end{array} \right) \times \left( \begin{array}{c} y_0 \\ y_1 \end{array} \right) \xrightarrow{K_{2 \times 2}} \left( \begin{array}{c} \frac{x_0 \cdot y_0}{x_0 \cdot y_1} \\ \frac{x_1 \cdot y_0}{x_1 \cdot y_1} \end{array} \right) \quad \left( \begin{array}{c} \mathbf{x}^U \\ \mathbf{x}^L \end{array} \right) \times \left( \begin{array}{c} \mathbf{y}^U \\ \mathbf{y}^L \end{array} \right) \xrightarrow{K_{2 \times 2} \otimes B} \left( \begin{array}{c} \frac{B(\mathbf{x}^U, \mathbf{y}^U)}{B(\mathbf{x}^U, \mathbf{y}^L)} \\ \frac{B(\mathbf{x}^L, \mathbf{y}^U)}{B(\mathbf{x}^L, \mathbf{y}^L)} \end{array} \right).$$

Comparing these two examples,  $\mathcal{A} = P$  yields a tensor product in which only corresponding parts of the input vectors are computed on, whereas  $\mathcal{A} = K$  yields a tensor product in which *all* combinations are computed on.

**Recursive algorithms as OL breakdown rules.** We express recursive algorithms for kernels as OL equations written as *breakdown rules*.



**Fig. 3.** Blocking matrix multiplication along each one of the three dimensions. For the horizontal and vertical blocking, the white (black) part of the result is computed by multiplying the white (black) part of the blocked input with the other, gray, input. For the depth blocking, the result is computed by multiplying both white parts and both black parts and adding the results.

The first example we consider is a blocked matrix multiplication. While it does not improve the arithmetic cost over a naive implementation, blocking increases reuse and therefore can improve performance [27,28]. We start with blocking along one dimension.

Fig. 3(a) shows a picture of a horizontally blocked matrix. Each part of the result  $C$  is produced by multiplying the corresponding part of  $A$  by the whole matrix  $B$ . In OL, this is expressed by a tensor product with a Kronecker product:

$$\text{MMM}_{m,k,n} \rightarrow \mathbf{K}_{m/m_b \times 1} \otimes \text{MMM}_{m_b,k,n}. \quad (1)$$

Note that the number of blocks  $m/m_b$  is a degree of freedom under the constraint that  $m$  is divisible by  $m_b$ <sup>3</sup>; in the picture,  $m/m_b$  is equal to 2 (white block and black block).

Fig. 3(b) shows a picture of a vertically tiled matrix. The result is computed by multiplying parts of the matrix  $B$  with  $A$  so the underlying tensor product again uses a Kronecker product. However, since matrices are linearized in row-major order, we now need two additional stages: a pre-processing stage where the parts of  $B$  are de-interleaved and a post-processing stage where the parts of  $C$  are re-interleaved<sup>4</sup>:

$$\text{MMM}_{m,k,b} \rightarrow (I_m \otimes L_{n/n_b}^n) \circ (\text{MMM}_{m,k,n_b} \otimes \mathbf{K}_{1 \times n/n_b}) \circ (I_{km} \times (I_k \otimes L_{n/n_b}^n)). \quad (2)$$

Finally, Fig. 3(c) shows a picture of a matrix tiled in the “depth”. This time, parts of one input corresponds to parts of the other input but all results are added together. Therefore, the corresponding tensor product is not done with a Kronecker product but with a scalar product:

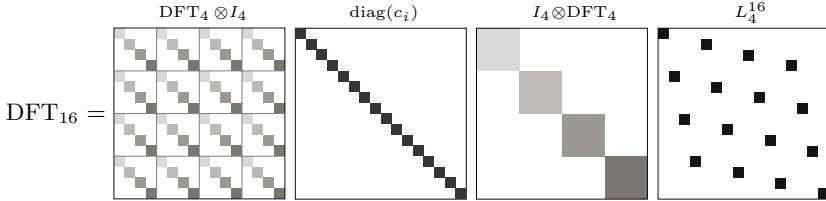
$$\text{MMM}_{m,k,n} \rightarrow (S_{k/k_b} \otimes \text{MMM}_{m,k_b,n}) \circ ((L_{k/k_b}^{mk/k_b} \otimes I_{k_b}) \times I_{kn}). \quad (3)$$

The three blocking rules we just described can actually be combined into a single rule with three degrees of freedom:

$$\begin{aligned} \text{MMM}_{m,k,n} \rightarrow & (I_{m/m_b} \otimes L_{m_b}^{mbn/n_b} \otimes I_{n_b}) \circ (\text{MMM}_{m/m_b, k/k_b, n/n_b} \otimes \text{MMM}_{m_b, k_b, n_b}) \\ & \circ ((I_{m/m_b} \otimes L_{k/k_b}^{mbk/k_b} \otimes I_{k_b}) \times (I_{k/k_b} \otimes L_{n/n_b}^{kbn/n_b} \otimes I_{n_b})). \end{aligned} \quad (4)$$

<sup>3</sup> In general, blocks may be of different sizes and thus a more general blocking rule can be formulated.

<sup>4</sup> As we will explain later, stages may be fused during the loop merging optimization, so three stages do not necessarily imply three different passes through the data. In this case, all stages would merge.



**Fig. 4.** Representation of the 4 stages of the Cooley-Tukey algorithm. Each frame corresponds to the matrix associated with one of the 4 operators from the equation (5), specialized with  $n = 16$  and  $m = 4$ . Only non-zero values are plotted. Shades of gray represent values that belong to the same tensor substructure.

The above rule captures the well-known mathematical fact that a multiplication of size  $(m, k, n)$  can be done by repeatedly using block multiplications of size  $(m_b, k_b, n_b)$ . Note that the coarse structure of a blocked matrix multiplication is itself a matrix multiplication. The fact that blocking can be captured as a tensor product was already observed by [29].

The second example we consider is the famous Cooley-Tukey fast Fourier transform (FFT) algorithm. It reduces the asymptotic cost of two-power sizes  $\text{DFT}_n$  from  $O(n^2)$  to  $O(n \log n)$ .

In this case the OL rule is equivalent to a matrix factorization and takes the same form as in [26] with the only difference that the matrix product is written as composition (of linear operators):

$$\text{DFT}_n \rightarrow (\text{DFT}_k \otimes I_m) \circ \text{diag}(c_i) \circ (I_k \otimes \text{DFT}_m) \circ L_k^n, \quad n = km. \quad (5)$$

Here,  $\text{diag}(c_i)$  is a diagonal matrix whose exact form is not of importance here [26].

As depicted in Fig. 4, this algorithm consists of 4 stages: The input vector is first permuted by  $L_m^n$ , then multiple  $\text{DFT}_m$  are applied to subvectors of the result. The result is scaled by  $\text{diag}(c_i)$  and finally again multiple  $\text{DFT}_k$  are computed, this time on strided subvectors.

Note that this algorithm requires the size  $n$  to be composite and leaves a degree of freedom in the integer factors<sup>5</sup>. Prime sizes require a different algorithm called Rader FFT [26].

Other examples of algorithms, written as OL rules, are presented in the end of the section.

**Base cases.** All recursive algorithms need to be terminated by base cases. In our case, these correspond to kernel sizes for which the computation is straightforward.

In the blocked multiplication case, the three dimensions can be reduced independently. Therefore, it is sufficient to know how to handle each one to be able to tackle

<sup>5</sup> When the algorithm is applied recursively, this degree of freedom is often called the *radix*.

any size. In the first two cases, the matrix multiplication degenerates into Kronecker products; in the last case, it simplifies into a scalar product:

$$\text{MMM}_{m,1,1} \rightarrow K_{m \times 1}, \quad (6)$$

$$\text{MMM}_{1,1,n} \rightarrow K_{1 \times n}, \quad (7)$$

$$\text{MMM}_{1,k,1} \rightarrow S_k. \quad (8)$$

Note that these three rules are degenerate special cases of the blocking rules (1)–(3).

Other bases cases could be used. For instance, Strassen's method to multiply  $2 \times 2$  matrices uses only 7 multiplications instead of 8 but requires more additions [29]:

$$\text{MMM}_{2,2,2} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \circ P_7 \circ \left( \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \end{bmatrix} \right) \quad (9)$$

Note that, due to the fact that blocking is a tensor product of two MMMs (4), the above base case can also be used in the structural part of the tensor, yielding a block Strassen algorithm of general sizes.

For the DFT of two-power sizes, the Cooley-Tukey FFT is sufficient together with a single base case, the DFT<sub>2</sub> nicknamed *butterfly*:

$$\text{DFT}_2 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (10)$$

**Algorithm space.** In this paper, we focus on implementing kernels of *fixed size*. In most applications that need to be fast, sizes are known at compilation time and therefore this generative approach is optimal because it removes all overhead. *General-size* code can also be generated from our domain specific language but it is mostly a different problem [30,31].

We say that a formula is *terminated* or *maximally expanded* if it does not contain any kernel symbols. Using different breakdown rules or different degrees of freedom, the same kernel can be expanded in different formulas. Each one of them represent a different algorithm to compute the kernel. The algorithm space can be explored using empiric search, strategies such as dynamic programming or machine learning algorithms [4].

For instance, we show here two different expansions of the same kernel,  $\text{MMM}_{2,2,2}$ . They are generated by applying rules (1)–(3) and base cases (6)–(8) in two different orders and simplifying:

$$\text{MMM}_{2,2,2} \rightarrow (\text{S}_2 \otimes \text{K}_{2 \times 2}) \circ (\text{L}_2^4 \times \text{I}_4)$$

$$\text{MMM}_{2,2,2} \rightarrow \circ((\text{K}_{2 \times 1} \otimes \text{S}_2) \otimes \text{K}_{1 \times 2}).$$

We now present additional kernels and algorithms:

**Circular convolution.** The circular convolution [26]  $\text{Conv}_n$  is an arity  $(2, 1)$  operator defined by

$$\text{Conv}_n : \mathbb{C}^n \times \mathbb{C}^n \rightarrow \mathbb{C}^n; (\mathbf{x}, \mathbf{y}) \mapsto \left( \sum_{j=0}^{n-1} x_i y_{i-j \bmod n} \right)_{0 \leq i < n}.$$

Circular convolution can be computed by going to the spectral domain using DFTs and inverse DFTs:

$$\text{Conv}_n \rightarrow \text{iDFT}_n \circ P_n \circ (\text{DFT}_n \times \text{DFT}_n). \quad (11)$$

**Sorting network.** A sorting network [32] sorts a vector of length  $n$  using a fixed (data-independent) algorithm. We define the ascending sort kernel  $\chi_n$  and the descending sort kernel  $\Theta_n$ :

$$\begin{aligned} \chi_n : \mathbb{N}^n &\rightarrow \mathbb{N}^n; (a_i)_{0 \leq i < n} \mapsto (a_{\sigma(i)})_{0 \leq i < n}, a_{\sigma(j)} \leq a_{\sigma(k)} \text{ for } j \leq k, \\ \Theta_n : \mathbb{N}^n &\rightarrow \mathbb{N}^n; (a_i)_{0 \leq i < n} \mapsto (a_{\sigma(i)})_{0 \leq i < n}, a_{\sigma(j)} \geq a_{\sigma(k)} \text{ for } j \leq k. \end{aligned}$$

The bitonic merge operator  $M_n$  merges an ascending sorted sequence  $(a_i)_{0 \leq i < n/2}$  and a descending sorted sequence  $(b_i)_{0 \leq i < n/2}$  into an ascending sorted sequence  $(c_i)_{0 \leq i < n}$ :

$$M_n : \mathbb{R}^n \rightarrow \mathbb{R}^n; (a_i)_{0 \leq i < n/2} \oplus_n (b_i)_{0 \leq i < n/2} \mapsto (c_i)_{0 \leq i < n} \text{ with } c_{i-1} \leq c_i.$$

The bitonic sorting network is described by mutually recursive breakdown rules:

$$\chi_n \rightarrow M_n \circ \oplus_n \circ (\chi_{n/2} \times \Theta_{n/2}) \circ \text{Split}_n \quad (12)$$

$$M_n \rightarrow (I_2 \otimes M_{n/2}) \circ (\Theta_2 \otimes I_{n/2}) \quad (13)$$

$$\Theta_n \rightarrow J_n \circ \chi_n \quad (14)$$

Finally, sorting the base case is given by

$$\chi_2 \rightarrow \oplus_2 \circ (\text{min}_1 \times \text{max}_1) \circ \text{Fork}_2 \quad (15)$$

**Viterbi decoding.** A Viterbi decoder computes the most likely convolutionally encoded message that was received over a noisy channel [33]. The computational bottleneck of the decoding is the *forward pass*  $\text{Vit}_{K,F}$  where  $F$  and  $K$  are the frame and constraint length of the message.

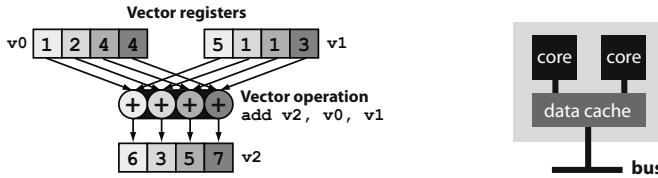
The algorithm structure in OL is:

$$\text{Vit}_{K,F} \rightarrow \prod_{i=1}^F \left( (I_{2^{K-2}} \otimes_j B_{F-i,j}) L_{2^{K-2}}^{2^{K-1}} \right). \quad (16)$$

$B_{F-i,j}$  is called the *viterbi butterfly* and is a base case. This formula features the indexed composition and the indexed tensor whose subscripts describe the number of the current iteration. More details are available in [34].

**Synthetic Aperture Radar (SAR).** The Polar formatting SAR operator  $\text{SAR}_{s,k}$  computes an image from radar pulses sent by a moving sensor [35]. Multiple measurement are synthesized into one aperture. There are many algorithms and methods to reconstruct the image, and we focus on an interpolation- and FFT-based variant, called polar formatting. In this paper we only consider two high-level parameters: a scenario (geometry)  $s = (m_1, n_1, m_2, n_2)$ , and an interpolator  $k = (k_r, k_a)$ , which parameterize the SAR operator,

$$\text{SAR}_{s,k} : \mathbb{C}^{m_1 \times n_1} \rightarrow \mathbb{C}^{m_2 \times n_2}.$$



**Fig. 5.** The SIMD vector paradigm and the shared memory paradigm

The polar format SAR algorithm is defined by the following breakdown rules:

$$\text{SAR}_{s,k} \rightarrow \text{DFT}_{m_2 \times n_2} \circ \text{2D-Intp}_k \quad (17)$$

$$\text{2D-Intp}_k \rightarrow (\text{Intp}_{k_a(i)}^{n_1 \rightarrow n_2} \otimes_i I_{m_2}) \circ (I_{n_1} \otimes_i \text{Intp}_{k_r(i)}^{m_1 \rightarrow m_2}) \quad (18)$$

$$\text{Intp}_{(w,k)}^{m \rightarrow n} \rightarrow G_w^{km \rightarrow n} \circ \text{iDFT}_{km} \circ Z^{m \rightarrow km} \circ \text{DFT}_m \quad (19)$$

Above,  $Z^{m \rightarrow km}$  describes zero-padding,  $G_w^{km \rightarrow n}$  non-uniform data gathering. Details are not essential for this paper and can be found in [36].

## 2.2 Abstracting Hardware Into Tags

Our goal is to automatically optimize algorithms by matching them to the target hardware. For portability, the actual computing *platforms* are abstracted behind simpler descriptions, the *hardware paradigms*. Paradigms capture essential properties of families of hardware. When more detailed properties are needed, one can always refer to the actual platform.

**Paradigms.** Paradigms are composable coarse structural descriptions of machines. They establish a set of properties that are common to certain classes of hardware. Actual platforms and instructions are abstracted behind this common layer. In Fig. 5 we show two examples: the single instruction multiple data (SIMD) vector instruction paradigm and the shared memory paradigm.

The *SIMD vector paradigm* models a class of processors with the following characteristics:

- The processor implements a vector register file and standard vector operations that operate pointwise: addition, multiplication and others. Using vector operations provide a significant speed-up over scalar operations.
- The most efficient data movement between memory and the vector register file is through aligned vector loads and stores. Unaligned memory accesses or subvector memory accesses are more expensive.
- The processor implements shuffle instructions that rearrange data inside a vector register (intra-register moves).

Most important examples of vector instruction sets are Intel's SSE family, the newly announced Intel extensions AVX and the Larrabee GPU native instructions, AMD's 3DNow! family, Motorola's AltiVec family including the IBM-developed variants for the Cell and Power processors.

The *shared memory paradigm* models a class of multiprocessor systems with the following characteristics:

- The system has multiple processors (or cores) that are all of the same type.
- The processors share a main, directly addressable memory.
- The system has a memory hierarchy and one cache line size is dominant.

Important processors modeled by the shared memory paradigm include Intel’s and AMD’s multicores and systems built with multiple of them. Non-uniform cache topologies are also supported as long as there is a shared memory abstraction.

Paradigms can be hierarchically composed. For instance, in single precision floating-point mode, an Intel Core2 Duo processor is characterized as two-processor shared memory system (cache line is 64 bytes). Both CPUs are 4-way SIMD vector units.

Besides these two paradigms, our framework experimentally<sup>6</sup> supports the following other paradigms: 1) distributed memory through message passing, 2) hardware streaming for FPGA design, 3) software streaming through multibuffering, 4) hardware-software partitioning, 5) general purpose programming on graphics card, and 6) adaptive dynamic voltage and frequency scaling.

**Platforms.** To ensure best cross-platform portability we capture most of the performance-critical structural information at the paradigm level, and avoid utilizing the actual platform information within the algorithm manipulation rules. Each generic operation required by the paradigms has to be implemented as efficiently as possible on the given platforms.

For instance, we required any hardware covered by the vector paradigm to provide some instructions for data reorganizations within a vector (called vector *shuffles*). However, the actual capabilities of the vector units depend vastly on the vector instruction family and worse, on the version of the family. Therefore, it is not portable enough to describe algorithms using directly these instructions and we choose to abstract shuffles at the paradigm level. By doing that, each platform disposes of its own custom implementation of the same algorithm. Note that the task of deriving efficient platform-specific implementations of the shuffles required by the vector paradigm can be also fully automated, straight from the instructions specification [37].

The actual platform information is also used further down in our program generation tool chain. In particular, our compiler translating the domain-specific language into C code relies on it.

**Paradigm tags.** We denote paradigms using *tags* which are a name plus some parameters. For instance, we describe a shared memory 2-core system with cache line length of 16 `float` (64 bytes) by “`smp(2, 16)`” and a 4-way `float` vector unit by “`vec(4)`”. Tags can be parameterized by symbolic constants: we will use `smp( $p, \mu$ )`, and `vec( $\nu$ )` throughout the paper.

Tags can be concatenated to describe multiple aspects of a target hardware. For instance, the 2-core shared memory machine above where each core is a 4-way vector unit will be described by “`smp(2, 16), vec(4)`”.

---

<sup>6</sup> For the experimental paradigms, we can only generate a limited subset of kernels.

When compiling the OL formulas, the paradigms tags are replaced by the actual platform. For instance, the shared memory tag  $\text{smp}(2, 16)$  leads to multi-threaded code using OpenMP or pthreads, the vector tag  $\text{vec}(4)$  creates either Intel's SSE or Cell SPU code.

### 2.3 Common Abstraction: Tagged Operator Language

In this section we show how to build a space of algorithms optimized for a target platform by introducing the paradigm tags into the operator language. The main idea is that the tags contain meta-information that enables the system to transform algorithms in an architecture-conscious way. These transformations are performed by adding paradigm-specific rewriting rules on top of the algorithm breakdown rules. The joint rule set spans a space of different formulas for a given tagged kernel where all formulas are proven to have good implementations on the target paradigm and thus platform.

We first discuss the components of the system and then show their interaction with a few illustrative examples.

**Tagged OL formula.** A formula  $A$  tagged by a tag  $t$ , denoted

$$\underbrace{A}_{t}$$

expresses the fact that  $A$  will be structurally optimized for  $t$  which can either be a paradigm or an architecture. We have multiple types of tagged formulas: 1) problem specifications (tagged kernels), 2) fully expanded expressions (terminated formulas), 3) partially expanded expressions, and 4) base cases. These tagged formulas serve various purposes during the algorithm construction process and are crucial to the underlying rewriting process.

The input to our algorithm construction is a *problem specification* given by a tagged kernel. For instance,

$$\underbrace{\text{MMM}_{m,k,n}}_{\text{smp}(p,\mu)}$$

asks the system to generate an OL formula for a MMM that is structurally optimized for a shared memory machine with  $p$  processors and cache line length  $\mu$ . Note that, while in the paper we always use variables, the actual input is a fixed-size kernel, and thus all variables have known values.

Any OL expression is a formula. The rewriting process continually changes formulas. A *terminated formula* does not have any kernel left, and no further rule is applicable. Any terminated formula that was derived from a tagged kernel is structurally optimized for the tag and is guaranteed to map well to the target hardware. Any OL formula that still contains kernels is only *partially expanded* and needs further rewriting to obtain an optimized OL formula.

A *base case* is a tagged formula that cannot be further broken down by any breakdown rule and the system has a paradigm-specific (or platform-specific) implementation template for the formula. The goal is to have as few base cases per paradigm as possible, but to support enough cases to be able to implement kernels based on them. Any

terminated formula is built from base cases and OL operations that are compatible with the target paradigm.

**Rewriting system.** At the heart of the algorithm construction process is a rewriting system that starts with a tagged kernel and attempts to rewrite it into a fully expanded (terminated) tagged OL formula. The system uses pattern matching against the left-hand side of rewrite rules like (1) to find subexpressions within OL formulas and replaces them with equivalent new expressions derived from the right-hand side of the rule. It selects one of the applicable rules and chooses a degree of freedom if the rule has one. The system keeps track of the choices to be able to backtrack and pick different rules or parameters in case the current choice is not leading to a terminated formula. The process is very similar to Prolog's computation of proofs.

The system uses a combined rule set that contains algorithm *breakdown rules and base cases* (1)–(19), *paradigm base cases* (20)–(24), and *paradigm-specific manipulation rules* (25)–(33). The breakdown rules are described in Section 2.1. In the remainder of the section we will describe the remaining two rule classes in more detail.

**Base cases.** Tagged base cases are OL formulas that have a known good implementation on every platforms covered by the paradigm. Every formula built only from these base cases is guaranteed to perform well.

We now discuss the base cases for the *shared memory* paradigm, expressed by the tag  $\text{smp}(p, \mu)$ . The tag states that our target system has  $p$  processors and cache length of  $\mu$ . This information is used to obtain load balanced, false-sharing free base cases [5]. OL operations in base cases are also tagged to mark the base cases as fully expanded. In the shared memory case we introduce three new tagged operators,  $\otimes_{\parallel}$ ,  $\bar{\otimes}$ ,  $\oplus_{\parallel}$ , which have the same mathematical meaning as their un-tagged counterparts. The following linear operators are shared memory base cases:

$$I_p \otimes_{\parallel} A_{m\mu \rightarrow n\mu}, \quad \bigoplus_{i=0}^{p-1} A_{m\mu \rightarrow n\mu}^i, \quad M \bar{\otimes} I_\mu \text{ with } M \text{ a permutation matrix} \quad (20)$$

The first two expression encodes embarrassingly parallel, load-balanced computations that distribute the data so that no cache line is shared by multiple processors. The third expression encodes data transfer that occurs on a cache line granularity, also avoiding false sharing. The following non-linear arity (2,1) operators are shared memory base cases. They generalize the idea of  $I_p \otimes_{\parallel} A_{m\mu \rightarrow n\mu}$  in (20),

$$P_p \otimes_{\parallel} A_{k\mu \times m\mu \rightarrow n\mu}, \quad K_{q \times r} \otimes_{\parallel} A_{k\mu \times m\mu \rightarrow n\mu} \text{ where } qr = p. \quad (21)$$

Building on these base cases we can build fully optimized (i.e., terminated) tagged OL formulas using OL operations. For instance, any formula  $A \circ B$  where  $A$  and  $B$  are fully optimized is also fully optimized. Similarly,  $\times$  allows to construct higher-arity operators that are still terminated. For instance,

$$(M \bar{\otimes} I_\mu) \times (N \bar{\otimes} I_\mu), \quad M, N \text{ are permutation matrices} \quad (22)$$

produces terminated formulas. Not all OL operations can be used to build larger terminated OL formulas from smaller ones. For instance, if  $A$  is an arity (1,1) shared memory base case, then in general  $A \otimes I_k$  is no shared memory base case.

Next we discuss the base cases for the *SIMD vector* paradigm. The tag for the SIMD vectorization paradigm is  $\text{vec}(\nu)$ , and implies  $\nu$ -way vector units which require all memory access operations to be naturally aligned vector loads or stores. Similar to the shared memory base cases, we introduce two special markers to denote base cases. The operator  $\hat{\otimes}$  denotes a basic block that can be implemented using solely vector arithmetic and vector memory access. Furthermore, the vector paradigm requires a set of base cases that have architecture-dependent implementations but can be implemented well on all architectures described by the SIMD vector paradigm. The exact implementation of these base cases is part of the architecture description. We limit our discussion to the Intel SSE instruction set, and mark such base with the following symbol,

$$\underbrace{A}_{\text{base(sse)}} ,$$

and imply the vector length  $\nu = 4$ . Other architectures or vector lengths would require similarly marked base cases with implementations stored in the architecture definition data base.

The base case library contains the implementation descriptions for the following linear arity (1,1) operators:

$$A_{m \rightarrow n} \hat{\otimes} I_\nu, \quad \underbrace{L_\nu^{\nu^2}}_{\text{base(sse)}}, \quad \underbrace{\text{diag}(c_i)}_{\text{base(sse)}}. \quad (23)$$

The formula  $A_{m \rightarrow n} \hat{\otimes} I_\nu$  is of special importance, as it can be implemented solely using vector instructions, independently of  $A_{m \rightarrow n}$  [38]. The generalization of the  $A_{m \rightarrow n} \hat{\otimes} I_\nu$  to arity (2,1) is given by

$$A_{k \times m \rightarrow n} \hat{\otimes} P_\nu. \quad (24)$$

Similar to the shared memory base cases, some OL operations allow to construct larger fully optimized (terminated) OL formulas from smaller ones. For instance, if  $A$  and  $B$  are terminated, then  $A \circ B$  is terminated. In addition, if  $A$  is terminated, then  $I_n \otimes A$  is terminated as well.

**Paradigm-specific rewrite rules.** Our rewriting system employs paradigm-specific rewriting rules to extract paradigm-specific base cases and ultimately obtain a fully optimized (terminated) OL formula. These rules are annotated mathematical identities, which allow for proving correctness of formula transformations. The linear arity (1,1) rules are derived from matrix identities [26], and non-linear and higher arity identities are based on generalizations of these identities.

Table 2 summarizes our shared memory rewrite rules. Using (25)–(29), the system transforms formulas into OL expressions that are built from the base cases defined in (20)–(21). Some of the arity (1,1) identities are taken from [5].

Table 3 summarizes SIMD vectorization-specific rewriting rules. Some of the arity (1,1) identities can be found in [6]. These rules translate unterminated OL formulas into formulas built from SIMD base cases.

**Examples.** We now show the result of the rewriting process for shared memory and SIMD vectorization, for DFT and MMM. We specify a  $\text{DFT}_{mn}$  kernel tagged with

**Table 2.** OL rewriting rules for SMP parallelization

$$\underbrace{(I_k \otimes L_n^{mn})}_{\text{smp}(p,\mu)} \circ \underbrace{L_{km}^{kmn}}_{\text{smp}(p,\mu)} \rightarrow (L_k^{kn} \otimes I_{m/\mu}) \bar{\otimes} I_\mu \quad (25)$$

$$\underbrace{L_n^{kmn}}_{\text{smp}(p,\mu)} \circ \underbrace{(I_k \otimes L_m^{mn})}_{\text{smp}(p,\mu)} \rightarrow (L_n^{kn} \otimes I_{m/\mu}) \bar{\otimes} I_\mu \quad (26)$$

$$\underbrace{A_{k \times m \rightarrow n} \otimes K_{1 \times p}}_{\text{smp}(p,\mu)} \rightarrow \underbrace{L_n^{pn}}_{\text{smp}(p,\mu)} \circ (K_{1 \times p} \otimes_{\parallel} A_{k \times m \rightarrow n}) \circ \underbrace{(I_k \times L_p^{pm})}_{\text{smp}(p,\mu)} \quad (27)$$

$$\underbrace{(A \times B)}_{\text{smp}(p,\mu)} \circ \underbrace{(C \times D)}_{\text{smp}(p,\mu)} \rightarrow \underbrace{(A \circ C)}_{\text{smp}(p,\mu)} \times \underbrace{(B \circ D)}_{\text{smp}(p,\mu)} \quad (\text{if arities are compatible}) \quad (28)$$

$$\underbrace{A \circ B}_{\text{smp}(p,\mu)} \rightarrow \underbrace{A}_{\text{smp}(p,\mu)} \circ \underbrace{B}_{\text{smp}(p,\mu)} \quad (29)$$

**Table 3.** OL vectorization rules

$$\underbrace{(A_{n \rightarrow n} \otimes I_m)}_{\text{vec}(\nu)} \rightarrow (A_{n \rightarrow n} \otimes I_{m/\nu}) \hat{\otimes} I_\nu \quad (30)$$

$$\underbrace{(I_m \otimes A_{n \rightarrow n}) \circ L_m^{mn}}_{\text{vec}(\nu)} \rightarrow \left( I_{m/\nu} \otimes \underbrace{L_\nu^{n\nu}}_{\text{vec}(\nu)} \circ (A_{n \rightarrow n} \hat{\otimes} I_\nu) \right) \circ (L_{m/\nu}^{mn/\nu} \hat{\otimes} I_\nu) \quad (31)$$

$$\underbrace{L_n^{n\nu}}_{\text{vec}(\nu)} \rightarrow \left( I_{n/\nu} \otimes \underbrace{L_\nu^{\nu^2}}_{\text{base(sse)}} \right) \circ (L_{n/\nu}^n \hat{\otimes} I_\nu) \quad (32)$$

$$\underbrace{A_{k \times m \rightarrow n} \otimes K_{1 \times \nu}}_{\text{vec}(\nu)} \rightarrow (A_{k \times m \rightarrow n} \otimes P_\nu) \circ (\text{dup}_k^\nu \times I_{m\nu}) \quad (33)$$

the SIMD vector tag  $\text{vec}(\nu)$  to instruct the system to produce a SIMD vectorized fully expanded OL formula;  $m$  and  $n$  are fixed numbers. The system applies the breakdown rules (5) and together with the SIMD vector-specific rewriting rules (30)–(33). The rewriting process yields

$$\begin{aligned} \underbrace{\text{DFT}_{mn}}_{\text{vec}(\nu)} &\rightarrow ((\text{DFT}_m \otimes I_{n/\nu}) \hat{\otimes} I_\nu) \circ \underbrace{\text{diag}(c_i)}_{\text{base(sse)}} \\ &\circ (I_{m/\nu} \otimes (I_{n/\nu} \otimes \underbrace{L_\nu^{\nu^2}}_{\text{base(sse)}}) \circ (L_{n/\nu}^n \hat{\otimes} I_\nu) \circ (\text{DFT}_n \hat{\otimes} I_\nu)) \circ (L_{m/\nu}^{mn/\nu} \hat{\otimes} I_\nu), \end{aligned}$$

which will be terminated independently of how  $\text{DFT}_m$  and  $\text{DFT}_n$  are further expanded by the rewriting system. A detailed earlier (SPL-based) version of the rewriting process can be found in [6].

Similarly, we tag  $\text{DFT}_{mn}$  with  $\text{smp}(p,\mu)$  to instruct the rewriting system to produce a DFT OL formula that is fully optimized for the shared memory paradigm. The

algorithm breakdown rules (5) are applied together with the paradigm-specific rewriting rules, (25)–(29). The rewriting process yields

$$\begin{aligned} \underbrace{\text{DFT}_{mn}}_{\text{smp}(p,\mu)} &\rightarrow ((L_m^{mp} \otimes I_{n/(p\mu)}) \bar{\otimes} I_\mu) \circ (I_p \otimes_{\parallel} (\text{DFT}_m \otimes I_{n/p})) \circ ((L_p^{mp} \otimes I_{n/(p\mu)}) \bar{\otimes} I_\mu) \\ &\circ \left( \bigoplus_{i=0}^{p-1} \parallel D_{m,n}^i \right) \circ (I_p \otimes_{\parallel} (I_{m/p} \otimes \text{DFT}_n)) \circ (I_p \otimes_{\parallel} L_{m/p}^{mn/p}) \circ ((L_p^{pn} \otimes I_{m/p\mu}) \bar{\otimes} I_\mu). \end{aligned}$$

Again, the above formula is terminated independently of how  $\text{DFT}_m$  and  $\text{DFT}_n$  are further expanded by the rewriting system. A detailed earlier (SPL-based) version of the rewriting process can be found in [5].

As next example we show the shared memory optimization of  $\text{MMM}$ . The kernel specification is given by  $\text{MMM}_{m,k,n}$  tagged by  $\text{smp}(p,\mu)$ . The algorithm breakdown rules is (4) and the shared memory-specific rewriting rules are again (25)–(29). The rewriting process finds the following result

$$\underbrace{\text{MMM}_{m,k,n}}_{\text{smp}(p,\mu)} \rightarrow K_{p \times 1} \otimes_{\parallel} \text{MMM}_{m/p,k,n},$$

which cuts the first matrix horizontally and distributes slices among different cores. The rewriting process also discovers automatically that it could distribute jobs by splitting the second matrix vertically:

$$\begin{aligned} \underbrace{\text{MMM}_{m,k,n}}_{\text{smp}(p,\mu)} &\rightarrow ((L_m^{mp} \otimes I_{n/(p\mu)}) \bar{\otimes} I_\mu) \\ &\circ (K_{1 \times p} \otimes_{\parallel} \text{MMM}_{m,k,n/p}) \circ ((I_{km/\mu} \bar{\otimes} I_\mu) \times ((L_p^{kp} \otimes I_{n/(p\mu)}) \bar{\otimes} I_\mu)). \end{aligned}$$

Our final example is the  $\nu$ -way vectorization of a  $\text{MMM}_{m,k,n}$ . Using the matrix blocking rule (4) and the vector rules (30)–(33), the rewriting system yields

$$\underbrace{\text{MMM}_{m,k,n}}_{\text{vec}(\nu)} \rightarrow (\text{MMM}_{m,k,n/\nu} \hat{\otimes} P_\nu) \circ (\text{dup}_{mk}^\nu \times I_{kn}).$$

### 3 Generating Programs for OL Formulas

In the last section we explained how Spiral constructs an algorithm that solves the specified problem on the specified hardware. The output is an OL formula that encodes the data flow, data layout, and implementation choices. In this section we describe how Spiral compiles this OL formula to a target program, usually C with library calls or pragmas.

**Formula rewriting.** An OL formula encodes the data flow of an algorithm and the data layout. It does not make explicit any control flow and loops. The intermediate

representation  $\Sigma$ -OL (which extends  $\Sigma$ -SPL [39] to OL and is beyond the scope of this paper) makes loops explicit while still being a declarative mathematical domain-specific language. Rewriting of  $\Sigma$ -OL formulas allows in particular to fuse data permutations inserted by paradigm-specific rewriting with neighboring looped computational kernels. The result is domain-specific loop merging, that is beyond the capabilities of traditional compilers but essential for achieving high performance.

$\Sigma$ -OL formulas are still only parameterized by paradigms, not actual architectures. This provides portability of domain-specific loop optimizations within a paradigm, as rewriting rules are at least reused for all architectures of a paradigm.

**$\Sigma$ -OL compiler.** The final  $\Sigma$ -OL expression is a declarative representation of a loop-based program that implements the algorithm on the chosen hardware. The  $\Sigma$ -OL compiler (an extension of the SPL compiler [3] and the  $\Sigma$ -SPL compiler [39]) translates this expression into an internal code representation (resembling a C program), using rules such as the ones in Table 4. The resulting code is further optimized using standard compiler optimizations like unrolling, common subexpression elimination, strength reduction, and constant propagation.

**Base case library.** During compilation, paradigm-specific constructs are fetched from the platform-specific base case library. After inclusion in the algorithm, a platform-specific strength reduction pass is performed. Details on the base case library can be found in [38,5,37].

**Unparser.** In a final step, the internal code representation is outputted as a C program with library calls and macros. The platform description carries the information of how to unparse special instructions and how to invoke the requisite libraries (for instance pthreads). Code can be easily retargeted to different libraries by simply using a different unparser. For instance, an OpenMP parallel program and a pthreads parallel program have the same internal representation and the target threading facility is only committed to when unparsing the program.

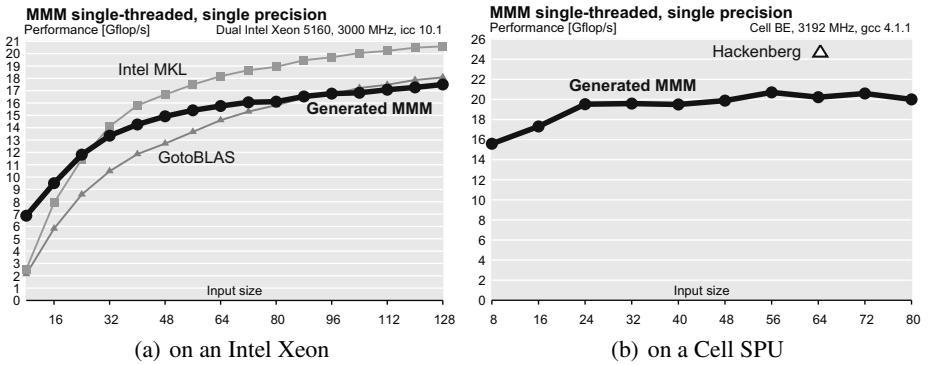
## 4 Experimental Results

We implement the language and rewriting system inside GAP [40] and evaluate the code generator on the MMM, DFT, SAR imaging and Viterbi decoding problems. In each case, we generate optimized code and compare it to the best available implementations, which are often optimized in assembly. As we detail below, we achieve comparable performance on all scenarios. In some cases, human developers cannot spend the time to support all functionalities on all platforms. In these cases, having a code generator based approach is a huge win.

**MMM.** We evaluate our MMM code on two platforms: an Intel Xeon (Fig. 6(a)) and an IBM Cell BE (Fig. 6(b)). On the Intel platform, we compare the generated vectorized single threaded code to the Intel Math Kernel Library (MKL) 10.0 and the Goto BLAS 1.26 which are both hand-optimized in assembly [41,42]. On the Cell, we compare the generated code that runs on a single SPU to the assembly code provided by D. Hackenberg [43]. Due to restrictions on the local store, [43] only provides code for sizes that are multiple of 64.

**Table 4.** Translating operator formulas to code.  $\mathbf{x}$  and  $\mathbf{y}$  denote the input and  $\mathbf{r}$  and  $\mathbf{s}$  the output vectors. The subscripts of  $\mathbf{A}$  and  $\mathbf{B}$  specify the signatures of the operators when they are relevant. We use Matlab-like notation:  $\mathbf{x}[b:s:e]$  denotes the subvector of  $\mathbf{x}$  starting at  $b$ , ending at  $e$  and extracted at stride  $s$ .

operator formula	code
<i>operators</i>	
$\mathbf{r} = \mathbf{P}_n(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;n; i++)     r[i] = x[i]*y[i];</pre>
$\mathbf{r} = \mathbf{S}_n(\mathbf{x}, \mathbf{y})$	<pre>r=0; for (i=0; i&lt;n; i++)     r += x[i]*y[i];</pre>
$\mathbf{r} = \mathbf{K}_{m \times n}(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;m; i++)     for (j=0; j&lt;n; j++)         r[i*m+j] = x[i]*y[j];</pre>
$\mathbf{r} = \mathbf{L}_m^{mn}(\mathbf{x})$	<pre>for (i=0; i&lt;m; i++)     for (j=0; j&lt;n; j++)         r[i+m*j] = x[n*i+j];</pre>
<i>higher-order operators</i>	
$\mathbf{r} = (\mathbf{A} \circ \mathbf{B})(\mathbf{x})$	<pre>t = B(x); r = A(t);</pre>
$(\mathbf{r}, \mathbf{s}) = (\mathbf{A} \times \mathbf{B})(\mathbf{x}, \mathbf{y})$	<pre>r = A(x); s = B(y);</pre>
$\mathbf{r} = (I_m \otimes \mathbf{A}_n)(\mathbf{x})$	<pre>for (i=0; i&lt;m; i++)     r[in:1:(i+1)n-1] = A(x[in:1:(i+1)n-1]);</pre>
$\mathbf{r} = (\mathbf{A}_m \otimes I_n)(\mathbf{x})$	<pre>for (i=0; i&lt;m; i++)     r[i:n:i+n(m-1)] = A(x[i:n:i+n(m-1)]);</pre>
$\mathbf{r} = (\mathbf{P}_p \otimes \mathbf{A}_{m \times n \rightarrow k})(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;p; i++)     r[i*k:1:(i+1)k-1] =         A(x[i*m:1:(i+1)m-1],             r[i*n:1:(i+1)n-1]);</pre>
$\mathbf{r} = (\mathbf{A}_{m \times n \rightarrow k} \otimes \mathbf{P}_p)(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;p; i++)     r[i:k:i+k(p-1)] =         A(x[i:m:i+m(p-1)])             r[i:n:i+n(p-1)];</pre>
$\mathbf{r} = (\mathbf{K}_{p \times q} \otimes \mathbf{A}_{m \times n \rightarrow k})(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;p; i++)     for (j=0; j&lt;q; j++)         r[(i*p+j)*k:1:(i*p+j+1)*k-1] =             A(x[i*m:1:(i+1)m-1],                 r[j*n:1:(j+1)n-1]);</pre>
$\mathbf{r} = (\mathbf{A}_{m \times n \rightarrow k} \otimes \mathbf{K}_{p \times q})(\mathbf{x}, \mathbf{y})$	<pre>for (i=0; i&lt;p; i++)     for (j=0; j&lt;q; j++)         r[i*p+j:p*q:i*p+j+p*q*(k-1)] =             A(x[i:p:i+p(m-1)],                 r[j:q:j+q(n-1))];</pre>



**Fig. 6.** Matrix Multiplication on two different platforms. All implementations are vectorized and single-threaded.

On both platforms, the generated vectorized code achieves 70% of the theoretical peak performance and is comparable to or slightly slower than hand-optimized code. The Cell platform offers a demonstration of the versatility of a code generator: due to the particularity of this architecture, matrix multiplications of sizes not multiple of 64 are simply not provided by high-performance experts; a code generator allows us to support any sizes that the user may be interested in.

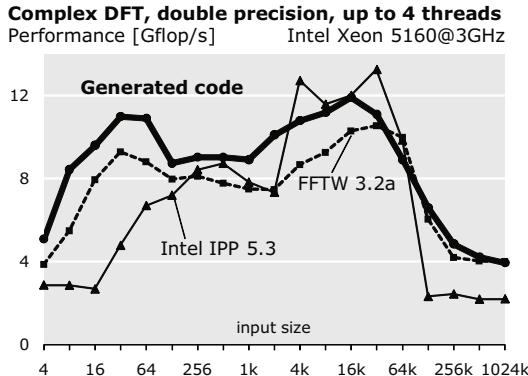
**DFT.** We evaluate our generated kernels for the DFT against the Intel Performance Primitives (IPP) and the FFTW library [1,19]. While IPP is optimized in assembly, FFTW’s approach shows some similarities with ours since small sizes are automatically generated.

Fig. 7 shows that our performance is comparable or better than both libraries for a standard Intel platform. All libraries are vectorized and multi-threaded.

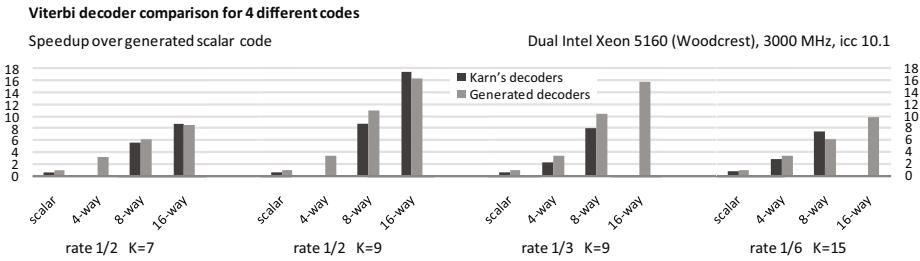
**SAR.** We evaluate the generated SAR code on an Intel Core2 Quad server (QX9650). We observe a steady performance of 34 Gigaflops/sec for 16 and 100 Megapixel SAR images with runtimes of 0.6 and 4.45 seconds respectively. This performance numbers are comparable with [44] who developed hand-optimized assembly code for a Cell Blade.

**Viterbi decoding.** We evaluate our generated decoders against Karn’s hand-written decoders [45]. Karn’s forward error correction software supports four different common convolutional codes and four different vector lengths. For each pair of code and vector length, the forward pass is written and optimized in assembly. Due to the amount of work required, some combinations are not provided by the library.

In Fig. 8, we generated an optimized decoder for each possible combination and compared their performances to Karn’s hand-optimized implementations. Analysis of the plot shows that our generated decoders have roughly the same performance than the hand-optimized assembly code from [45]. However, due to the code generator approach, we cover the full cross-product of codes and architectures. In particular, note that we are not limited to these four codes. An online interface is provided to generate decoders on demand [34,46].



**Fig. 7.** Discrete Fourier transform (DFT) performance comparison between our generated code, Intel Performance Primitives (IPP) and FFTW. All implementations are vectorized and multi-threaded.



**Fig. 8.** Performance comparison between generated Viterbi decoders and hand-optimized decoders from Karn's Forward Error Correction (FEC) library. For each of the 4 different convolutional codes supported by the FEC library, Karn provides up to 4 implementations that vary in vector length. All performances are normalized to the scalar performance of the generated decoder.

## 5 Conclusion

In this paper we presented OL, a framework to automatically generate high-performance implementations for a set of important kernels. Our approach aims at fully automating the implementation process, from algorithm selection and manipulation down to compiler-domain optimizations. It builds on and extends the library generation system for linear transforms, Spiral, to support multi-input/output and nonlinear kernels. The performance of the implementations our system produces rivals expertly hand-tuned implementations for the considered kernels. The approach is currently restricted to kernels with data-independent control flow, and we currently support limited functionality from a broad selection of domains. We plan to extend the approach to more kernels and domains in future work.

## References

1. Intel: Integrated Performance Primitives 5.3, User Guide
2. Chellappa, S., Franchetti, F., Püschel, M.: How to write fast numerical code: A small introduction. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 196–259. Springer, Heidelberg (2008)
3. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A language and compiler for DSP algorithms. In: Proc. Programming Language Design and Implementation (PLDI), pp. 298–308 (2001)
4. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 232–275 (2005)
5. Franchetti, F., Voronenko, Y., Püschel, M.: FFT program generation for shared memory: SMP and multicore. In: Proc. Supercomputing (2006)
6. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: Daydé, M., Palma, J.M.L.M., Coutinho, Á.L.G.A., Pacitti, E., Lopes, J.C. (eds.) VECPAR 2006. LNCS, vol. 4395, pp. 363–377. Springer, Heidelberg (2007)
7. GPCE: ACM conference on generative programming and component engineering
8. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
9. Batory, D., Johnson, C., MacDonald, B., von Heeder, D.: Achieving extensibility through product-lines and domain-specific languages: A case study. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2), 191–214 (2002)
10. Batory, D., Lopez-Herrejon, R., Martin, J.P.: Generating product-lines of product-families. In: Proc. Automated Software Engineering Conference (ASE) (2002)
11. Smith, D.R.: Mechanizing the development of software. In: Broy, M. (ed.) Calculational System Design, Proc. of the International Summer School Marktoberdorf. NATO ASI Series. IOS Press, Amsterdam (1999); Kestrel Institute Technical Report KES.U.99.1
12. Gough, K.J.: Little language processing, an alternative to courses on compiler construction. SIGCSE Bulletin 13(3), 31–34 (1981)
13. Bentley, J.: Programming pearls: little languages. Communications of the ACM 29(8), 711–721 (1986)
14. Hudak, P.: Domain specific languages. Available from author on request (1997)
15. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 51–72. Springer, Heidelberg (2004)
16. Taha, W.: Domain-specific languages. In: Proceedings of International Conference on Computer Engineering and Systems (ICCES 2008) (2008)
17. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software (ATLAS). In: Proc. Supercomputing (1998), [math-atlas.sourceforge.net](http://math-atlas.sourceforge.net)
18. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int'l. J. High Performance Computing Applications 18(1) (2004)
19. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 216–231 (2005)
20. Frigo, M.: A fast Fourier transform compiler. In: Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), pp. 169–180 (1999)

21. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishnamoorthy, S., Krishnan, S., Lam, C.C., Lu, Q., Noojien, M., Pitzer, R.M., Ramanujam, J., Sadayappan, P., Sibiryakov, A.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. of the IEEE, special issue on Program Generation, Optimization, and Adaptation* 93(2) (2005)
22. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Orti, E., van de Geijn, R.: The science of deriving dense linear algebra algorithms. *TOMS* 31(1), 1–26 (2005)
23. Dershowitz, N., Plaisted, D.A.: Rewriting. In: Robinson, A., Voronkov, A. (eds.) *Handbook of Automated Reasoning*, vol. 1, pp. 535–610. Elsevier, Amsterdam (2001)
24. Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog*, 2nd edn. John Wiley & Sons Inc., Chichester (1995)
25. Field, A.J., Harrison, P.G.: *Functional Programming*. Addison-Wesley, Reading (1988)
26. Van Loan, C.: *Computational Framework of the Fast Fourier Transform*. SIAM, Philadelphia (1992)
27. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. *Parallel Computing* 27(1–2), 3–35 (2001)
28. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: A comparison of empirical and model-driven optimization. *Proc. of the IEEE, special issue on Program Generation, Optimization, and Adaptation* 93(2) (2005)
29. Johnson, R.W., Huang, C.H., Johnson, J.R.: Multilinear algebra and parallel programming. In: *Supercomputing 1990: Proceedings of the 1990 conference on Supercomputing*, pp. 20–31. IEEE Computer Society Press, Los Alamitos (1990)
30. Voronenko, Y.: Library Generation for Linear Transforms. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2008)
31. Voronenko, Y., de Mesmay, F., Püschel, M.: Computer generation of general size linear transform libraries. In: *Intl. Symposium on Code Generation and Optimization, CGO* (2009)
32. Batcher, K.: Sorting networks and their applications. In: *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, pp. 307–314 (1968)
33. Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13(2), 260–269 (1967)
34. de Mesmay, F., Chellappa, S., Franchetti, F., Püschel, M.: Computer generation of efficient software Viterbi decoders: submitted for publication
35. Carrara, W.G., Goodman, R.S., Majewski, R.M.: *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Artech House (1995)
36. McFarlin, D., Franchetti, F., Moura, J.M.F., Püschel, M.: High performance synthetic aperture radar image formation on commodity architectures. In: *SPIE Conference on Defense, Security, and Sensing* (2009)
37. Franchetti, F., Voronenko, Y., Milder, P.A., Chellappa, S., Telgarsky, M., Shen, H., D'Alberto, P., de Mesmay, F., Hoe, J.C., Moura, J.M.F., Püschel, M.: Domain-specific library generation for parallel software and hardware platforms. In: *NSF Next Generation Software Program workshop, NSFNGS* (2008)
38. Franchetti, F., Püschel, M.: Short vector code generation for the discrete Fourier transform. In: *Proc. IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS)*, pp. 58–67 (2003)
39. Franchetti, F., Voronenko, Y., Püschel, M.: Loop merging for signal transforms. In: *Proc. Programming Language Design and Implementation (PLDI)*, pp. 315–326 (2005)
40. The GAP Team University of St. Andrews, Scotland: *GAP—Groups, Algorithms, and Programming* (1997), <http://www-gap.dcs.st-and.ac.uk/~gap/>
41. Intel: *Math Kernel Library 10.0, Reference Manual*
42. Goto, K.: *GotoBLAS 1.26* (2008),  
<http://www.tacc.utexas.edu/resources/software/#blas>

43. Hackenberg, D.: Fast matrix multiplication on Cell (SMP) systems,  
<http://www.tu-dresden.de/zih/cell/matmul>
44. Rudin, J.A.: Implementation of polar format SAR image formation on the IBM Cell Broadband Engine. In: Proc. High Performance Embedded Computing (HPEC) (2007)
45. Karn, P.: FEC library version 3.0.1 (August 2007),  
<http://www.ka9q.net/code/fec/>
46. de Mesmay, F.: Online generator for Viterbi decoders (2008),  
<http://www.spiral.net/software/viterbi.html>

# Author Index

- Agron, Jason 20, 262  
Allwein, Gerard 20
- Charles, Julien 187  
Cochran, Dermot 187  
Consel, Charles 78  
Cook, William R. 58
- Danvy, Olivier 1  
de Mesmay, Frédéric 385  
Drey, Zoé 78  
Du Bois, André Rauber 170
- Ebert, Jürgen 148  
Echevarria, Marcos 170  
Erwig, Martin 310, 335
- Fairmichael, Fintan 187  
Falcon, Jose 58  
Franchetti, Franz 385
- Gill, Andy 285  
Gokhale, Aniruddha 100  
Gray, Jeff 125  
Grigore, Radu 187
- Harrison, William L. 20  
Holub, Viliam 187  
Hudak, Paul 211
- Ionescu, Cezar 236
- Janota, Mikoláš 187  
Jansson, Patrik 236
- Kimmell, Garrin 20  
Kiniry, Joseph R. 187  
Kiselyov, Oleg 360
- Larsen, Ken Friis 45  
Lincke, Daniel 236
- McFarlin, Daniel 385  
Mercadal, Julien 78  
Mernik, Marjan 125
- Procter, Adam M. 20  
Püschel, Markus 385
- Shan, Chung-chieh 1, 360
- Tambe, Sumant 100
- Voellmy, Andreas 211
- Walkingshaw, Eric 310, 335  
Walter, Tobias 148  
Wu, Hui 125
- Zalewski, Marcin 236  
Zerny, Ian 1