Basic Research in Computer Science

BRICS NS-99-1 O. Danvy (ed.): PEPM '99 Proceedings

ACM SIGPLAN Workshop on

Partial Evaluation and Semantics-Based Program Manipulation

PEPM '99

San Antonio, Texas, USA, January 22–23, 1999

Olivier Danvy (editor)

BRICS Notes Series **NS-99-1**

ISSN 0909-3206 **January 1999**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/99/1/`

Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)

Olivier Danvy (editor)

January 1999

The PEPM'99 workshop is bringing together researchers working in the areas of semantics-based program manipulation and partial evaluation. The workshop focuses on techniques and supporting theory for the analysis and manipulation of programs. Technical topics include, but are not limited to:

Program manipulation techniques: program transformation, program specialization, type specialization, syntax-directed partial evaluation, type-directed partial evaluation, normalization, continuation-passing conversion, reflection, rewriting, run-time code generation.

Program analysis techniques: abstract interpretation, static analysis, binding-time analysis, attribute grammars, constraints.

Related issues in language design and models of computation: imperative, functional, logical, object-oriented, parallel, distributed, mobile, secure.

Programs as data objects: staging, meta-programming, incremental computation, mobility, tools and techniques, prototyping and debugging.

Applications: systems programming, scientific computing, algorithmics, graphics, security checking, simulation, compiler generation, compiler optimization, decompilation.

Original results that bear on these and related topics were solicited.

PEPM'99 is taking place on January 22nd and 23rd, 1999, following POPL'99. It consists of 13 contributed papers, as well as three invited talks by Alan Bawden, Charles Consel, and Olin Shivers. The present BRICS technical report (distributed at the workshop) serves as an informal proceedings. The 13 papers were selected among 24 submissions. These submissions came from all over the world: US, UK, France, Germany, Denmark, Spain, Australia, New Zealand, Japan, and Singapore, incidentally all from academia. 85 reviews were generated (3.5 per submission). The PC meeting was electronic and took place in a week. The notifications included both the reviews and, when it was possible, a synthesis of the relevant part of the PC meeting.

Program committee

Kenichi Asai, University of Tokyo, Japan
Mike Ashley, University of Kansas, Kansas, USA
Anindya Banerjee, Stevens Institute of Technology, New Jersey, USA
Olivier Danvy, BRICS, University of Aarhus, Denmark [chair]
Mayer Goldberg, Ben Gurion University, Israel
Nevin Heintze, Bell Labs, New Jersey, USA
Morry Katz, Stanford University, California, USA
Michael Leuschel, University of Southampton, UK
Jacques Malenfant, Laboratoire VALORIA, Université de Bretagne Sud, France
Renaud Marlet, IRISA / INRIA, France
Kristoffer Rose, ENS-Lyon, France
Peter Sestoft, Royal Veterinary and Agricultural University, Denmark

Acknowledgements

Thanks to Alan Bawden, Charles Consel and Olin Shivers for accepting our invitation to come and give a talk. Olin in particular kindly stepped in to replace Neil Jones quite late in the game.

Thanks are also due to the following outside reviewers for their assistance: Philippe Boinot, Sandrine Chirokoff, Charles Consel, Andrzej Filinski, Karoline Malmkjær, David Naumann, Dino Oliva, Jens Palsberg, Morten Rhiger, Jon Riecke, Eva Rose, and Ulrik Pagh Schultz.

Finally, I am grateful to the PC members for their time, their careful reviews, and our spirited electronic PC meeting.

Olivier Danvy, Aarhus, Denmark, January 4th, 1999

Contents

Quasiquotation in Lisp

Alan Bawden

Brandeis University

bawden@cs.brandeis.edu

Abstract

Quasiquotation is the technology commonly used in Lisp to write program-generating programs. In this paper I will review the history and development of this technology, and explain why it works so well in practice.

1 Introduction

The subject of this paper is "quasiquotation". Quasiquotation is a parameterized version of ordinary quotation, where instead of specifying a value *exactly*, some holes are left to be filled in later. A quasiquotation is a "template".

Quasiquotation appears in various Lisp dialects, including Scheme [4] and Common Lisp [13], where it is used to write syntactic extensions ("macros") and other program-generating programs.

I have two goals in this paper. First, I wish to simply introduce the reader to quasiquotation and to record some of its history. Second, I wish to draw attention to the under appreciated synergy between quasiquotation and Lisp's "S-expression" data structures.

In this paper when I use the word "Lisp" I will mean primarily the Lisp dialects Common Lisp and Scheme, although what I say will usually be true of most other Lisp dialects. When I write examples of Lisp code, I will write in Scheme, although what I write will often run correctly in many other Lisp dialects.

0

2 Why Quasiquotation?

Before looking at how you would use quasiquotation to write program-generating programs in Lisp, let us consider what would happen if you had to use the C programming language instead. Seeing what can and cannot be easily accomplished in C will help clarify what a truly useful and well-integrated quasiquotation technology should look like.

2.1 Quasiquotation in C

Suppose that you are writing a C program that generates another C program. The most straightforward way for your program to accomplish its task is for it to textually construct the output program via string manipulation. Your program will probably contain many statements that look like:

```
fprintf(out, "for (i = 0; i < %s; i++)
                %s[i] = %s;\n",
        array_size,
        array_name,
        init_val);
```

C's `fprintf` procedure is a convenient way to generate the desired C code and to specialize it as required. Without `fprintf`, you would have had to write:

```
fputs("for (i = 0; i < ", out);
fputs(array_size, out);
fputs("; i++) ", out);
fputs(array_name, out);
fputs("[i] = ", out);
fputs(init_val, out);
fputs(";\n", out);
```

You can tell at a glance that the `fprintf` statement generates a syntactically legal C statement, but when looking at the sequence of calls to `fputs`, this isn't as clear.

Using `fprintf` achieves the central goal of quasiquotation: It assists you by allowing you to write

expressions that look as much like the desired output as possible. You can write down what you want the output to look like, and then modify it only slightly in order to parameterize it.

Although `fprintf` makes it *easier* for you to write C programs that generate C programs, two problems with this technology will become clear to you after you have used it for a while:

- The parameters are associated with their values positionally. You have to count arguments and occurrences of "`%s`" to figure out which matches up with which. If there are a large number of parameters, errors will occur.
- The string substitution that underlies this technology has *no* understanding of the syntactic structure of the programming language being generated. As a result, unusual values for any of the parameters can change the meaning of the resulting code fragment in unexpected ways. (Consider what happens if `array_name` was "`*x`": C's operator precedence rules cause the resulting code to be parsed as "`*(x[i])`" rather than the presumably intended "`(*x)[i]`".)

The first problem could be addressed by somehow moving the parameter expressions into the template. You would much rather write something like:

```
subst("for (i = 0; i < $array_size; i++)
        $array_name[i] = $init_val;");
```

But even if this could be made to work in C,[1] you will still be left with the second problem: flat character strings are not really a very good way to represent recursive structures like expressions. You will probably wind up adopting some convention that inserts extra pairs of parenthesis into your output just to be sure that it parses the way you intended.[2]

We have identified three goals for a successful implementation of quasiquotation:

- Quasiquotation should enable the programmer to write down what she wants the output to look like, modified only slightly in order to parameterize it.
- The parameter expressions should appear inside the template, in the positions where their values will be inserted.
- The underlying data structures manipulated by quasiquotation should be rich enough to represent recursively defined structures such as expressions.

[1] I can think of several ways to do it—none of them very pleasant.

[2] A technique often employed by users of the C preprocessor for the very same reason.

As we shall see, the achievement of that last goal is where Lisp will really shine.

2.2 Quasiquotation in Lisp

Now suppose that you are writing a Lisp program that generates another Lisp program. It would be highly unnatural for a Lisp program to accomplish such a task by working with character strings, as the C code in the previous section did, or even to work with tokens, as the C preprocessor does. The natural way for a Lisp program to generate Lisp code is for it to work with Lisp's "S-expression" data structures: lists, symbols, numbers, etc. So suppose your aim is to generate a Lisp expression such as:

```
(do ((i 0 (+ 1 i)))
    ((>= i array-size))
  (vector-set! array-name
               i
               init-val))
```

The primitive Lisp code to construct such an S-expression is:

```
(list 'do '((i 0 (+ 1 i)))
      (list (list '>= 'i array-size))
      (list 'vector-set! array-name
            'i
            init-val))
```

It is an open question whether this code is more or less readable than the C code in the previous section that used repeated calls to `fputs` instead of calling `fprintf`. But Lisp's quasiquotation facility will let you write instead:

```
`(do ((i 0 (+ 1 i)))
     ((>= i ,array-size))
   (vector-set! ,array-name
                i
                ,init-val))
```

A backquote character (`) precedes the entire template, and a comma character (,) precedes each parameter expression inside the template. (The comma is sometimes described as meaning "unquote" because it turns off the quotation that backquote turns on.)

It is clear what this backquote notation is trying to *express*, but how can a Lisp implementation actually make this *work*? What is the underlying technology?

The answer is that the two expressions above are actually *identical* S-expressions! That is, they are identical in the same sense that

```
(A B . (C D E . ()))
```

and

```
(A B C D E)
```

are identical. Lisp's S-expression parser (traditionally called "`read`") expands a backquote followed by a template into Lisp code that constructs the desired S-expression. So you can write:

```
`(let ((,x ',v)) ,body)
```

and it will be exactly as if you had written:

```
(list 'let
      (list (list x (list 'quote v)))
      body)
```

Backquote expressions are just a handy notation for writing complicated combinations of calls to list constructors. The exact expansion of a backquote expression is not specified—`read` is allowed to build any code that constructs the desired result.[3] (One possible expansion algorithm is described in appendix A.) So the backquote notation doesn't change the fact that your program-generating Lisp program works by manipulating Lisp's list structures.

Clearly this backquote notation achieves at least the first two of our three goals for quasiquotation: the code closely resembles the desired output and the parameter expressions appear directly where their values will be inserted.

Our third goal for a quasiquotation technology was that the underlying data structures it manipulates should be appropriate for working with programming language expressions. It is not immediately clear that we have achieved that goal. List structure is not quite as stark a representation as character strings, but it is still pretty low-level.

We can represent expressions using list structure, but perhaps we would be happier if, instead of manipulating lists, our quasiquotation technology manipulated objects from a set of abstract data types that were designed specifically for each of the various different syntactic constructs in our language (variables, expressions, definitions, `cond`-clauses, etc.). After abandoning character strings as too low-level, it seems very natural to keep moving towards even higher-level data structures that capture even more of the features of the given domain.

But this would be unnecessary complexity.

The problem with strings was that string substitution didn't respect the intended recursive structure of expressions represented as strings. But list structure substitution *does* respect the recursive structure of expressions represented as lists–and we haven't yet identified any additional problems that switching to an even higher-level representation would solve. The introduction of additional data types, and the procedures to manipulate them, would certainly introduce additional complexity into the system. The question is, would that complexity pay for itself by solving some problem, or making something more convenient?

For example, perhaps a well-designed set of abstract data types for various programming language constructs would prevent us from accidentally using quasiquotation to construct programs with illegal syntax. This additional safety might make the additional complexity worthwhile. Or perhaps a higher-level representation would enable us to do more powerful operations on programming language fragments beyond simply plugging them into quasiquotation templates. This additional functionality might also offset the increased complexity.

Such possibilities can't be ruled out, but after twenty years in its present form, no clearly superior representations for quasiquotation to work with have appeared. There just don't seem to be any compelling reasons to complicate matters my moving up to a higher-level representation. All three of our goals for a quasiquotation technology are nicely achieved by S-expression-based quasiquotation.

2.3 Synergy

In fact, there's a wonderful synergy between quasiquotation and S-expressions. They work together to yield a technology that's more powerful than the sum of the two ideas taken separately.

As we saw in the last section, Lisp code that constructs non-trivial S-expressions by directly calling Lisp's list constructing procedures tends to be extremely unreadable. The most experienced Lisp programmers will have trouble seeing what this does:

```
(cons 'cond
      (cons (list (list 'eq?
                        var
                        (list 'quote
                              val))
                  expr)
            more-clauses))
```

But even a novice can see what the equivalent quasiquotation does:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses)
```

[3] Actually, in Scheme [4], the exact expansion *is* specified: it expands into a special `quasiquote` expression. But I have never seen a programmer take advantage of this fact in a way that wasn't somehow problematic, so I am skeptical of its utility, and I will therefore ignore it in this paper.

S-expressions were at the core of McCarthy's original version of Lisp [6]. The ability to manipulate programs as data has always been an important part of what Lisp is all about. But without quasiquotation, actually working with S-expressions can be painful. Quasiquotation corrects an important inadequacy in Lisp's original S-expression toolkit.

The benefits flow the other way as well. As we've seen, character string based quasiquotation is an untidy way to work with recursive data structures such as expressions. But if our data is represented using S-expressions, substitution in quasiquotation templates works cleanly. So S-expressions correct an inadequacy in string-based quasiquotation.

Quasiquotation and S-expressions compensate for each other's weaknesses. Together they form a remarkably effective and flexible technology for manipulating and generating programs. So it is not surprising that although quasiquotation didn't become an official feature of any Lisp dialect until twenty years after the invention of Lisp, it was in common use by Lisp programmers for many years before then.

3 Embellishments

Now that the reader is convinced that quasiquotation in Lisp is an important idea, we can proceed to fill in the rest of the picture. Two important points about the technology and how it is used need to be presented. First, we need to introduce an additional feature, called "splicing". Second, we need to take a look at what happens when quasiquotations are nested.

3.1 Splicing

We brushed very close to needing splicing in a previous example. Recall:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses)
```

The value of the variable `more-clauses` is presumably a list of additional `cond`-clauses to be built into the `cond`-expression we are constructing. Suppose we knew (for some reason) that that list did not include an `else`-clause, and we wanted to supply one. We can always write:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,(append more-clauses
                  '((else #T))))
```

But calls to things like `append` are exactly what quasiquotation is supposed to help us avoid.

The backquote notation seems to suggest that we should be able to write instead:

```
`(cond ((eq? ,var ',val) ,expr)
       . ,more-clauses
       (else #T))
```

Unfortunately, this abuse of Lisp's "dot notation" will be rejected by the Lisp parser, `read`. Fortunately, this is a common enough thing to want to do that the backquote notation allows us to achieve our goal by writing:

```
`(cond ((eq? ,var ',val) ,expr)
       ,@more-clauses
       (else #T))
```

This new two-character prefix, comma-atsign (`,@`), is similar to the plain comma prefix, except the following expression should return a *list* of values to be "spliced" into the containing list.[4]

The expanded code might read:

```
(cons 'cond
      (cons (list (list 'eq?
                        var
                        (list 'quote
                              val))
                  expr)
            (append more-clauses
                    '((else #T)))))
```

The reader who finds this expansion unenlightening has my sympathy. A simple example should make everything clear: If the value of `X` is `(1 2 3)`, then the value of

```
`(normal= ,X splicing= ,@X see?)
```

is

```
(normal= (1 2 3) splicing= 1 2 3 see?)
```

Splicing comes in handy in many situations. A look at the BNF for any Lisp dialect will reveal many kinds of expressions where a sequence of some sub-part occurs: the arguments in a function call, the variables in a `lambda`-expression, the clauses in a `cond`-expression, the variable binding pairs in a `let`-expression, etc. When generating code that uses any of these kinds of expressions, splicing may prove useful.

There is no analog of splicing for character string based quasiquotation. (This, incidentally, is another argument for the superiority of S-expression based quasiquotation.)

[4] Given the example, the reader may well wonder why the prefix comma-period (`,.`) wasn't chosen instead. The history section will address this question!

3.2 Nesting

Sometimes a program-generating program actually generates another program-generating program. In this situation, it will often be the case that a quasiquotation will be used to construct another quasiquotation. Quasiquotations will be "nested".

This sounds like the kind of highly esoteric construction that would only be needed by the most wizardly compiler-writers, but in point of fact, even fairly ordinary Lisp programmers can easily find themselves in situations where they need to nest quasiquotations. This happens because Lisp's "macro" facility works by writing Lisp macros in Lisp itself.[5] Once a programmer starts writing any macros at all, it is only a matter of time before he notices a situation where he has written a bunch of similar looking macro definitions. Clearly his next step is to design a macro-defining macro that he can use to generate all those similar looking definitions for him. In order to do this he needs nested quasiquotations.

To illustrate this point, imagine that you had written the following macro definition:[6]

```
(define-macro (catch var expr)
  `(call-with-current-continuation
     (lambda (,var) ,expr)))
```

This defines `catch` as a macro so that the call

```
(catch escape
  (loop (car x) escape))
```

is expanded by binding `var` to the symbol `escape` and `expr` to the list `(loop (car x) escape)` and executing the body of the macro definition. In this example, the definition's body is a quasiquotation that will return:

```
(call-with-current-continuation
  (lambda (escape)
    (loop (car x) escape)))
```

which is then used in place of the original `catch`-expression.

Procedures that accept a single-argument auxiliary procedure, and invoke it in some special way, are a fairly common occurrence. Calls to such procedures are often written using a `lambda`-expression to create the auxiliary procedure. So you may later find yourself writing another macro similar to `catch`:

```
(define-macro (collect var expr)
  `(call-with-new-collector
     (lambda (,var) ,expr)))
```

If you suspect you'll be writing many more instances of this kind of macro definition, you may decide to automate the process by writing the macro-defining macro:

```
(define-macro (def-caller abbrev proc)
  `(define-macro (,abbrev var expr)
     `(,',proc
        (lambda (,var) ,expr))))
```

The previous two macro definitions can then be written as

```
(def-caller catch
  call-with-current-continuation)
```

and

```
(def-caller collect
  call-with-new-collector)
```

The definition of `def-caller` would be completely straightforward if it wasn't for the mystical incantation comma-quote-comma (`,',`)—where the heck did *that* come from? It is *not* some new primitive notation, as comma-atsign (`,@`) was. It is the quasiquote notation and the traditional Lisp quote notation (`'`) being used together in a way that can easily be derived from their basic definitions.

Here is how you could have arrived at the definition of `def-caller`: First, manually expand the quasiquotation notation used in the definition of `catch`:

```
(define-macro (catch var expr)
  (list 'call-with-current-continuation
        (list 'lambda (list var) expr)))
```

Now you don't have to worry about being confused by nested quasiquotations, and you can write `def-caller` this way:

```
(define-macro (def-caller abbrev proc)
  `(define-macro (,abbrev var expr)
     (list ',proc
           (list 'lambda
                 (list var)
                 expr))))
```

Now turning the calls to `list` back into a quasiquotation, taking care to treat `',proc` as an expression, not a constant, yields the original definition.

Of course no Lisp programmer actually rederives comma-quote-comma every time she needs it. In practice this is a well-known nested quasiquotation cliché. Every Lisp programmer who uses nested quasiquotation knows the following three clichés:

[5] Most other programming languages with a macro facility use a different language to write the macros (e.g., the C preprocessor).

[6] This is not how macros are defined in any actual Lisp dialect that I am aware of—but this isn't a paper about how to write Lisp macros.

`,X` `X` itself will appear as an expression in the intermediate quasiquotation and its value will thus be substituted into the final result.

`,,X` The value of `X` will appear as an expression in the intermediate quasiquotation and the value of that expression will thus be substituted into the final result.

`,',X` The value of `X` will appear as a constant in the intermediate quasiquotation and will thus appear unchanged in the final result.

3.3 Nested Splicing

The interaction of nesting with splicing yields additional interesting fruit. Beyond the three clichés listed at the end of the previous section, things like `,@,X`, `,,@X`, `,@,@X` and `,@',X` will occasionally prove useful. To illustrate the possibilities consider just the following two cases:

`,@,X` The value of `X` will appear as an expression in the intermediate quasiquotation and the value of that expression will be *spliced* into the final result.

`,,@X` The value of `X` will appear as a *list* of expressions in the intermediate quasiquotation. The individual values of those expressions will be substituted into the final result.

Intuitively, an atsign has the effect of causing the comma to be "mapped over" the elements of the value of the following expression.

Making nested splicing work properly in all cases is difficult. The expander in appendix A gets it right, but at the expense of expanding into atrocious code.

4 History

The name "Quasi-Quotation" was coined by W. V. Quine [10] around 1940. Quine's version of quasiquotation was character string based. He had no explicit marker for "unquote", instead any Greek letter was implicitly marked for replacement. Quine used quasiquotation to construct expressions from mathematical logic, and just as we would predict from our experience representing expressions from C, he was forced to adopt various conventions and abbreviations involving parentheses. (He should clearly have used S-expressions instead!)

McCarthy developed Lisp and S-expressions [6] around 1960, but he did not propose any form of S-expression based quasiquotation.[7]

During 1960s and 1970s the artificial intelligence programming community expended a lot of effort learning how to program with S-expressions and list structure. Many of the AI programs from those years developed Lisp programming techniques that involved S-expression pattern matching and template instantiation in ways that are connected to today's S-expression quasiquotation technology. In particular, the notion of splicing, described in section 3.1, is clearly descended from those techniques.

But nothing from those years resembles today's Lisp quasiquotation notation as closely as the notation in McDermott and Sussman's Conniver language [7]. In Conniver's notation `` `X ``, `,X` and `,@X` were written `!"X`, `@X` and `!@X` respectively, but the idea was basically the same. (Conniver also had a `,X` construct that could be seen as similar to `@X`, so it is possible that this is how the comma character eventually came to fill its current role.)

The Conniver Manual credits the MDL language [3] for inspiring some of Conniver's features. MDL's notation for data structure construction is related, but it is sufficiently different that I'm unwilling to call it a direct ancestor of today's quasiquote. I'll have more to say about this issue in section 5.1.

After Conniver, quasiquote entered the Lisp programmer's toolkit. By the mid-1970s many Lisp programmers were using their own personal versions of quasiquote—it wasn't yet built in to any Lisp dialect.

My personal knowledge of this history starts in 1977 when I started programming for the Lisp Machine project at MIT. At that time quasiquotation was part of the Lisp Machine system. The notation was almost the same as the modern notation, except that `,,X` was being used instead of `,@X` to indicate splicing. This would obviously interfere with nested quasiquotation, but this didn't bother anyone because it was commonly believed that nested quasiquotation did not "work right".

I set out to figure out why nested quasiquotation should fail to work. Employing the same reasoning process I outlined above in section 3.2, I developed some test cases and tried them out. To my surprise, they actually worked perfectly. The problem was simply that no one had been able to make nested quasiquotation do what they wanted, not that there was a fixable bug.[8]

Now that we knew that nested quasiquotation did in fact work, we wanted to start using it, and so a new notation for splicing had to be found. I suggested

[7] Given that he was inspired by the λ-calculus, which has its own notion of substitution, one can't help but wonder what Lisp would have been like if he had also tried to work quasiquotation into the mixture!

[8] The expander in use at that time did have bugs handling nested splicing—but I didn't notice that.

`,.X` because `.,X` already does a kind of splicing (see section 3.1). I thought that this would be a good pun. Other members of the group thought it might be confusing. Probably inspired by Scheme, which in those days was using just `@X` to indicate splicing [14], we finally decided on `,@X` [15].[9]

Meanwhile McDermott altered the Conniver notation slightly by changing `!"X` to `|"X`. In this form it appeared in [1] in 1980.

As far as I know, the problems of nested splicing didn't get worked out until 1982. In January of that year Guy Steele circulated an example of quasiquotations nested *three* levels deep. He remarked that `,',',X` was "fairly obvious", but that it took him "a few tries" to get his use of `,,@X` right [12]. I responded with an analysis of nested splicing in which I observed that in order to get nested splicing correct, an expansion algorithm like the one presented in appendix A was required. A correct semantics and expansion algorithm for quasiquotation based on this observation now appears in [13].

Sometime during the 1980s we started to spell "quasi-quote" without the hyphen. My guess is that this is the result of the adoption of a special form named "`quasiquote`" into Scheme.

By the end of the 1980s, the standards for Common Lisp [13] and Scheme [4] had adopted the modern quasiquote notation.

5 Related Ideas

Here are three ideas related to quasiquotation that I think the reader might be interested in.

5.1 Alternate views of quotation

The backquote notation for quasiquotation (`` `X ``) is clearly inspired by Lisp's "forward quote" notation for ordinary quotation (`'X`). While the backquote notation is an abbreviation for a (potentially) complex series of calls to various list constructors, the forward quote notation is an abbreviation for a simple `quote` expression. (I.e., `'X` is the same as `(quote X)`.) McCarthy invented `quote` expressions as a mechanism for representing the constants that appeared in his M-expression language [6].

Smith [11] and Muller [8] have both argued that that there is something suspect about McCarthy's `quote`. Both are worried that `quote` somehow confuses levels of representation and reference. They would like to replace Lisp's `eval` function with something more in line with the λ-calculus notion of normalization.

Now given that a backquote followed by an expression that contains no commas is indistinguishable from a front quote, and given that these authors have concerns about front quote, they presumably have the same concerns about backquote. So it is interesting that both Smith's 2-LISP and Muller's M-LISP resemble the MDL language [3] in that expressions are notationally distinct from constants, and so constants are (in some sense) implicitly quasiquotations. E.g. an expression like

```
<cdr (X <+ 2 3> Z)>
```

returns

```
(5 Z)
```

This is why I would deny MDL's direct ancestry, via Conniver, of modern Lisp quasiquotation. MDL and Conniver were wandering around in the dark looking for convenient ways to construct list structure. Both found solutions, and Conniver may even have thought that it was following MDL. But in the light of hindsight, we can recognize that Conniver was firmly on the traditional Lisp path, while MDL had stepped off that path and had started in the direction suggested by Smith and Muller.

5.2 Parameterized code

The 'C language [2] adds a backquote operator to ordinary C as a way of specifying dynamically generated code. Their backquoted expressions, while explicitly inspired by Lisp's use of backquote, do not construct ordinary C data structures (such as structures and arrays), they only build dynamic code objects. This is not surprising since C has no ordinary data structures for representing C programs (other than character strings). Inside a backquote, atsign (`@`) is used to indicate the substitution of another dynamic code object. Splicing is meaningless and backquotes do not nest.

A particularly interesting difference from Lisp's quasiquotation is that in 'C backquoted code *can* reference variables from the lexically enclosing code. I.e., dynamic code objects are also closures. So a 'C backquote expression is sort of a cross between a quasiquotation and a λ-expression.

This is similar to Lamping's system of parameterization [5]. His `data`-expressions specify parameterized objects that are sort of cross between a quasiquotation and a closure. Lamping was motivated by a desire to manipulate expressions the way that quasiquotation would allow, but without disconnecting them from the context that they came from.

[9] When quasiquotation was migrated to MacLisp [9] `,.X` was chosen to mean a "destructive" version of splicing. They also thought it was a good pun. This notation was also adopted in Common Lisp [13].

Systems like these demonstrate that there is a lot of unexplored territory in between the notion of data and the notion of expression.

5.3 Self-generation

No paper on quasiquotation in Lisp would be complete without mentioning quasiquote's contribution to the perennial problem of writing a Lisp expression whose value is itself. The quasiquote notation enables many elegant solutions to this problem, but the following solution, due to Mike McMahon, is my personal favorite:

```
(let ((let ''(let ((let ',let))
                  ,let)))
  `(let ((let ',let))
     ,let))
```

6 Conclusion

It took a while for the Lisp community to discover it, but there's a synergy between quasiquotation and S-expressions.

Acknowledgments

Parts of this paper can trace their lineage back to two electronic mail conversations I had in 1982, one with Drew McDermott and one with Guy Steele. While collecting additional information I had helpful discussions with Robert Muller, Brian C. Smith, Gerald Jay Sussman, John Lamping, Glenn Burke and Jonathan Bachrach. It was Olivier Danvy who thought that the world needed a paper about quasiquotation in Lisp. Julia Lawall provided helpful feedback on early drafts. Pandora Berman also assisted in the preparation of this paper.

References

[1] E. Charniak, C. K. Riesbeck, and D. V. Mcdermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Assoc., Hillsdale, NJ, first edition, 1980.

[2] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for fast, efficient, high-level dynamic code generation. In *Proc. Symposium on Principles of Programming Languages.* ACM, Jan. 1996.

[3] S. W. Galley and G. Pfister. The MDL programming language. TR 293, MIT LCS, May 1979.

[4] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[5] J. Lamping. A unified system of parameterization for programming languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 316–326. ACM, July 1988.

[6] J. McCarthy. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[7] D. V. McDermott and G. J. Sussman. The Conniver reference manual. Memo 259a, MIT AI Lab, May 1972.

[8] R. Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–616, Oct. 1992.

[9] K. M. Pitman. The revised MacLisp manual. TR 295, MIT LCS, May 1983.

[10] W. V. Quine. *Mathematical Logic.* Harvard University Press, revised edition, 1981.

[11] B. C. Smith. Reflection and semantics in Lisp. In *Proc. Symposium on Principles of Programming Languages*, pages 23–35. ACM, Jan. 1984.

[12] G. L. Steele Jr., Jan. 1982. Electronic mail message.

[13] G. L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[14] G. L. Steele Jr. and G. J. Sussman. The revised report on SCHEME: A dialect of Lisp. Memo 452, MIT AI Lab, Jan. 1978.

[15] D. Weinreb and D. Moon. *Lisp Machine Manual.* Symbolics Inc., July 1981.

A Expansion Algorithm

This appendix contains a correct S-expression quasiquotation expansion algorithm.

I assume that some more primitive Lisp parser has already read in the quasiquotation to be expanded, and has somehow tagged all the quasiquotation markup. This primitive parser must supply the following four functions:

`tag-backquote?` This predicate should be true of the result of reading a backquote (`‘`) followed by an S-expression.

`tag-comma?` This predicate should be true of the result of reading a comma (`,`) followed by an S-expression.

`tag-comma-atsign?` This predicate should be true of the result of reading a comma-atsign (`,@`) followed by an S-expression.

`tag-data` This function should be applied to an object that satisfies one of the previous three predicates. It will return the S-expression that followed the quasiquotation markup.

The main entry point is the function `qq-expand`, which should be applied to an expression that immediately followed a backquote character. (I.e., the outermost backquote tag should be stripped off *before* `qq-expand` is called.)

```
(define (qq-expand x)
  (cond ((tag-comma? x)
         (tag-data x))
        ((tag-comma-atsign? x)
         (error "Illegal"))
        ((tag-backquote? x)
         (qq-expand
           (qq-expand (tag-data x))))
        ((pair? x)
         `(append
            ,(qq-expand-list (car x))
            ,(qq-expand (cdr x))))
        (else `',x)))
```

Note that any embedded quasiquotations encountered by `qq-expand` are recursively expanded, and the expansion is then processed as if it had been encountered instead.

`qq-expand-list` is called to expand those parts of the quasiquotation that occur inside a list, where it is legal to use splicing. It is very similar to `qq-expand`, except that where `qq-expand` constructs code that returns a value, `qq-expand-list` constructs code that returns a list containing that value.

```
(define (qq-expand-list x)
  (cond ((tag-comma? x)
         `(list ,(tag-data x)))
        ((tag-comma-atsign? x)
         (tag-data x))
        ((tag-backquote? x)
         (qq-expand-list
           (qq-expand (tag-data x))))
        ((pair? x)
         `(list
            (append
              ,(qq-expand-list (car x))
              ,(qq-expand (cdr x)))))
        (else `',x)))
```

Code created by `qq-expand` and `qq-expand-list` performs all list construction by using either `append` or `list`. It must never use `cons`. This is important in order to make nested quasiquotations containing splicing work properly.

The code generated here is correct but inefficient. In a real Lisp implementation, some optimization would need to be done. But care must be taken not to perform any optimizations that alter the behavior of nested splicing.

A properly optimizing quasiquotation expander for Common Lisp can be found in [13, Appendix C]. I am not aware of the existence of a correct optimizing quasiquotation expander for Scheme. (None of the Scheme implementations that I tested implement nested splicing correctly.)

C++ Templates as Partial Evaluation

Todd L. Veldhuizen*

Abstract

This paper explores the relationship between C++ templates and partial evaluation. Templates were designed to support generic programming but unintentionally provided the ability to write code generators and perform static computations. These features are accidental, and as a result their syntax and semantics are awkward. Despite being unwieldy, these techniques have become somewhat popular because they partially solve an important problem in scientific computing– how to provide libraries of domain-specific abstractions without performance loss. It turns out that the C++ template mechanism is really partial evaluation in disguise: C++ may be regarded as a two-level language in which types are first-class values and template instantiation resembles offline partial evaluation. That C++ templates have proven so useful underscores the potential importance of partial evaluation as a language feature.

1 Introduction

1.1 Overview

C++ templates are of interest since they solve some important performance problems in designing scientific computing class libraries (Section 1.2). Templates were intended to support generic programming, but accidentally provided the ability to perform static computations and code generation (Section 2). It turns out that the C++ template mechanism is a form of partial evaluation (Section 3); the experience of library developers working with templates may offer some useful insights about partial evaluation as a language feature (Section 4).

1.2 Motivation

Scientific computing requires many abstractions. Every subdomain has its own requirements, such as interval arithmetic, tensors, polynomials, automatic differentiation, meshes, and so on. For economic reasons, languages can only provide the few concepts common to all, such as floating-point numbers and arrays. In the past, people requiring more than the limited abstractions provided by mainstream languages have developed domain-specific languages (DSLs) for sparse arrays, intervals, automatic differentiation, adaptive mesh refinement, and others. Such languages are not ideal solutions: they tend to have short life-spans due to limited support and portability, suffer from a lack of tools (particularly debuggers), and it is usually impossible to use two DSLs in the same source file.

With the advent of languages such as C++ and Fortran 90 which provided object-oriented features and operator overloading, it has become possible to create abstractions *using the language itself*, and have notations which resemble the mathematics being implemented. In recent years there has been a proliferation of libraries which provide abstractions previously implemented as DSLs: data-parallel arrays, sparse arrays, interval arithmetic, and automatic differentiation are the most prominent examples.

However, the code generated by such libraries tends to be naive. For example, array objects implemented using operator overloading in C++ were originally 3-20 times slower than the corresponding low-level implementation. This was not because of poor design on the part of library developers, but rather because the language forced a style of implementation which was grossly inefficient. These performance problems are commonly called the *abstraction penalty*; efforts to solve them have been many and ongoing.

One might think that a sufficiently smart optimizer would eliminate the abstraction penalty. However, compilers have difficulty because they lack semantic knowledge of the abstractions: they do not know that a given piece of code represents (for example) a sparse array operation; instead, they just see pointers and loops. Knowledge of the semantics is essential for doing appropriate optimizations. Efforts to describe the semantics of a class to compilers have been largely unsuccessful. Libraries tend to have layers of abstraction and side-effects which cause further difficulties for optimizers. Also, it is doubtful there is a general-purpose solution: every problem domain has its own optimization tricks and peculiarities.

A more promising approach is to construct libraries which both provide abstractions, and control how they are optimized. This concept has been called "active libraries" [5]. Such libraries handle high-level optimization themselves, leaving only low-level optimizations (register allocation, instruction scheduling, software pipelining) to the optimizer.

Meta-level processing systems such as Xroma [5], MPC++ [9], Open C++ [3], and Magik [6] provide one possible route to building active libraries. Such systems open up the compilation system and allow libraries to plug in their own translation modules. While these approaches are showing promise, a potential disadvantage is the complexity

*Extreme Computing Laboratory, Indiana University Computer Science Department, Bloomington Indiana 47405, USA. tveldhui@acm.org

of code which one must write: modern languages have complicated syntax trees, and so code which manipulates and generates these trees tends to be complex as well.

C++ templates may point the way to a more usable solution. Template techniques have been used to solve the performance problems of C++ for arrays and linear algebra, and several libraries based on these techniques are being distributed (e.g. POOMA [11], Blitz++ [20], and MTL [14]). The syntax used to implement these libraries is awkward. Partial evaluation offers hope for a cleaner syntax: there is a strong resemblance between templates and partial evaluation, so some mechanism based directly on partial evaluation might solve the abstraction penalty problem, and avoid the awkward syntax of template techniques. There is some precedent for this hope: at least one scientific computing project [15] reinvented the notion of two-level languages to provide a convenient notation for generating runtime library routines for parallelizing compilers.

So there is potential for fruitful collaboration: library developers need the technologies being developed by the partial evaluation community. Researchers in partial evaluation might benefit from the experience of library developers working with templates: there is a growing understanding of what features are useful for creating active libraries.

2 The capabilities templates provide

2.1 Generic programming

The original intent of templates was to support generic programming, which can be summarized as "reuse through parameterization". Generic functions and objects have parameters which customize their behavior.[1] These parameters must be known at compile time (i.e. must be statically bound). For example, a generic vector class can be declared as:

```
1     template<typename T, int N>
      class Vector {
         ...

5     private:
          T data[N];
      };

      // Example use of Vector
10    Vector<int,4> x;
```

The `Vector` class takes two template parameters (line 1): `T`, a *type parameter*, specifies the element type for the vector; `N`, a *nontype parameter*, is the length of the vector. To use the vector class, template arguments must be provided (line 10). This causes the template to be *instantiated*: an instance of the template is created by replacing all occurrences of `T` and `N` in the definition of `Vector` with `int` and `4`, respectively.

Functions may also be templates. Here is a function template which sums the elements of an array:

```
11    template<typename T>
      T sum(T* array, int numElements)
      {
          T result = 0;
15        for (int i=0; i < numElements; ++i)
              result += array[i];
          return result;
      }

20    // Example use
      double a[] = { 1, 2, 3, 4 };
      double a_sum = sum(a,4);
```

This function works for built-in types, such as `int` and `float`, and also for user-defined types provided they have appropriate operators (`=`, `+=`) defined. Note that in line 22, no template parameters are provided – the compiler infers the template parameter `T` from the type of `a`. Templates allow programmers to develop classes and functions which are general-purpose, yet retain the efficiency of statically configured code.

2.2 Compile-time computations

Templates can be exploited to perform computations at compile time. This was discovered by Erwin Unruh [17], who wrote a program which produced these compile errors:

```
23    unruh.cpp 10: Cannot convert 'enum' to 'D<2>'
      unruh.cpp 10: Cannot convert 'enum' to 'D<3>'
      unruh.cpp 10: Cannot convert 'enum' to 'D<5>'
      unruh.cpp 10: Cannot convert 'enum' to 'D<7>'
      unruh.cpp 10: Cannot convert 'enum' to 'D<11>'
      ...
```

The program tricked the compiler into calculating a list of prime numbers! This capability was quite accidental, but has turned out to be very useful. Here is a simpler example which calculates `pow(X,Y)` at compile time:

```
29    template<int X, int Y>
      struct ctime_pow {
        static const int result =
           X * ctime_pow<X,Y-1>::result;
      };

35    // Base case to terminate recursion
      template<int X>
      struct ctime_pow<X,0> {
        static const int result = 1;
      };
40
      // Example use:
      const int z = ctime_pow<5,3>::result;  // z = 125
```

In C++, `::` is a scope resolution operator: `A::B` means, "the symbol B in scope A." The first template defines a structure `ctime_pow` which has a single data member `result`. The `static const` qualifiers of `result` indicate that its value must be known at compile time. `ctime_pow<X,Y>` refers to `ctime_pow<X,Y-1>`, so the compiler must recursively instantiate the template for `Y`,`Y-1`, `Y-2`, ... until it hits the base case provided by the second template, which is called a *partial specialization*. C++ compilers include a pattern-matching system to select among templates; in general the most specialized template is selected. This pattern-matching aspect of templates results in a resemblance to logic-programming systems; the implementation of `ctime_pow` above resembles a logic-programming implementation of `pow`:

```
43    pow(X,Y) :- X * pow(X,Y-1).
      pow(X,0) :- 1.
```

[1] In generic programming, *generic function* means a parameterized function; this is a different meaning than in e.g. CLOS and Dylan.

Here is an array class which uses `ctime_pow` to calculate the number of array elements needed:

```
45      // Array which is the same length in
        // every dimension
        template<typename T, int Length,
           int Dims>
        class IsoDimArray {
50         // ...
           static const int numElements =
              ctime_pow<Length,Dims>::result;
           T data[numElements];
      }
55
        // A 3x3 array: will have 9 elements
        IsoDimArray<float,3,2> x;

        // A 3x3x3 array: will have 27 elements
60      IsoDimArray<float,3,3> x;
```

When the `IsoDimArray` template is instantiated, `ctime_pow` is used to calculate the array size required. This allows the array elements to be allocated on the stack, which is much faster than dynamic memory allocation. Similar template techniques can be used to find greatest common divisors, test for primality, and so on – all at compile time. As an extreme example, it is possible to implement a subset of Lisp (encoded in templates) which is "interpreted" at compile time [4].

2.3 Code generation

It turns out that control structures (loops, if/else, case switches) can be mimicked in templates. For example, the definition of `ctime_pow` (Section 2.2) emulates a `for` loop using recursion. These compile-time programs can perform code generation by selectively inlining code as they are "interpreted" by the compiler. This technique is called *template metaprogramming* [19]. Here is a template metaprogram which generates a specialized dot product algorithm:

```
61      template<int I>
        inline float meta_dot(float* a, float* b)
        {
             return meta_dot<I-1>(a,b) + a[I]*b[I];
65      }

        template<>
        inline float meta_dot<0>(float* a, float* b)
        {
70           return a[0]*b[0];
        }

        // Example use:
        float x[3], y[3];
75      float z = meta_dot<2>::f(x,y);
```

In the above example, the call to `meta_dot` in line 75 results in code equivalent to:

```
76      float z = x[0]*y[0] + x[1]*y[1] + x[2]*y[2];
```

Recursion is used to unroll the loop over the vector elements. The syntax for writing such code generators is clumsy. However, the technique has proven very useful in producing specialized algorithms for scientific computing. The MTL library [14] uses similar generators to construct fast, fixed-size kernels for use in linear algebra routines. By composing these kernels, MTL is able to provide linear algebra operations which are sometimes faster than the native libraries provided by hardware vendors. Similar generators are used by the Blitz++ library [21] to specialize algorithms for small, fixed-size vectors and matrices.

It is even possible to create and manipulate static data structures at compile time, by encoding them as templates. This is the basis of the *expression templates* technique [18], which creates parse trees of array expressions at compile time. These parse trees are used to generate efficient evaluation routines for array expressions. This technique is the backbone of several libraries for object-oriented numerics [11, 20].

2.4 Traits

The *traits* technique [12] allows programmers to define "functions" which operate on and return *types* rather than data. As a motivating example, consider a template function which calculates the average value of an array. What should its return type be? If the array contains integers, a floating-point result should be returned. But a floating-point return type will not suffice for all arrays (for example, complex-valued arrays).

The problem may be solved by defining a *traits class* which maps from the type of the array elements to a type suitable for containing their average. Here is a simple implementation:

```
77      // default behavior: T -> T
        template<typename T>
        struct average_traits {
80        typedef T T_average;
      };

        // specialization: int -> float
        template<>
85      struct average_traits<int> {
          typedef float T_average;
      };
```

An appropriate type for averaging an array of type `T` is given by `average_traits<T>::T_average`. This pair of templates encodes the behavior, "use the array element type for calculating averages, except use float for arrays of integers." Again, note the strong resemblance between this traits class and a corresponding logic-programming implementation:

```
88      average_type(T)   :- T.
        average_type(int) :- float.
```

Here is an implementation of `average`:

```
90      template<class T>
        typename average_traits<T>::T_average
        average(T* array, int N)
        {
             typename average_traits<T>::T_average
95              result = sum(array,N);
             return result / N;
        }
```

This version correctly handles arrays of integers, floating-point, and complex arrays.

Similar problems are constantly encountered in templated class libraries. Templates provide general-purpose rules for creating functions and classes; traits allow you to handle the many exceptions which arise.

3 Templates as partial evaluation

Partial evaluators [10] regard a program's computation as containing two subsets: static computations, which are performed at compile time, and dynamic computations performed at run time. A partial evaluator evaluates the static portion of the program and outputs a specialized *residual* program.

To determine which portions of a program may be evaluated, a partial evaluator may perform *binding time analysis* to label language constructs and data as static or dynamic. Such a labelled language is called a *two-level language*. For example, a binding-time analysis of some scientific computing code might produce this two-level code fragment:

```
float volumeOfCube(float length)
{
      return pow(length,3);
}

float pow(float x, int N)
{
      float y = 1;
      for (int i=0; i < N; ++i)
            y *= x;
      return y;
}
```

in which static constructs have been overlined. A partial evaluator such as CMix [1] would evaluate the static constructs to produce the residual code:

```
 float volumeOfCube(float length)
{
     return pow3(length);
}

 float pow3(float x)
{
     float y = 1;
     y *= x;
     y *= x;
     y *= x;
     return y;
}
```

Such specializations can result in substantial performance improvements for scientific code [2, 8].

3.1 C++ as a two-level language

C++ templates resemble a two-level language. Function templates take both template parameters (statically bound) and function arguments (dynamically bound). For example, the `pow` function of the previous example might be declared in C++ as:

```
template<int N>
float pow(float x);     // Calculate pow(x,N)
```

The static data (`N`) is a template parameter, and the dynamic data (`x`) is a function argument. To incorporate template type parameters into this viewpoint, we need to regard types as first-class values. For example, in a declaration such as

```
template<typename X, int Y>
void func(int i, int j);
```

we regard `X` as a *type variable*, as in ML. Since C++ is statically typed, type variables may only be statically bound. This point of view has a certain simplifying power: for example, one can view `typedef`s as declarations of type variables:

```
typedef float float_type;
```

can be regarded as equivalent to the (fictional syntax)

```
typename float_type = float;
```

3.2 Template instantiation as offline PE

Partial evaluation of programs which contain explicit binding-time information is called *offline* partial evaluation. Template instantiation resembles offline partial evaluation: the compiler takes template code (a two-level language) and evaluates those portions of the template which involve template parameters (statically bound values). For example, consider this template class:

```
template<int X>
struct ulam {
    static const int result =
      ulam<(X % 2 == 0) ? (X/2) : (3*X+1)>::result;
};

// Base case: X = 1
template<>
struct ulam<1> {
    static const int result = 0;
};
```

The syntax `A ? B : C` is C's equivalent to the functional *if A then B else C*. When `ulam<X>` is instantiated, the `const` qualifier on `result` requires the compiler to evaluate the right-hand side of the assignment at compile time. So it determines if `X % 2 == 0` (whether `X` is even). If true, it instantiates `ulam<X/2>`; otherwise `ulam<3*X+1>` is instantiated. In theory, this continues until the compiler hits the base case `X=1`. Whether this recursion terminates for all `X` is a well-known open problem. In C++, it is impossible to determine if a chain of template instantiations will ever terminate. For this reason, compilers place arbitrary limits on the depth of template instantiation chains.

In C++, the binding time of code is inferred from the binding time of data: if an expression is assigned to a statically-bound value, the expression must be statically bound. Templates in C++ only allow monovariant binding times; it is not possible to have data or code which is statically bound in one context, but dynamically bound in another. For example, standard library routines such as `pow` and `cos` cannot be used at compile-time and for practical applications this limitation is frustrating.

C++ does allow part-static, part-dynamic structures. For example, the class

```
class Example {
    static const int x = 5;
    int y;
};
```

contains both a statically bound member (`x`), and a dynamic member (`y`). (Note that the `static` keyword refers to `x` being shared among objects of type `Example`, and not to binding times). Mixed static-dynamic data structures have proven very useful; for example, they are the basis of the expression templates technique [18].

4 What can be learned from the C++ experience?

Asymmetry between the static and dynamic language is bad. In C++, the static and dynamic aspects of the language bear little resemblance to each other. The static version of the language is maddeningly limited: there are no floating-point numbers, no objects, and no side-effects. It might be desirable to have near-perfect symmetry between the static and dynamic languages, even to the extent of allowing side-effects at compile-time. For example, being able to do file I/O and issue console messages at compile-time would be very useful: a library could generate specialized code based on the contents of a data or configuration file, and issue compile-time errors and warning messages. Not being able to issue customized diagnostic messages in C++ has hurt the usability of template libraries.

Reflection and meta-objects would be useful. A common headache in using C++ templates is that there is no way to enforce constraints on template parameters. If users unwittingly violate constraints, the result might be a cryptic error message, or in the worst case, the program might crash mysteriously. A staged or multilevel language with some simple reflection capabilities could provide a straightforward way to enforce constraints on template parameters (an idea due to Vandevoorde [5]). For example, the `sum` function template of Section 2.1 assumes that the template type parameter `T` is some numeric type which may be initialized to `0` and has the operator `+=` defined. With reflection and staging, the `sum`function could examine the parameter `T` and issue a friendly diagnostic message if these requirements were not met.

There are open problems in reconciling multilevel capabilities with other language features. Templates interact with other language features in bizarre ways. Some might regard this as evidence of poor design, but some of these shortcomings point the way to interesting, possibly unsolved problems. How can static initialization be handled sensibly in a multilevel language, particularly one with relative binding times? Is it possible to provide a multilevel language which preserves separate compilation? (This has been an enormous headache for C++). How should object-oriented language features interact with multilevel language features?

First-class types are good. The ability to construct and manipulate types has proven extraordinarily useful in writing scientific C++ libraries. In particular, the traits technique– being able to write functions which operate on and return types– has proven very valuable.

Fixed rules for selecting among multiple templates are bad. In C++, the rules for matching templates are well-designed – they work 95% of the time. It is the other 5% which is annoying, since the rules are hard-coded into the language. One can sometimes trick the compiler into selecting particular templates, but sometimes not. In a multilevel language, it would be possible to implement these rules in the language itself while avoiding the overhead of systems such as CLOS which resolve multimethods at run-time. This would allow matching rules to be customized when necessary. Languages in the ML family, with their pattern matching syntax, might provide a natural mechanism for duplicating the template pattern matching features.

Explicit binding-time annotations are good. Good optimizers with inter-procedural analysis and procedure cloning are approaching the power of online partial evaluation. But undirected specialization is only marginally useful to library developers. Optimizing the trade-off between compile-time and run-time evaluation is tricky: some non-trivial scientific computing codes have no dynamic inputs. Most importantly, online partial evaluation implies an assumption that execution time is proportional to the *amount of computation*. In most scientific computing codes, the cost is in *data flow* between the caches and main memory. Online PE can generate residual code with "less computation," but so far not "smarter computation". To generate smarter code, one needs a predictive model of the hardware: its caches, pipelines, and so on. These are very difficult decisions to automate. For example, the Cray optimizer has roughly 10^8 possible settings of its optimization switches; finding the best settings for a given code requires tedious experimentation, even by experts. However, in the hands of a library developer who understands the hardware, explicit binding-time annotations can be a powerful performance tool, since compile-time computations can be used to rearrange the flow of the run-time computation for efficient cache use.

It is sufficient to label data with binding times. In C++ templates, binding-time annotations apply to data only – there is no way to label pieces of code as static or dynamic. The binding times of code constructs follows naturally from binding times of data. Although this may sound limited, in practice it has been sufficient to solve many important performance problems in C++.

5 Conclusions

C++ templates have acquired a reputation as being overly complex. In their defense, templates started as a simple mechanism, and developed gradually over a decade in response to experimentation and the needs of users. This incremental process contributed to their current state of disarray. However, this same process has resulted in a useful inventory of the capabilities which library developers require. Anyone developing similar mechanisms based on partial evaluation may benefit from examining this inventory.

C++ with templates may be regarded as a two-level language in which types are first-class, statically-bound values. Template instantiation bears a striking resemblance to offline partial evaluation. That templates have proven so useful in C++ is an encouragement for continued work on partial evaluation as a language feature. Languages incorporating partial evaluation may offer a way to provide generic programming, code generation, and compile-time computation via a single mechanism with simple syntax. In particular, research on explicit binding-time annotations, staging, and the relationship between partial evaluation and type systems could have many fruitful applications; developers of scientific computing libraries would benefit from language features like these.

6 Acknowledgments

This work was supported in part by NSF grants CDA-9601632 and CCR-9527130. I am grateful to Robert Glück for useful discussions about partial evaluation and templates, to Michael Ashley and Olivier Danvy for shepherding this paper, and to anonymous reviewers for many helpful suggestions.

References

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] Berlin, A., and Weise, D. Compiling scientific code using partial evaluation. *Computer 23*, 12 (Dec 1990), 25–37.

[3] Chiba, S. A Metaobject Protocol for C++. In *OOPSLA'95* (1995), pp. 285–299.

[4] Czarnecki, K., and Eisenecker, U. Meta-control structures for template metaprogramming. http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm.

[5] Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., and Veldhuizen, T. L. Generative Programming and Active Libraries. In *Proceedings of the 1998 Dagstuhl-Seminar on Generic Programming* (1998), vol. TBA of *Lecture Notes in Computer Science*. (in review).

[6] Engler, D. R. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages (DSL'97)* (October 15–17, 1997).

[7] Glück, R., and Jørgensen, J. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation 10*, 2 (1997), 113–158.

[8] Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.

[9] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., and Kubota, K. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Reflection'96* (1996).

[10] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.

[11] Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., and Williams, T. Array design and expression evaluation in POOMA II. In *ISCOPE'98* (1998), vol. 1505, Springer-Verlag. Lecture Notes in Computer Science.

[12] Myers, N. A new and useful template technique: "Traits". *C++ Report 7*, 5 (June 1995), 32–35.

[13] Nielson, F., and Neilson, H. R. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, Mass., 1992.

[14] Siek, J. G., and Lumsdaine, A. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments* (1998).

[15] Stichnoth, J., and Gross, T. Code composition as an implementation language for compilers. In *USENIX Conference on Domain-Specific Languages* (1997).

[16] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices 32*, 12 (1997), 203–217.

[17] Unruh, E. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.

[18] Veldhuizen, T. L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[19] Veldhuizen, T. L. Using C++ template metaprograms. *C++ Report 7*, 4 (May 1995), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

[20] Veldhuizen, T. L. Arrays in Blitz++. In *ISCOPE'98* (1998), vol. 1505 of *Lecture Notes in Computer Science*.

[21] Veldhuizen, T. L., and Ponnambalam, K. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools 21*, 8 (Aug. 1996), 38–44.

Gödelisation in the untyped lambda calculus

Torben Æ. Mogensen
DIKU, University of Copenhagen, Denmark
email: torbenm@diku.dk

Abstract

It is well-known that one cannot inside the pure untyped lambda calculus determine equivalence. I.e., one cannot determine if two terms are beta-equivalent, even if they both have normal forms. This implies that it is impossible in the pure untyped lambda calculus to do Gödelisation, i.e. to write a function that can convert a term to a representation of (the normal form of) that term, as equivalence of normal-form terms is decidable given their representation. If the lambda calculus is seen as a programming language, this means that you can't from the value of a function find its text.

Things are different for simply typed lambda calculus: Berger and Schwichtenberg showed that, given its type, it is possible to convert a function into a representation of its normal form. This was termed "an inverse to the evaluation function", as it turns values into representations. However, the main purpose was for normalising terms. Similarly, Goldberg has shown that for a subset (proper combinators) of the pure untyped lambda calculus, Gödelisation is possible. However, the Gödeliser itself is not a proper combinator, though it (as all closed lambda terms) can be written by combining proper combinators.

In this paper, we investigate Gödelisation for the full untyped lambda calculus. To overcome the theoretical impossibility of this, we extend the lambda calculus with a feature that allows limited manipulation of extensional aspects: A finite set of labels on lambda terms and a predicate for comparing these. Within this extended lambda calculus, we can convert terms in the subset corresponding to normal form terms in the classical lambda calculus into their representation.

The extension of the lambda calculus (we conjecture) retains the Church-Rosser property. This implies that Gödelisation must yield identical results for beta-equivalent terms. We show only that terms in normal form Gödelise to their representation, but the implication is that any term that has a normal form will Gödelise to a representation of its normal form. Hence, Gödelisation can be used as a tool for normalising lambda terms.

1 Introduction

There are various ways to represent lambda terms as "data" inside the lambda calculus. One is to represent the term by its Gödel number and then represent that number inside the lambda calculus by e.g. a Church-numeral. More tractable representations can also be used, see e.g. Mogensen's papers [6], [7]. The representations used in these papers use the notion of *higher order abstract syntax* [9]. In essence, this means that variable bindings are represented by variable bindings. Given three *constructors*, *VAR*, *APP* and *ABS*, we can represent lambda terms by the following scheme:

$$\begin{array}{lcl} \lceil x \rceil & \equiv & VAR(x) \\ \lceil \lambda x.E \rceil & \equiv & ABS(\lambda x.\lceil E \rceil) \\ \lceil E_1\ E_2 \rceil & \equiv & APP(\lceil E_1 \rceil, \lceil E_2 \rceil) \end{array}$$

The constructors can be expressed in the lambda calculus in a way that allow operations on syntax, including alpha-equivalence testing.

The goal of this paper is to construct a lambda calculus term G such that $G\ E \longrightarrow \lceil E \rceil$ if E is in normal form. Equivalently (due to confluence), we can say that G takes a term and produces the representation of its normal form (if such exist). However, such a term G does provably not exist (see section 6.6 of Barendregts book on the lambda calculus [1]). Hence, we must relax the condition somewhat.

Mayer Goldberg [5] relaxes the condition by restricting the class of terms E that G works for to be the set of proper combinators. Berger and Schwichtenberg [2] relax the condition by requiring E to be in the simply typed lambda calculus and that the type of E is given.

Instead of restricting the set of terms that can be Gödelised, we want our Gödeliser to be able to take any normalizing closed lambda term and return a representation of its normal form. To obtain this we allow G to be written in an extension of the lambda calculus. G can not Gödelise all terms in the extended calculus, but it can do so for all closed normalising terms in the classical lambda calculus.

2 An extended lambda calculus

We extend the classical lambda calculus with labels: Each lambda abstraction is given a label. The labelling is not unique; different abstractions can share the same label. Indeed, we only need 3 different labels. To make the labelling visible we introduce a way of inspecting labels. The syntax of the extended lambda calculus is

$$
\begin{array}{lcll}
\Lambda^L & \rightarrow & x & \text{variable} \\
 & | & \lambda^l x.\Lambda^L & \text{labelled abstraction} \\
 & | & \Lambda_1^L\ \Lambda_2^L & \text{application} \\
 & | & l?\Lambda_1^L\ \Lambda_2^L\ \Lambda_3^L & \text{label inspection}
\end{array}
$$

We have the following reduction rules for the extended lambda calculus:

$$
\begin{array}{lcll}
(\lambda^l.E_1)\ E_2 & \longrightarrow & E_1[x \setminus E_2] & (\beta) \\
l?(\lambda^l x.E_1)\ E_2\ E_3 & \longrightarrow & E_2 & (L1) \\
l?(\lambda^{l'} x.E_1)\ E_2\ E_3 & \longrightarrow & E_3 \ \text{ if } l \neq l' & (L2)
\end{array}
$$

The (β) rule is the usual beta-reduction rule. $(L1)$ and $(L2)$ handle inspection of labels: If the first term is in weak head normal form and its label matches the label that is tested for, the second term is selected. If its label does not match, the third term is selected.

Reduction in the extended lambda calculus is strongly believed to be confluent, but at the moment we have not looked at proving this.

3 Gödelisation

As the basis of our Gödeliser we use the Gödeliser for the typed lambda calculus by Berger and Schwichtenberg [2], but using a notation similar to the extension of this work found in Danvy's type-directed partial evaluators [3]. In this, Gödelisation is defined by a pair of type-indexed functions $\downarrow_t$ and $\uparrow^t$, where $\downarrow_t$ takes a value of type t and produces an expression of type t and $\uparrow^t$ takes an expression of type t and produces a value of type t. $\downarrow_t$ and $\uparrow^t$ are mutually recursively defined by

$$
\begin{array}{lcl}
\downarrow_b\ v & = & v \\
\downarrow_{t_1 \rightarrow t_2}\ v & = & ABS(\lambda\ x : t_1.\ \downarrow_{t_2}\ (v\ (\uparrow^{t_1}\ (VAR(x))))) \\
 & & \text{where } x \text{ is a fresh variable}
\end{array}
$$

$$
\begin{array}{lcl}
\uparrow^b\ e & = & e \\
\uparrow^{t_1 \rightarrow t_2}\ e & = & \lambda\ x : t_1.\ \uparrow^{t_2}\ (APP(e, (\downarrow_{t_1}\ x)))
\end{array}
$$

If $\downarrow_t$ is applied to a closed term of type t, the representation of the normal form of that term is produced. The constructors *VAR*, *APP* and *ABS* are like those described in section 1, suitably modified to handle typed terms.

3.1 Untyped Gödelisation

As can be seen, the functions $\downarrow_t$ and $\uparrow^t$ use the type t to select between two actions: Either returning the argument unchanged or doing what can be seen as a two-level eta-expansion of the argument [4][3].

In the untyped world we don't have any type argument to base this selection of actions on. So when do we want to return the argument unchanged and when do we want to eta-expand?

Initially, the $\downarrow$ function will be applied to the term we wish to Gödelise. In this situation, we surely want to eta-expand to get the representation of the top-level abstraction[1]. But we also apply the $\downarrow$ function to the body of the abstraction we build in the representation of the term. This body is obtained by, in the original body, substituting the bound variable by the result of applying $\uparrow$ to the representation of a variable. If the function we Gödelise is $\lambda x.x$, we will hence apply $\downarrow$ to $\uparrow (VAR(x))$. In this situation we want to return $VAR(x)$ directly, in essence letting $\downarrow$ and $\uparrow$ cancel.

This is in fact the general idea: Whenever we apply $\downarrow$ to something produced by $\uparrow$, we let these cancel. Otherwise, we eta-expand.

We can use the labels and label testing capability of our extended lambda calculus to facilitate this: We let the results of applying $\uparrow$ use labels different from those used in the term we want to Gödelise. Now, $\downarrow$ can use the label to decide its action: If the label indicates that the argument is the result of applying $\downarrow$, it "undoes" the $\downarrow$ operation (cancelling the $\uparrow$ and $\downarrow$ operations), otherwise it does the eta-expansion. If we assume we use the label 1 as label for the results of applying $\uparrow$, we can write this as

$$
\downarrow\ v\ =\ 1?v\ (cancel\ v)\ (ABS(\lambda^0 x.\ \downarrow (v\ (\uparrow (VAR(x))))))
$$

The remaining problem is how we can make $\uparrow$ cancelable, i.e. how to program $\uparrow$ and *cancel*. We first look at a normal (not canceled) use of a value returned by $\uparrow$. This is inside the $\downarrow$ function, when it is used as an argument to the original term that we want to Gödelise. The original term might use this as a function or it might return it. We have already covered the latter case. The former case uses the eta-expansion done by $\uparrow$. Since we can not in advance know how the result of $\uparrow$ is used, we must assume that the eta-expansion is necessary and hence let $\uparrow$ do this always, making our first attempt at $\uparrow$ be

$$
\uparrow\ e\ = \lambda^1 x.\ \uparrow (APP(e, (\downarrow x)))
$$

However, this eta-expansion can not in general be undone, as any argument we give to it just produces another eta-expansion and so on *ad infinitum*. However, we can use the argument to the eta-expanded term as a signal that selects between undoing the last eta-expansion and doing another. We can use labels and label testing again for this purpose: We let $(cancel\ v)$ pass v an argument with a special label. The abstraction that is the result of $\uparrow$ will test for this label in its input and when it gets this it will undo the last eta-expansion. If we use 2 as this special signal-label, we get the final versions of $\downarrow$ and $\uparrow$:

$$
\begin{array}{lcl}
\downarrow\ v & = & 1?v\ (v\ \lambda^2 a.a)\ (ABS(\lambda^0 x.\ \downarrow (v\ (\uparrow (VAR(x))))))) \\
\uparrow\ e & = & \lambda^1 x.2?x\ e\ (\uparrow (APP(e, (\downarrow x))))
\end{array}
$$

We can then encode these mutually recursive functions by using Y-combinators:

$$
\begin{array}{lcl}
\downarrow & \equiv & Y\ (\lambda d.(\lambda u.D)\ (Y\ (\lambda u.U))) \\
 & & \text{where} \\
D & \equiv & \lambda v.1?v\ (v\ \lambda^2 a.a)\ (ABS(\lambda^0 x.d\ (v\ (u\ (VAR(x))))))) \\
U & \equiv & \lambda e.\lambda^1 x.2?x\ e\ (u\ (APP(e, (d\ x))))
\end{array}
$$

We have omitted the labels for the abstractions used in this encoding. We can use any label for these, as they will never get to a position where they are tested. Hence, we need only a total of three labels: 0 for use in the input term, 1 to designate results of $\uparrow$ and 2 to denote the special signal value. We can e.g. use 0 for all remaining abstractions.

For ease of reading, we will in the following use the mutually recursive definition of the functions.

As an example, figure 1 shows Gödelisation of $\lambda ab.a\ b$.

[1] Since we work with closed terms, we are sure that any normal-form term *will* have a top-level abstraction.

$$
\begin{array}{ll}
 & \downarrow (\lambda^0 a.\lambda^0 b.a\ b) \\
\longrightarrow & 1?(\lambda^0 a.\lambda^0 b.a\ b) \\
 & ((\lambda^0 a.\lambda^0 b.a\ b)\ \lambda^2 a.a) \\
 & (ABS(\lambda^0 x.\downarrow((\lambda^0 a.\lambda^0 b.a\ b)\ (\uparrow(VAR(x))))))) \\
\longrightarrow & ABS(\lambda^0 x.\downarrow((\lambda^0 a.\lambda^0 b.a\ b)\ (\uparrow(VAR(x))))) \\
\longrightarrow & ABS(\lambda^0 x.\downarrow(\lambda^0 b.\uparrow(VAR(x))\ b)) \\
\longrightarrow & ABS(\lambda^0 x.(1?(\lambda^0 b.\uparrow(VAR(x))\ b) \\
 & \quad((\lambda^0 b.\uparrow(VAR(x))\ b)\ \lambda^2 a.a) \\
 & \quad(ABS(\lambda^0 y.\downarrow((\lambda^0 b.\uparrow(VAR(x))\ b)\ (\uparrow(VAR(y)))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^0 b.\uparrow(VAR(x))\ b)\ (\uparrow(VAR(y))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\uparrow(VAR(x)))\ (\uparrow(VAR(y))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^1 z.2?z\ (VAR(x))\ (\uparrow(APP(VAR(x),(\downarrow z)))))\ (\uparrow(VAR(y))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow((\lambda^1 z.2?z\ (VAR(x))\ (\uparrow(APP(VAR(x),(\downarrow z)))))) \\
 & \qquad(\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(2?(\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))) \\
 & \qquad(VAR(x)) \\
 & \qquad(\uparrow(APP(VAR(x),(\downarrow(\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w))))))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),(\downarrow(\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x), \\
 & \qquad(1?(\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))) \\
 & \qquad((\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\lambda^2 a.a)) \\
 & \qquad(ABS(\lambda p.\downarrow((\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\uparrow(VAR(p)))))))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x), \\
 & \qquad((\lambda^1 w.2?w\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow w)))))\ (\lambda^2 a.a))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x), \\
 & \qquad(2?(\lambda^2 a.a)\ (VAR(y))\ (\uparrow(APP(VAR(y),(\downarrow(\lambda^2 a.a))))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\uparrow(APP(VAR(x),(VAR(y))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.\downarrow(\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y)))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y. \\
 & \qquad(1?(\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))) \\
 & \qquad((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\lambda^2 a.a)) \\
 & \qquad(ABS(\lambda^0 z.(\downarrow((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\uparrow(VAR(z)))))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y. \\
 & \qquad((\lambda^1 v.2?v\ (APP(VAR(x),(VAR(y))))\ (\uparrow(APP(VAR(x),(VAR(y))))))\ (\lambda^2 a.a))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y. \\
 & \qquad(2?(\lambda^2 a.a) \\
 & \qquad(APP(VAR(x),(VAR(y)))) \\
 & \qquad(\uparrow(APP(VAR(x),(VAR(y)))))))))) \\
\longrightarrow & ABS(\lambda^0 x.(ABS(\lambda^0 y.(APP(VAR(x),(VAR(y))))))) \\
\equiv & \lceil \lambda^0 x.\lambda^0 y.x\ y \rceil
\end{array}
$$

Figure 1: Example of Gödelisation

We prove lemma 2 by induction over the structure of N and D. The induction hypothesis is the statement of lemma 2: For $N \in \Lambda^N$, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$ and for $D \in \Delta$, $\overline{D} \longrightarrow^* \uparrow (\lceil D \rceil)$.

$N \equiv \lambda^0 x.N_1$:

$$
\begin{array}{lll}
 & \downarrow (\overline{\lambda^0 x.N_1}) & \\
\longrightarrow & 1?(\overline{\lambda^0 x.N_1})\ ((\overline{\lambda^0 x.N_1})\ (\lambda^2 a.a))\ (ABS(\lambda^0 x.\downarrow ((\overline{\lambda^0 x.N_1})\ (\uparrow (VAR(x))))))) & \text{by def. of } \downarrow \\
\longrightarrow & ABS(\lambda^0 x.\downarrow ((\overline{\lambda^0 x.N_1})\ (\uparrow (VAR(x))))) & \text{by } (L2) \\
\equiv & ABS(\lambda^0 x.\downarrow (((\lambda^0 x.N_1)[x_i \backslash \uparrow (VAR(x_i))])\ (\uparrow (VAR(x))))),\ x_i \in FV(\lambda^0 x.N_1) & \text{by def. of } \bar{\cdot} \\
\longrightarrow & ABS(\lambda^0 x.\downarrow (N_1[x_i \backslash \uparrow (VAR(x_i))][x \backslash \uparrow (VAR(x))])),\ x_i \in FV(\lambda^0 x.N_1) & \text{by } (\beta) \\
\equiv & ABS(\lambda^0 x.\downarrow (N_1[x_i \backslash \uparrow (VAR(x_i))])),\ x_i \in FV(N_1) & \\
\equiv & ABS(\lambda^0 x.\downarrow (\overline{N_1})) & \text{by def. of } \bar{\cdot} \\
\longrightarrow^* & ABS(\lambda^0 x.\lceil N_1 \rceil) & \text{by induction} \\
\equiv & \lceil \lambda^0 x.N_1 \rceil &
\end{array}
$$

$N \equiv D \in \Delta$:

$$
\begin{array}{lll}
 & \downarrow (\overline{D}) & \\
\longrightarrow^* & \downarrow (\uparrow (\lceil D \rceil)) & \text{by induction} \\
\longrightarrow^* & \lceil D \rceil & \text{by lemma 1}
\end{array}
$$

$D \equiv x$:

$$
\begin{array}{lll}
 & \overline{x} & \\
\equiv & \uparrow (VAR(x)) & \text{by def. of } \bar{\cdot} \\
\equiv & \uparrow (\lceil x \rceil) &
\end{array}
$$

$D \equiv D_1\ N_1$:

$$
\begin{array}{lll}
 & \overline{D_1\ N_1} & \\
\equiv & \overline{D_1}\ \overline{N_1} & \\
\longrightarrow^* & \uparrow (\lceil D_1 \rceil)\ \overline{N_1} & \text{by induction} \\
\longrightarrow & 2?\overline{N_1}\ \lceil D_1 \rceil\ (\uparrow (APP(\lceil D_1 \rceil, \downarrow (\overline{N_1})))) & \text{by def. of } \uparrow \\
\longrightarrow & \uparrow (APP(\lceil D_1 \rceil, \downarrow (\overline{N_1}))) & \text{by } (L2) \\
\longrightarrow^* & \uparrow (APP(\lceil D_1 \rceil, \lceil N_1 \rceil)) & \text{by induction} \\
\equiv & \uparrow (\lceil D_1\ N_1 \rceil) &
\end{array}
$$

□

Figure 2: Proof of lemma 2

4 Proof of correctness

In this section we will prove the correctness of the Gödeliser.

We start by proving that $\downarrow$ and $\uparrow$ cancel in the expected way, which we state in

Lemma 1 *For all $E \in \Lambda^L$, $\downarrow (\uparrow E) \longrightarrow^* E$.*

This is simple to prove:

$$
\begin{array}{lll}
\downarrow (\uparrow E) & \longrightarrow & \downarrow (\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x)))) \\
 & \longrightarrow & 1?(\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x))))\ ((\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x))))\ (\lambda^2 a.a))\ (ABS(\lambda^0 y.\downarrow (\cdots))) \\
 & \longrightarrow & ((\lambda^1 x.2?x\ E\ (\uparrow (APP(E, \downarrow x))))\ (\lambda^2 a.a)) \\
 & \longrightarrow & 2?(\lambda^2 a.a)\ E\ (\uparrow (APP(E, \downarrow (\lambda^2 a.a)))) \\
 & \longrightarrow & E
\end{array}
$$

□

We next define the input to the Gödeliser: Lambda terms in normal form with label 0 on all abstractions and not containing label tests:

$$
\begin{array}{lll}
\Lambda^N & \rightarrow & \lambda^0 x.\Lambda^N \\
 & | & \Delta \\
 & & \\
\Delta & \rightarrow & x \\
 & | & \Delta\ \Lambda^N
\end{array}
$$

Note that this includes open terms. We will need to handle open terms in a lemma below, even though the input to the Gödeliser is assumed to be closed.

We now define

$$\overline{N} \equiv N[x_i \backslash \uparrow (VAR(x_i))],\ x_i \in FV(N)$$

where $FV(N)$ is the set of free variables of N. Hence, $\overline{N}$ replaces all free variables of N by $\uparrow$ applied to the representations of the variables. Note that for closed N, $\overline{N} = N$.

We continue with the central lemma of our proof:

Lemma 2 *For $N \in \Lambda^N$, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$ and for $D \in \Delta$, $\overline{D} \longrightarrow^* \uparrow (\lceil D \rceil)$.*

The proof of lemma 2 can be found in figure 2.

We can now state the correctness theorem

Theorem 3 *If $N \in \Lambda^N$ and N is closed, then $\downarrow N \longrightarrow^* \lceil N \rceil$.*

The proof is simple: Since N is closed, $N \equiv \overline{N}$ and by lemma 2, $\downarrow \overline{N} \longrightarrow^* \lceil N \rceil$.

□

5 Implementation

The Gödeliser has been implemented in Scheme, where it has been used to "decompile" functions. Scheme doesn't have labels and label testing, but it does have pointer equality tests. While not quite equivalent to label testing, it has in conjunction with some of Scheme's non-functional features been sufficient to emulate the label testing needed in the Gödeliser. We will in this paper just show the program text (in figure 3) of the Scheme implementation and refer to another paper [8] for more details. Note that we have extended the $\downarrow$-part of the Gödeliser to work with base-type values. Hence, terms containing base-type values can be reified.

The call-by-value nature of Scheme makes the implementation unable to Gödelise terms that do not reduce to normal form under call-by-value reduction.

6 Discussion

While the extended lambda calculus is able to Gödelise classical lambda terms, it is not self-Gödelisable. For example, it is not possible inside the calculus to distinguish $(\lambda^0 x.x)$ from $(\lambda^0 x.1?x\ x\ x)$. It will be interesting to study what extensions are needed to the lambda calculus to make it fully self-Gödelisable, short of adding Gödelisation as a primitive operation. On a related issue, can we make smaller extensions of the classical lambda calculus than we have in this paper and still get Gödelisation of the classical fragment of this calculus? In other words, how many of the properties of the classical lambda calculus can we retain while allowing Gödelisation of the classical fragment?

While the extended lambda calculus inherits many properties of the classical lambda calculus, for example (we conjecture) confluence, it does not inherit all of them. As an example, eta-reduction is not valid in the extended calculus. Indeed, $(\lambda^0 x.x)$ and $(\lambda^0 x.(\lambda^0 y.x\,y))$ are Gödelised to representations that can be distinguished even in the classical lambda calculus.

In [10], it is shown that adding explicit Gödelisation (by reification) to the lambda calculus makes textual identity the only valid equivalence. By retaining beta-equivalence, we feel that our extension is less disruptive and more useful than explicit Gödelisation. In particular, it allows the Gödeliser to be used for normalisation of terms.

The Gödeliser has some similarities to Mogensen's self-reducer for the lambda calculus [6], and was indeed partly derived from this. The $\downarrow$ function in the Gödeliser corresponds to the R' function in the self-reducer while the $\uparrow$ function corresponds to the P function in the self-reducer. Where the P function in the self-reducer builds a pair of two values, the $\uparrow$ function in the Gödeliser builds a function that selects between two values based on the form of the argument. This isn't too far from how pairs are traditionally represented in the lambda calculus.

```
(define (downarrow v)
  (cond
   ((number? v) v)
   ((boolean? v) v)
   ((char? v) v)
   ((string? v) v)
   ((vector? v) v)
   ((symbol? v) (list 'quote v))
   ((null? v) v)
   ((pair? v)
     (list 'cons (downarrow (car v))
                 (downarrow (cdr v))))
   ((procedure? v)
     (if (memq v registered) (v special)
        (let ((x (gensym)))
          (list 'lambda (list x)
                 (downarrow (v (uparrow x)))))))))

(define (uparrow e)
  (let
    ((f (lambda (v)
          (if (eq? v special) e
              (uparrow (list e (downarrow v)))))))
    (set! registered (cons f registered))
    f))

(define registered '())

(define special '(special))

(define count 0)

(define (gensym)
  (set! count (+ 1 count))
  (string->symbol
    (string-append "x" (number->string count))))

(define (goedelise v)
  (set! count 0)
  (set! registered '())
  (downarrow v))
```

Figure 3: Scheme implementation of Gödeliser

7 Conclusion

We have presented an extension to the lambda calculus which allows Gödelisation of terms from the subset that corresponds to the classical lambda calculus. We have shown this by developing a Gödeliser and proving it correct.

Since the extensions can be modeled by the standard non-functional features of Scheme, the result can be used to make a decompiler (and partial evaluator) for a fragment of Scheme that includes the classical lambda calculus. When used as a partial evaluator or normaliser, the Scheme implementation of the Gödeliser performs call-by-value reduction to normal form, which is not a complete reduction strategy. However, this is a small limitation compared to the requirement that the residual programs must have normal forms.

References

[1] H. P. Barendregt. *The lambda Calculus, its syntax and semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, New York, Oxford, 2 edition, 1984.

[2] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, 1991.

[3] O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257. ACM, 1996.

[4] O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–228, September 1995.

[5] Mayer Goldberg. Gödelisation in the λ-calculus (extended version). Technical Report RS-96-5, BRICS, 1996.

[6] T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[7] T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.

[8] T. Æ. Mogensen. Normalization for a subset of scheme. In O. Danvy and P. Dybjer, editors, *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation*. to appear, 1998.

[9] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM, ACM Press, 1988.

[10] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, (10):189–199, 1998.

Deriving a Statically Typed Type-Directed Partial Evaluator

Morten Rhiger

BRICS *
Department of Computer Science
University of Aarhus †

Abstract

Type-directed partial evaluation was originally implemented in Scheme, a dynamically typed language. It has also been implemented in ML, a statically Hindley-Milner typed language. This note shows how the latter implementation can be derived from the former through a functional representation of inductively defined types.

1 Introduction

Type-directed partial evaluation is an approach to specializing a term written in a higher-order language. Such a higher-order term is specialized by normalizing it with respect to its type. Normalization is done by eta-expanding the term in a two-level lambda-calculus and statically beta-reducing the expanded term.

The two stages — eta-expansion and beta-reduction — share an intermediate result: a two-level term. We consider two versions of the type-directed partial evaluation algorithm:

(i) If the two-level term is represented as a value of an inductively defined data type then the algorithm can be implemented in any (Turing complete) language. In this case both eta-expansion and beta-reduction are implemented in this language.

(ii) If the algorithm is implemented in a higher-order language an alternative is to represent the two-level term as a mixture of object-language terms (dynamic parts) and implementation-language terms (static parts). In this case eta-expansion is implemented in the implementation language. It generates a two-level term whose implementation-language parts are beta-reduced by the native beta-reducer of the implementation language.

*Basic Research in Computer Science (http://www.brics.dk), Centre of the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: mrhiger@brics.dk

1.1 The Problem

Both of the versions of the algorithm are directed by the type of the term to be normalized. They are applied to a term and a representation of the type of the term. In (ii), if the type is given as an element of an inductively defined type then it is impossible to statically type-check the algorithm, and indeed this also has been implemented in a dynamically types language, Scheme [2, 4].

This note shows how to derive a statically typed analogue of (ii) from (i). It is obtained by Church-encoding types as higher-order values.

Andrzej Filinski was the first to use a Church-encoding of types for type-directed partial evaluation.

1.2 Related Work

The first statically typed version of type-directed partial evaluation is due to Andrzej Filinski in the spring of 1995. This unpublished work showed that type-directed partial evaluation could be implemented at all using a Hindley-Milner type system. The second one is due to Zhe Yang in the spring of 1996 [9]. Yang provides general methods for encoding type-indexed values in a Hindley-Milner typed language and applies them to type-directed partial evaluation. The present work was carried out in the fall of 1997 and, according to Olivier Danvy, it is the third independent implementation of type-directed partial evaluation in a Hindley-Milner typed language [7].

Kristoffer Rose implemented type-directed partial evaluation in Haskell in the spring of 1998 using type classes [8]. Haskell's type classes permit overloaded functions, i.e., functions that have several definitions, one for each type of argument. Type-directed partial evaluation fits exactly into this pattern.

In her M. Sc. thesis, Belmina Dzafic implements type-directed partial evaluation in Elf, a statically typed, constraint logical language (summer 1998) [5]. Furthermore, she proves the equivalence of the dynamically typed and the statically typed versions of type-directed partial evaluation. Earlier on, Catarina Coquand stated and proved the correctness of the type-directed partial evaluation algorithm using the proof editor Alf [3].

1.3 The Derivation

Type-directed partial evaluation is given by $\downarrow$ (reify) and $\uparrow$ (reflect) in Figure 1. Based on this algorithm we derive a

$$
\begin{array}{lrcl}
\text{(types)} & t & ::= & \mathrm{b} \mid t_1 \to t_2 \\
\\
\text{(reify)} & \downarrow^{\mathrm{b}} v & = & v \\
 & \downarrow^{t_1 \to t_2} v & = & \underline{\lambda} x.\ \downarrow^{t_2} (v \underline{@} (\uparrow_{t_1} x)) \\
 & & & \text{where } x \text{ is fresh} \\
\\
\text{(reflect)} & \uparrow_{\mathrm{b}} e & = & e \\
 & \uparrow_{t_1 \to t_2} e & = & \lambda x.\ \uparrow_{t_2} (e \underline{@} (\downarrow^{t_1} x)) \\
\\
 & \mathrm{tdpe} t\, v & = & \downarrow^{t} v
\end{array}
$$

Figure 1: Type-directed partial evaluation

statically typed, type-directed partial evaluator analogous to (ii) in the following steps:

- (Section 2.1) Starting from (i), we represent types as a datatype `Type` and representing both static and dynamic terms as a datatype `Term` we translate the $\downarrow$ and $\uparrow$ of Figure 1 into into `reify` and `reflect`. This solution is interpretive in that it uses an explicit static beta-reducer.
- (Section 2.2) We change the representation of static terms from elements of the datatype `Term` into higher-order values and replace the explicit beta reducer by the native beta reducer of the implementation language. This solution is not interpretive but it is also not statically typeable.
- (Section 3.1) We observe an invariant property: `reify` is applied only to types occurring covariantly in the source type, and `reflect` only to types appearing contravariantly in the source type. We encode this distinction in the datatype `Type`. This solution is not statically typeable but by changing the representation of types from the datatype `Type` to higher-order values we obtain a statically typed solution which is still not interpretive: it uses the native (implicit) beta-reducer to statically reduce the two-level term.

Using the implicit static beta-reducer or an explicit one is independent of the representation of types. However, using the implicit static beta-reducer together with a datatype representation of types does not yield a solution that is statically typeable in a Hindley-Milner typing system.

1.4 Overview

We consider four successive implementations of type-directed partial evaluation in this note.

In Section 2, object-language types are represented by an inductively defined type. Terms are represented by values of an inductively defined type in Section 2.1 and by a mixture of higher-order values and terms in Section 2.2.

In Section 3, types are represented by higher-order values. The main result of this note is in Section 3.1 where terms are represented by higher-order values. For completeness we take a step backwards in Section 3.2 where terms are represented by values of an inductively defined data type.

In Section 3.3 we briefly consider the efficencies of the programs in the note. Section 4 concludes and in appendix A we present two extensions over the statically typed algorithm.

All programs in this note are written in a functional notation à la Haskell [6]. We defer the problem of generating fresh identifiers [7].

2 Inductively Defined Representation of Types

First we consider two implementations of type-directed partial evaluation that use a representation of types as elements of the following inductively defined type.

```
data Type =  Base | Func Type Type
```

The two implementations differ in the way the static parts of two-level terms are represented.

2.1 Inductively Defined Representation of Terms

In this approach, of which only an outline is given here, both the static and the dynamic parts of terms are represented by an inductively defined type.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term
```

The static syntax constructors are prefixed with an "`S`" and the dynamic syntax constructors are prefixed with a "`D`". Constants are both static and dynamic.

The algorithm proceeds in two stages: First, given a completely static term and its type, a fully eta-expanded two-level term is constructed using `reify` and `reflect` below. Second, the static parts of the term are beta-reduced (the beta-reducer is omitted here).

```
reify, reflect          :: Type -> Term -> Term

reify   Base          v =  v
reify   (Func t1 t2) v =
    DLam x (reify t2 (SApp v (reflect t1 (DVar x))))
    where x = fresh "x"

reflect Base          v =  v
reflect (Func t1 t2) v =
    SLam x (reflect t2 (DApp v (reify t1 (SVar x))))
    where x = fresh "x"
```

```
etaExpand t v      =  reify t v
staticBetaReduce v =  ...

tdpe               :: Type -> Term -> Term
tdpe t v           =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable* in a Hindley-Milner typing system and can be implemented in any language with inductively defined types, regardless of the typing discipline. The implementation-language type of the output of `reify` and `reflect` does not depend on the representation of the object-language type (a value of type `Type`).

The explicit use of a beta-reducer is undesirable since it embeds the lambda calculus into the implementing language via an interpreter. This is inefficient and the question arises whether we could not implement the algorithm in a higher-order language using the underlying beta-reduction mechanism of this implementation language.

2.2 Higher-Order Representation of Terms

The following approach uses the beta-reduction mechanism of the higher-order implementation language.

Types are represented by the same type as above. Two-level terms are represented by a mixture of implementation-language terms (values) and object-language terms.

```
reify   Base         v =  v
reify   (Func t1 t2) v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect Base         v =  v
reflect (Func t1 t2) v =
    \x -> reflect t2 (DApp v (reify t1 x))

etaExpand  t v  = reify t v

tdpe t v        = etaExpand t v
```

This program is *not statically typeable* in a Hindley-Milner typing system: The implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (a value of type `Type`).

Short of dependent types, at compile time, there is not enough information available so that the type-checker can accept the program. The above program corresponds to the original implementation of type-directed partial evaluation in Scheme [4].

3 Higher-Order Representation of Types

Observe that `reify` is always applied to types that occur *covariantly* in the source type (a value of type `Type`) and that `reflect` is always applied to types that occur *contravariantly* in the source type. We make this explicit by distinguishing between covariant occurrences (postfixed by "`P`" for positive) and contravariant occurrences (postfixed by "`N`" for negative):

```
data TypeP =  BaseP | FuncP TypeN TypeP
data TypeN =  BaseN | FuncN TypeP TypeN
type Type  =  TypeP
```

3.1 Higher-Order Representation of Terms

Now `reify` and `reflect` are

```
reify BaseP          v =  v
reify (FuncP t1 t2)  v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect BaseN        v =  v
reflect (FuncN t1 t2) v =
    \x -> reflect t2 (DApp v (reify t1 x))
```

This program is *not statically typeable* in a Hindley-Milner typing system. Again, the implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (values of types `TypeP` and `TypeN`).

In order to obtain a statically typeable program we apply the following change: instead of representing a positively occuring type t as a `TypeP` we represent it as a value equal to `(reify` t`)` and instead of representing a negatively occuring type t as a `TypeN` we represent it as a value equal to `(reflect` t`)`.

```
baseP, baseN     :: a -> a
funcP ::
    (Term -> a) -> (b -> Term) -> (a -> b) -> Term
funcN ::
    (a -> Term) -> (Term -> b) -> Term -> (a -> b)

baseP       v    =  v
funcP t1 t2 v    =  DLam x (t2 (v (t1 (DVar x))))
                    where x = fresh "x"
baseN       v    =  v
funcN t1 t2 v    =  \x -> t2 (DApp v (t1 x))

etaExpand t v    =  t v

tdpe             :: (a -> b) -> a -> b
tdpe t v         =  etaExpand t v
```

This program is *statically typeable* in a Hindley-Milner typing system. The implementation-language type of `tdpe` does not depend on the representation of the object-language type, but on the type of the object-language type.

Thus, even without dependent types, the type-checker has enough information to instantiate the polymorphic type of `tdpe`. This is the main result of this note.

For example, the type b (a base type) is represented by `Base` of type `Type` in Section 2.1. In the current section it is represented by `baseP` (i.e., the identity function) of type `a -> a`. Filinski and Yang's representation of this type is the pair of functions $(\downarrow^{\mathrm{b}}, \uparrow_{\mathrm{b}})$, i.e., a pair of identity functions. (See section A.2).

The type b $\rightarrow$ b is represented by `Func Base Base` of type `Type` in Section 2.1. In the current section it is represented by `funcP baseN baseP` of type `(Term -> Term) -> Term`. Filinski and Yang's representation of this type is the pair of functions $(\downarrow^{\mathrm{b}\rightarrow\mathrm{b}}, \uparrow_{\mathrm{b}\rightarrow\mathrm{b}})$.

As an example, let's specialize some terms that contains static redeces using the result of this section:

```
> :t tdpe baseP
tdpe baseP :: a -> a
> :type tdpe (funcP baseN baseP)
tdpe (funcP baseN baseP) :: (Term -> Term) -> Term
> tdpe baseP ((\x -> x) (Num 42))
Num 42
> tdpe (funcP baseN baseP) ((\x -> \y -> x) (Num 42))
DLam "x0" (Num 42)
>
```

3.2 Inductively Defined Representation of Terms

For the record, let us repeat the solution above using a "traditional" inductively defined representation of terms. To this end we use again the type of terms.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term

baseP       v =  v
funcP t1 t2 v =  DLam x (t2 (SApp v (t1 (DVar x))))
                 where x = fresh "x"
baseN       v =  v
funcN t1 t2 v =  SApp x (t2 (DApp v (t1 (SVar x))))
                 where x = fresh "x"

etaExpand t v         =  t v
staticBetaReduce v    =  ...

tdpe     :: Type -> Term -> Term
tdpe t v =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable*. It also requires explicit static beta-reduction (which is omitted here).

3.3 Pragmatics

We have compared the performance of the three statically typed solution in ML. We used a simple-minded, hand-coded static beta-reducer for the programs in Section 2.1 and Section 3.2, and the native beta-reducer of ML for the program in Section 3.1. The hand-coded reducer uses the same reduction strategy as the native reducer of ML.

Specializing the power function with respect to a static exponent of value 12 is about 9 times faster using the solution of Section 3.1 than using the solutions of sections 2.1 and 3.2. Specializing $(S\,K)\,K$ at type $\mathrm{b} \to \mathrm{b}$ is about 4 times faster using the solution of Section 3.1 than using the solutions of sections 2.1 and 3.2. These results confirm Berger, Eberl, and Schwichtenberg's empirical observations [1].

There do not appear to be any perceptible difference between the two solutions that use the hand-coded static reducer (Section 2.1 and Section 3.2).

4 Conclusion

Being directed by the *type* of a term, the type-directed partial evaluation algorithm requires a way of representing types. In dynamically typed languages an inductively defined sum over the different kind of types (base types, product types, function types, etc.) suffices. In statically typed languages with a Hindley-Milner typing system this does not work: the *type* of the algorithm depends on the *value* of the representation of the type.

The solution is to represent types as higher-order polymorphic functions. This works since the *type* of the algorithm thus depends on the implementation-language *type* of the representation of the object-language type.

Our work suggests to view the higher-order encoding as a functional representation of types, specialised to the purpose of being deconstructed by reify and reflect.

Acknowledgements

Thanks are due to Olivier Danvy and the anonymous referees for commenting earlier versions of this paper. Also many thanks to Kristoffer Rose for shepherding this paper.

A Extending the Statically Typed Algorithm

Consider again

```
tdpe baseP               :: a -> a
tdpe (funcP baseN baseP) :: (Term -> Term) -> Term
```

This indicates that constants of base type must be coerced to dynamic values in the source programs. For example, to obtain something of type `Term` we must coerce the integer `42` into a `Term` in

```
> tdpe baseP (Num 42)
Num 42
> tdpe (funcP baseN baseP) (\x -> (Num 42))
DLam "x0" (Num 42)
>
```

Furthermore, we must explicitly indicate the variance of types (by the postfix "`P`" or "`N`"). Both shortcomings are alleviated below.

A.1 The Need for Coercing Base Values

At covariant base types the explicit coercion of values of base type can be removed by distinguishing the base types. We introduce a more specific version of `baseP` for each base type.

```
numP    v =  Num v
strP    v =  Str v

tdpe numP               :: Int -> Term
tdpe strP               :: String -> Term
tdpe (funcP baseP numP) :: (Term -> Int) -> Term
```

These new functions will reify a static value into its dynamic counterpart. Static values of base type, such as integers and strings, are represented uniquely and these values are used directly when constructing the dynamic term. A similar solution does not work at contravariant base type since dynamic values of base type can be any dynamic term.

```
> tdpe (funcP baseN numP) (\x -> 42)
DLam "x0" (Num 42)
> tdpe (funcP baseN strP) (\x -> "fortytwo")
DLam "x0" (Str "fortytwo")
>
```

A.2 The Need for Specifying the Variance

The other shortcoming of the implementation — the explicit distinction between covariant and contravariant types — can also be alleviated. Instead of representing a type in two parts (i.e., a covariant part and a contravariant part as above) we can merge the two parts into a pair that represents the type, obtaining Filinski and Yang's solution [9].

```
base       :: (a -> a, b -> b)
func       :: (a -> Term, Term -> b) ->
                (c -> Term, Term -> d) ->
                  ((b -> c) -> Term, Term -> (a -> d))

base       = (reify, reflect)
    where reify   v = v
          reflect v = v
func t1 t2 = (reify, reflect)
    where reify   v =
              DLam x (fst t2 (v (snd t1 (DVar x))))
              where x = fresh "x"
          reflect v =
              \x -> (snd t2 (DApp v (fst t1 x))))

etaExpand t v  =  (fst t) v

tdpe           :: (a -> b,c) -> a -> b
tdpe t v       =  etaExpand t v
```

Using this implementation we can specialise terms without specifying the covariance and contravariance of the type involved.

```
> tdpe (func (func base base) base) (\f -> f (Num 8))
DLam "x0" (DApp (DVar "x0") (Num 8))
>
```

References

[1] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. October 1998. Draft. `http://www.mathematik.uni-muenchen.de/-schwicht/n8kurz.ps.Z`

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[3] Catarina Coquand. From semantics to rules: A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Proceedings of CSL'93*, number 832 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[4] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[5] Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.

[6] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

[7] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[8] Kristoffer Rose. Type-directed partial evaluation using type classes. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.

[9] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998. ACM Press. Extended version available as the technical report BRICS RS-98-9.

Interpreting Specialization in Type Theory

Peter Thiemann*
Institut für Informatik
Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract

We define the static semantics of offline partial evaluation for the simply-typed lambda calculus using a translation into a Martin-Löf-style type theory with suitable extensions. Our approach clarifies that the distinction between specialization-time and run-time computation in partial evaluation can model the phase distinction between compile-time and run-time computation in a module language. Working backwards from that connection, we define partial evaluation for a core language with modules.

Key Words lambda calculus, dependent types

1 Introduction

Program specialization subsumes a whole range of automatic transformations that aim at making programs faster while preserving their semantics. One particular instance is offline partial evaluation, a specialization technique that transforms an annotated program and some part of the input into a specialized program. In an annotated program each expression is either marked executable at specialization time or it is marked executable at run time. The specializer executes the specialization-time expressions and generates code for the run-time expressions.

Lambdamix [21, 22] is a specializer for the lambda calculus. Its operation depends on the *well-formedness* of annotated expressions, a criterion which ensures that no type errors occur during specialization. Lambdamix employs a partial type system [20] to verify well-formedness. Specialization of a well-formed expression either loops or it returns a specialized program, but it cannot fail due to type mismatches.

There are many approaches to define the semantics of Lambdamix-style specializers, based on denotational semantics [22, 21], operational semantics [27], logical frameworks [26], modal and temporal logics [16, 15], category theory [35], and higher-order rewriting [14]. We add a further color to this palette by employing type theory as a semantic foundation. Our plan is as follows:

- define the static semantics by translating an annotated expression into a term in a suitable type theory;
- obtain the specialized expression by applying an extraction function to the type derivation of the translated expression;
- reverse the connection to obtain an extension of Lambdamix that encompasses a module calculus.

The translation establishes a semantics for specialization via the semantics of type theory. We use a Martin-Löf-style type theory [32, 8, 37] with some non-standard extensions. Employing type theory as a semantic metalanguage yields denotational as well as operational models for free. Furthermore, it installs a sound framework for formal reasoning about specialization and a way of comparing different methods of specialization if they can all be specified in the same type theory.

Our particular translation to type theory applies to Lambdamix restricted to simply-typed programs[1]. Type inference for the translated expression performs the specialization-time computation. Extracting the specialized program from the type derivation involves reducing all redexes involving types in the type theory.

The main technical results are:

1. If E is a well-formed annotated expression then its translation type checks and the extraction function returns the same expression as applying Lambdamix to E, up to α-conversion.
2. A well-formedness criterion for an extension of Lambdamix with a module language.

The rest of the paper is structured as follows. We start with an introduction to the basic principles of Lambdamix for the simply-typed lambda calculus in Section 2. Next, in Section 3, we investigate the translation from Lambdamix to type theory in a step-by-step manner, discovering the necessary features of the type theory along the way. In Section 4 we state the formal properties of our translation. The connection to the phase distinction and to module calculus is outlined in Sec. 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

*This work has been done while at the School of Computer Science and Information Technology, University of Nottingham, UK. The author acknowledges support by EPSRC grant GR/M22840 "Semantics of Specialization".

[1] Full Lambdamix admits partially-typed programs where the untyped parts must be run-time expressions.

Basic knowledge of partial evaluation [7, 29, 13] is helpful for understanding this paper. Appendix A defines some notation used throughout. For convenience, Appendix B summarizes all translations defined in this paper and Appendix C states the rules of the underlying type theory.

2 Principles of Lambdamix

This section recalls the principles of Lambdamix specialization restricted to the simply-typed lambda calculus [21, 29]. We define the syntax of annotated expressions and annotated types along with a type system formalizing the well-formedness criterion of types and expressions.

Figure 1 shows the syntax of annotated expressions and types. The metavariable b ranges over binding times which denote phases in the processing of an expression, S for specialization-time or D for run-time. Constants k are specialization-time values by definition. They are restricted to numbers. The expression lift e converts a specialization-time integer into the corresponding run-time value. Addition $e + e$ serves as a representative primitive operation. Lambda abstraction $\lambda x.e$ and application $e@e$ have the usual call-by-name meaning. The conditional if0 $e_1\, e_2\, e_3$ tests the value of e_1 for the number zero and returns the value of e_2 or e_3. The fixpoint operator fix $x.e$ computes the fixpoint of $\lambda x.e$.

There are only integer and function types. Both have specialization-time and run-time variants. The Top function extracts the top-level binding-time annotation from a type.

Lambdamix insists that annotated types are *well-formed*. A type τ is well-formed if τ wft is a consequence of the rules in Fig. 2. Well-formedness restricts run-time types with a top-level annotation of D to have only run-time components. For a type assumption A, A wft means that τ wft for all $x : \tau$ in A.

Annotated expressions have annotated types as described in Fig 3. Apart from the annotations, it defines a type system with simple types. Regarding the annotations, all introduction rules (b-cst), (b-abs), and (b-fix) enforce the well-formedness of the type of the introduced object and their expressions carry the same annotation as the top-level annotation of the type. The annotation of the expression in the elimination rules (b-add), (b-app), and (b-if) is equal to the top-level annotation of the type of the eliminated object. The (b-lift) rule is special because its argument is always a specialization-time value while its result is a run-time value. The (b-if) rule enforces the usual restriction: if the condition has a run-time type (i.e., the specializer cannot evaluate it) then both branches need to have a run-time type, too.

There is a tiny addition with respect to the usual system: the type assumption A contains annotated pairs $x :^b \tau$ where b indicates the binding time of the binding expression for x. This does not change the set of typable terms or the assigned types in any way but it is necessary for proving one of the properties in Sec. 4.

2.1 Denotational specification

The Lambdamix specializer is the interpreter of annotated expressions shown in Fig. 4. The intended semantics extends a call-by-name lambda calculus. The interpretation of the run-time constructs is code generation in all cases, indicated by the underlined syntax constructors which are considered operators on an abstract type RExpr of run-time expressions. The cases for λ^D and $\mathbf{fix}^D$ involve the generation of a fresh variable name for n. Although name generation leads to complications in a semantic soundness-proof with respect to the rules in Fig. 3 (as demonstrated and repaired by Moggi [35]), the issue is not central to the topic of this paper.

The defining equations of the specializer rely on a number of auxiliary operators. $\mathbf{int}(k)$ returns the integer value of the syntactic representation of the constant k. **if0** tests its first argument for zero and returns its second or third argument. **fix** is a fixpoint operator of type (Val → Val) → Val. $\mathbf{quote}(v)$ maps the integer v to its syntactic representation of type RExpr.

2.2 Operational specification

The denotational framework just outlined follows the standard definition of Lambdamix. However, to prove the properties that we are interested in, an operational framework is more appropriate. Therefore, we define an alternative semantics of 2Expr in terms of reductions. For Lambdamix, all reductions are *static reductions*, that is, every redex only involves constructs which are annotated as specialization-time.

$$
\begin{array}{lcl}
i +^S j & \to & i + j \\
(\lambda^S x.e_1)@^S e_2 & \to & e_1\{x := e_2\} \\
\text{if0}^S\ 0\, e_1\, e_2 & \to & e_1 \\
\text{if0}^S\ n\, e_1\, e_2 & \to & e_2 \text{ if } n \neq 0 \\
\text{fix}^S\ x.e & \to & e\{x := \text{fix}^S\ x.e\}
\end{array}
$$

The usual static reduction relation $\Rightarrow$ is the compatible closure of $\to$ and $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. It is easy to prove the following lemma by induction on the definition of $\Rightarrow$.

Lemma 1 *If $e_1 \Rightarrow e_2$ then, for all ρ, $\mathcal{S}[\![e_1]\!]\rho = \mathcal{S}[\![e_2]\!]\rho$.*

A further simple induction on the length of a reduction sequence yields the next lemma.

Lemma 2 *If $e_1 \stackrel{*}{\Rightarrow} e_2$ then, for all ρ, $\mathcal{S}[\![e_1]\!]\rho = \mathcal{S}[\![e_2]\!]\rho$.*

3 From Lambdamix to type theory

We discuss a translation from annotated expressions to type theory by going through the different syntactic constructs. The basic idea is to translate a compile-time expression to an expression on the type level and a run-time expression to a value expression. The translation also requires to explicitly state the type of a translated compile-time expression, which is a kind.

For these reasons, there are two functions that work on 2Expr (more precisely, on a Lambdamix type derivation):

- $\mathcal{E}$ maps a typed 2Expr to a term at the value level and
- $\mathcal{T}$ maps a typed 2Expr to a term at the type level.

In addition, there is a function to classify terms at the type level:

- $\mathcal{K}$ which maps a 2Type to a kind.

The intuition is that $\mathcal{T}$ extracts the specialization-time parts whereas $\mathcal{E}$ extracts the run-time parts. $\mathcal{K}$ provides the "type" of the terms generated by $\mathcal{T}$, that is, , $\vdash_{\mathrm{tt}} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$ for suitable , . Here , $\vdash_{\mathrm{tt}} M : N$ is the typing judgement of the type theory with M and N ranging over its terms and , over its environments. For $\mathcal{E}$ the connection is somewhat more involved, since $\mathcal{T}$ does not always generate a type. It may also yield a constructor function, which must first be

$$
\begin{array}{lrcl}
\text{binding times} & b & ::= & S \mid D \qquad \text{ordered by} \qquad S < D \\
\text{expressions} & 2\text{Expr} \ni e & ::= & k \mid \text{lift}\ e \mid e +^b e \mid x \mid \lambda^b x.e \mid e@^b e \mid \text{if0}^b\ e\ e\ e \mid \text{fix}^b\ x.e \\
\text{types} & 2\text{Type} \ni \tau & ::= & \text{int}^b \mid \tau \xrightarrow{b} \tau \\
\text{top annotation} & & & \text{Top}(\text{int}^b) = b \\
 & & & \text{Top}(\tau_2 \xrightarrow{b} \tau_1) = b
\end{array}
$$

Figure 1: Syntax of Lambdamix

$$
\frac{\tau_1 \text{ wft} \quad \tau_2 \text{ wft} \quad b \le \text{Top}(\tau_1) \quad b \le \text{Top}(\tau_2)}{\tau_1 \xrightarrow{b} \tau_2 \text{ wft}}
$$

Figure 2: Well-formedness of annotated types

$$
\text{(b-cst)}\ \frac{A \text{ wft}}{A \vdash_{\text{lm}} k : \text{int}^S}
$$

$$
\text{(b-lift)}\ \frac{A \vdash_{\text{lm}} e : \text{int}^S}{A \vdash_{\text{lm}} \text{lift}\ e : \text{int}^D}
$$

$$
\text{(b-add)}\ \frac{A \vdash_{\text{lm}} e_1 : \text{int}^b \quad A \vdash_{\text{lm}} e_2 : \text{int}^b}{A \vdash_{\text{lm}} e_1 +^b e_2 : \text{int}^b}
$$

$$
\text{(b-var)}\ \frac{A \text{ wft} \quad x :^b \tau \text{ in } A}{A \vdash_{\text{lm}} x : \tau}
$$

$$
\text{(b-abs)}\ \frac{A\{x :^b \tau_2\} \vdash_{\text{lm}} e : \tau_1 \quad \tau_2 \xrightarrow{b} \tau_1 \text{ wft}}{A \vdash_{\text{lm}} \lambda^b x.e : \tau_2 \xrightarrow{b} \tau_1}
$$

$$
\text{(b-app)}\ \frac{A \vdash_{\text{lm}} e_1 : \tau_2 \xrightarrow{b} \tau_1 \quad A \vdash_{\text{lm}} e_2 : \tau_2}{A \vdash_{\text{lm}} e_1 @^b e_2 : \tau_1}
$$

$$
\text{(b-if)}\ \frac{A \vdash_{\text{lm}} e_0 : \text{int}^b \quad A \vdash_{\text{lm}} e_1 : \tau \quad A \vdash_{\text{lm}} e_2 : \tau \quad b \le \text{Top}(\tau)}{A \vdash_{\text{lm}} \text{if0}^b\ e_0\ e_1\ e_2 : \tau}
$$

$$
\text{(b-fix)}\ \frac{A\{x :^b \tau\} \vdash_{\text{lm}} e : \tau \quad \tau \text{ wft} \quad b \le \text{Top}(\tau)}{A \vdash_{\text{lm}} \text{fix}^b\ x.e : \tau}
$$

Figure 3: Typing rules of Lambdamix

$$
\begin{array}{lrcl}
\text{semantic values} & \text{Val} & = & \text{Int} + \text{RExpr} + (\text{Val} \to \text{Val}) \\
\text{environments} & \rho \in \text{Env} & = & \text{Var} \to \text{Val}
\end{array}
$$

$$\mathcal{S} : \text{2Expr} \to \text{Env} \to \text{Val}$$

$$
\begin{array}{lcl}
\mathcal{S}[\![k]\!] & = & \lambda\rho.\mathbf{int}(k) \\
\mathcal{S}[\![e_1 +^S e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho + \mathcal{S}[\![e_2]\!]\rho \\
\mathcal{S}[\![x]\!] & = & \lambda\rho.\rho(x) \\
\mathcal{S}[\![\lambda^S x.e]\!] & = & \lambda\rho.\lambda y.\mathcal{S}[\![e]\!]\rho[y/x] \\
\mathcal{S}[\![e_1 @^S e_2]\!] & = & \lambda\rho.(\mathcal{S}[\![e_1]\!]\rho)(\mathcal{S}[\![e_2]\!]\rho) \\
\mathcal{S}[\![\text{if0}^S\, e_0\, e_1\, e_2]\!] & = & \lambda\rho.\mathbf{if0}\,(\mathcal{S}[\![e_0]\!]\rho)\,(\mathcal{S}[\![e_1]\!]\rho)\,(\mathcal{S}[\![e_2]\!]\rho) \\
\mathcal{S}[\![\text{fix}^S\, x.e]\!] & = & \lambda\rho.\mathbf{fix}\;\lambda y.\mathcal{S}[\![e]\!]\rho[y/x] \\
 & & \\
\mathcal{S}[\![\text{lift}\, e]\!] & = & \lambda\rho.\mathbf{quote}(\mathcal{S}[\![e]\!]\rho) \\
\mathcal{S}[\![e_1 +^D e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho \underline{+} \mathcal{S}[\![e_2]\!]\rho \\
\mathcal{S}[\![\lambda^D x.e]\!] & = & \lambda\rho.\underline{\lambda} n.\mathcal{S}[\![e]\!]\rho[n/x] \\
\mathcal{S}[\![e_1 @^D e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho \underline{@} \mathcal{S}[\![E_2]\!]\rho \\
\mathcal{S}[\![\text{if0}^D\, e_1\, e_2\, e_3]\!] & = & \lambda\rho.\underline{\text{if0}}\,(\mathcal{S}[\![e_1]\!]\rho)\,(\mathcal{S}[\![e_2]\!]\rho)\,(\mathcal{S}[\![e_3]\!]\rho) \\
\mathcal{S}[\![\text{fix}^D\, x.e]\!] & = & \lambda\rho.\underline{\text{fix}}\; n.\mathcal{S}[\![e]\!]\rho[n/x]
\end{array}
$$

Figure 4: Lambdamix specializer

converted into a type by an auxiliary function **type**. Section 4 states the exact relations between Lambdamix type derivations and the translated terms.

Appendix C gives a summary of the target language.

3.1 Specialization-time integers

The encoding of specialization into type theory relies on creating specialization-time copies of the basic types "elevated" to kinds [4, 5]. For example, while int is the type of the run-time integers $0, 1, \ldots$, its elevated copy, the kind $\overline{\text{int}}$ with the types $\overline{0}, \overline{1}, \ldots$ as elements, models the compile-time integers. Technically, for each integer i, the type $\overline{i}$ has exactly one element $\top : \overline{i}$. Similarly, the primitive operator $e + e$ has an elevated copy, the binary type operator $. \overline{+} .$ (using infix notation) which comes with a set of associated δ rules: $\overline{i} \overline{+} \overline{j} \to \overline{i+j}$. Just like the value operator $+$ has type $\text{int} \to \text{int} \to \text{int}$, the type operator $\overline{+}$ has kind $\overline{\text{int}} \to \overline{\text{int}} \to \overline{\text{int}}$.

An alternative way of viewing the δ rules is to consider them as type equivalences so that the algebra of types is no longer a free term algebra, but rather a quotient algebra factored over the compatible closure of the δ rules. Such approaches are common in dealing with recursive types [6] and with record types [40].

For specialization-time integer expressions, all computation in the translated term takes place on the type level. However, we still need an expression of that type. This expression is never inspected, it is just there because we cannot have a type without an expression. Therefore, we introduce an expression $\top$ which can assume any type $\overline{i}$. The expression translation maps any constant and any specialization-time addition to this expression.

$$
\begin{array}{lcl}
\mathcal{E}[\![k]\!]\varphi & = & \top \\
\mathcal{E}[\![e_1 +^S e_2]\!]\varphi & = & \top
\end{array}
$$

The actual specialization-time computation takes place at the type level. The accompanying translation $\mathcal{T}$ from annotated expressions to terms at the type level takes care of that:

$$
\begin{array}{lcl}
\mathcal{T}[\![k]\!]\varphi & = & \overline{k} \\
\mathcal{T}[\![e_1 +^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \,\overline{+}\, \mathcal{T}[\![e_2]\!]\varphi
\end{array}
$$

Finally, there is a translation from annotated types to kinds of which we can deduce one case, so far.

$$\mathcal{K}[\![\text{int}^S]\!] \;=\; \overline{\text{int}}$$

For the minimal subset of expressions considered, we see that if $\vdash_{\text{m}} e : \tau$ then $\vdash_{\text{tt}} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$, which is true in general as we will see.

It happens to be the case that $\vdash_{\text{tt}} \mathcal{E}[\![e]\!]\varphi : \mathcal{T}[\![e]\!]\varphi$. This is a special case of the general relation between a type derivation for an annotated term and the type derivation of its translation. See Sec. 4 for the general case.

3.2 Run-time integers

The kind structure for specialization-time integers provides for values $\top$, types $\overline{i}$, and the kind $\overline{\text{int}}$. For run-time integers there is an analogous structure, which comprises values i, the type int, and the singleton kind INT. The only member of INT is the type int.

In Lambdamix, the expression lift e converts specialization-time integers to run-time integers. The translation must transform an expression e of type $\overline{i} : \overline{\text{int}}$ to an expression of type int : INT. To this end we require a non-standard feature in the type theory: a constant $\mathbf{choose} : \prod t : \overline{\text{int}} \,.\, t \to \text{int}$ which maps a one-element type of kind $\overline{\text{int}}$ and an expression of this type to the corresponding element of int. Hence, **choose** un-elevates these types.[2]

$$
\begin{array}{lcl}
\mathcal{E}[\![\text{lift}\, e]\!]\varphi & = & \mathbf{choose}\;(\mathcal{T}[\![e]\!]\varphi)\;(\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{T}[\![\text{lift}\, e]\!]\varphi & = & \text{int} \\
\mathcal{K}[\![\text{int}^D]\!] & = & \text{INT}
\end{array}
$$

[2] The operator **choose** is related to Girard's 0 operator of type $\prod \alpha : *\,.\,\alpha$, which can be used together with Girard's J operator of type $\prod \alpha : *\,.\, \prod \beta : *\,.\, \alpha \to \beta$ to construct a diverging term in System F [18]. In our calculus, this problem does not arise since **choose** is suitably restricted to the one-element types corresponding to elevated integers.

$$
\begin{array}{lcl}
\mathcal{E}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\lambda x : \mathbf{type}(\tau_2)(t).\mathcal{E}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{E}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi)(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![x]\!]\varphi & = & x
\end{array}
$$

Figure 5: Translation to expressions: specialization-time functions

$$
\begin{array}{lcl}
\mathbf{type} : \prod \tau : 2\mathrm{Type} \,.\, \mathcal{K}[\![\tau]\!] \to \mathrm{TYPE} & & \\
\mathbf{type}(\mathrm{int}^S)(f : \overline{\mathrm{int}}) & = & f \\
\mathbf{type}(\mathrm{int}^D)(f : \mathrm{INT}) & = & \mathrm{int} \\
\mathbf{type}(\tau_2 \xrightarrow{S} \tau_1)(f : \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!] \,.\, \mathbf{type}(\tau_2)(t) \to \mathbf{type}(\tau_1)(ft)
\end{array}
$$

Figure 6: Mapping constructors to types

The translation of the addition of run-time integers is straightforward.

$$
\begin{array}{lcl}
\mathcal{E}[\![e_1 +^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi + \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{T}[\![e_1 +^D e_2]\!]\varphi & = & \mathrm{int}
\end{array}
$$

3.2.1 Example

Consider the expression lift $(17 +^S 4)$.

1. For 17, we have $\mathcal{E}[\![17]\!]\varphi = \top$ and $\mathcal{T}[\![17]\!]\varphi = \overline{17}$.
2. For 4, we have $\mathcal{E}[\![4]\!]\varphi = \top$ and $\mathcal{T}[\![4]\!]\varphi = \overline{4}$.
3. For $e \equiv 17 +^S 4$, we have $\mathcal{E}[\![e]\!]\varphi = \top$ and $\mathcal{T}[\![e]\!]\varphi = \overline{17} \,\overline{+}\, \overline{4}$ by items 1 and 2. The last term is type-correct (or rather well-kinded) since $\overline{+} : \overline{\mathrm{int}} \to \overline{\mathrm{int}} \to \overline{\mathrm{int}}$.
4. By the conversion rule for types
$$\frac{\Gamma \vdash_{\mathrm{tt}} \top : \overline{17} \,\overline{+}\, \overline{4} \quad \Gamma \vdash_{\mathrm{tt}} \overline{21} : \overline{\mathrm{int}} \quad \overline{17} \,\overline{+}\, \overline{4} =_\delta \overline{21}}{\Gamma \vdash_{\mathrm{tt}} \top : \overline{21}}$$
5. For $e_2 \equiv \mathrm{lift}\ (17 +^S 4)$, we have $\mathcal{E}[\![e_2]\!]\varphi = \mathbf{choose}\ \overline{21}\ \top$ and $\mathcal{T}[\![e_2]\!]\varphi = \mathrm{int}$.

The extraction function (to be defined in Sec. 3.7) maps the type derivation of $\mathbf{choose}\ \overline{21}\ \top$ to the specialized expression 21.

3.3 Specialization-time functions

The specializer executes specialization-time functions. Consequently, in the translated term, we expect the type checker to reduce specialization-time β-redexes, too. Although the type checker cannot perform a reduction at the term level, it can perform substitution into the types as part of the application rule:

$$\frac{\Gamma \vdash_{\mathrm{tt}} M : \prod x : A \,.\, B \quad \Gamma \vdash_{\mathrm{tt}} N : A}{\Gamma \vdash_{\mathrm{tt}} MN : B\{x := N\}}$$

Hence, the translation of specialization-time functions involves dependent product types. The following example gives additional evidence. Consider the expression

$$(\lambda^S f.\lambda^S x.f @^S (f @^S x)) @^S (\lambda^S y.y +^S 1) @^S 7$$

Recalling that all specialization-time computation should happen at the type level, we intuitively expect for the translation of f which is bound to $\lambda^S y.y +^S 1$ a type like $\overline{n} \to \overline{n+1}$ for some suitable n. However, f has two uses, one at type $\overline{7} \to \overline{8}$, and the other at type $\overline{8} \to \overline{9}$. Hence, f must have a polymorphic type which abstracts over all possible n. The dependent product type $\prod t : \overline{\mathrm{int}} \,.\, t \to t \,\overline{+}\, \overline{1}$ does the trick.

To provide a term of such a dependent product type, a translated specialization-time lambda expression must take *two* parameters, the first one being a type-level parameter that holds all the specialization-time information and the second one being a value parameter of an associated type containing the run-time information. In the example, the specialization-time information is in $t : \overline{\mathrm{int}}$ and the value parameter has type t. Consequently, the translation of a specialization-time application must provide an additional parameter, too. Figure 5 shows the translation.

The translation of the specialization-time lambda refers to a function **type**. $\mathbf{type}(\tau)(t)$ converts the constructor t of kind $\mathcal{K}[\![\tau]\!]$ to a type. It is necessary for two reasons: Firstly, the annotation of x on the right side of Fig. 5 must be a type, i.e., a term of kind TYPE. Secondly, in general, t may not be a type, but it can be a constructor function. The latter case arises when the type τ is a specialization-time function type $\tau_2 \xrightarrow{S} \tau_1$. In this case, t's kind is defined by

$$\mathcal{K}[\![\tau_2 \xrightarrow{S} \tau_1]\!] = \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]$$

and the type translation produces the corresponding expressions at the type level:

$$
\begin{array}{lcl}
\mathcal{T}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{T}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi) \\
\mathcal{T}[\![x]\!]\varphi & = & \varphi(x)
\end{array}
$$

That is, a specialization-time function becomes a constructor abstraction and a specialization-time application becomes a constructor application.

In the example, the task of the **type** function is to map a constructor f of kind $\overline{\mathrm{int}} \to \overline{\mathrm{int}}$ to the type $\prod t : \overline{\mathrm{int}} \,.\, t \to t \,\overline{+}\, \overline{1}$. The solution is to abstract the parameter $t : \overline{\mathrm{int}}$ and to apply f to it to produce the result type. Figure 6 gives the details. It is trivial if its constructor argument is already a type.

$$\begin{array}{lcl}\mathcal{T}[\![\mathrm{fix}^S\, x.e:\tau]\!]\varphi & = & \mu\, c : \mathcal{K}[\![\tau]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto c] \\ \mathcal{E}[\![\mathrm{fix}^S\, x.e:\tau]\!]\varphi & = & \mu\, x : \mathbf{type}(\tau)(\mathcal{T}[\![\mathrm{fix}^S\, x.e]\!]\varphi).\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{T}[\![\mathrm{fix}^S\, x.e]\!]\varphi]\end{array}$$

Figure 7: Translation of specialization-time fixpoints

3.3.1 Example

In the running example, the translation produces for

$$\lambda^S y.y +^S 1 : \mathrm{int}^S \stackrel{S}{\to} \mathrm{int}^S$$

a term of type $\prod t : \overline{\mathrm{int}}\,.\, t \to t\,\overline{+}\,\overline{1}$.

Formally, we get for $f = \lambda^S y.y +^S 1$ and for $\tau = \mathrm{int}^S \stackrel{S}{\to} \mathrm{int}^S$:

$$\mathcal{E}[\![f]\!]\varphi = \lambda t : \overline{\mathrm{int}}.\lambda x : t.\top$$

$$\mathcal{T}[\![f]\!]\varphi = \lambda t : \overline{\mathrm{int}}.t\,\overline{+}\,\overline{1}$$

$$\begin{array}{l}\mathbf{type}(\tau)(\mathcal{T}[\![f]\!]\varphi) \\ = \prod t : \mathcal{K}[\![\mathrm{int}^S]\!]\,.\,\mathbf{type}(\mathrm{int}^S)(t) \to \mathbf{type}(\mathrm{int}^S)((\lambda t : \overline{\mathrm{int}}.t\,\overline{+}\,\overline{1})t) \\ = \prod t : \overline{\mathrm{int}}\,.\, t \to (\lambda t : \overline{\mathrm{int}}.t\,\overline{+}\,\overline{1})t \\ = \prod t : \overline{\mathrm{int}}\,.\, t \to t\,\overline{+}\,\overline{1}\end{array}$$

3.4 Run-time functions

Just as the translation maps a specialization-time function to a function with a polymorphic type level argument, the translation of a run-time function remains monomorphic. This intuition is straightforward to carry through.

The translation of run-time function types to kinds is just a constant function.

$$\mathcal{K}[\![\tau_2 \stackrel{D}{\to} \tau_1]\!] \quad = \quad \mathrm{TYPE}.$$

The well-formedness criterion reveals that the mapping from two-level types with top annotation D is straightforward. If $A \vdash_{\mathrm{m}} \lambda^D x.e : \tau_2 \stackrel{D}{\to} \tau_1$ it follows that $\tau_2 \stackrel{D}{\to} \tau_1$ wft, which means that all annotations in this type are D. Hence, there is exactly one constructor $\mathcal{C}[\![\tau_2]\!]$ with $\mathcal{C}[\![\tau_2]\!] : \mathcal{K}[\![\tau_2]\!]$:

$$\begin{array}{lcl}\mathcal{C}[\![\mathrm{int}^D]\!] & = & \mathrm{int} \\ \mathcal{C}[\![\tau_2 \stackrel{D}{\to} \tau_1]\!] & = & \mathcal{C}[\![\tau_2]\!] \to \mathcal{C}[\![\tau_1]\!]\end{array}$$

The translation to expressions and to constructors is straightforward, as promised.

$$\mathcal{E}[\![\lambda^D x.e : \tau_2 \stackrel{D}{\to} \tau_1]\!]\varphi \quad = \quad \lambda x : \mathcal{C}[\![\tau_2]\!].\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{C}[\![\tau_2]\!]]$$

$$\mathcal{T}[\![\lambda^D x.e : \tau_2 \stackrel{D}{\to} \tau_1]\!]\varphi \quad = \quad \mathcal{C}[\![\tau_2 \stackrel{D}{\to} \tau_1]\!]$$

We extend the conversion function from constructors to types accordingly.

$$\mathbf{type}(\tau_2 \stackrel{D}{\to} \tau_1)(c : \mathrm{TYPE}) \quad = \quad c$$

The translation of run-time application follows immediately from the translation of the lambda abstraction.

$$\begin{array}{lcl}\mathcal{E}[\![e_1 @^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{E}[\![e_2]\!]\varphi) \\ \mathcal{T}[\![e_1 @^D e_2 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]\end{array}$$

3.5 Conditionals

The conditional does not introduce new problems. We express the type of a specialization-time conditional using a conditional in the type expression and reduce it during type checking using the conversion rule. To this end, there is a conditional expression $\overline{\mathrm{if0}}\,T\,M\,N$ at the type level that reduces to M if T is the type $\overline{0}$ and to N, otherwise. This is a δ rule at the type level just as for elevated addition.

$$\frac{\Gamma \vdash_{\mathrm{tt}} T : \overline{\mathrm{int}} \quad \Gamma \vdash_{\mathrm{tt}} M : A \quad \Gamma \vdash_{\mathrm{tt}} N : B}{\Gamma \vdash_{\mathrm{tt}} \overline{\mathrm{if0}}\,T\,M\,N : \overline{\mathrm{if0}}\,T\,A\,B}$$

Using $\overline{\mathrm{if0}}\,T\,M\,N$ it is straightforward to define the translations.

$$\begin{array}{lcl}\mathcal{E}[\![\mathrm{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\mathrm{if0}}\,(\mathcal{T}[\![e_1]\!]\varphi)\,(\mathcal{E}[\![e_2]\!]\varphi)\,(\mathcal{E}[\![e_3]\!]\varphi) \\ \mathcal{T}[\![\mathrm{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\mathrm{if0}}\,(\mathcal{T}[\![e_1]\!]\varphi)\,(\mathcal{T}[\![e_2]\!]\varphi)\,(\mathcal{T}[\![e_3]\!]\varphi)\end{array}$$

The translations for the run-time conditional are obvious, thanks to the well-formedness criterion (b-if) which states that the branches of a run-time conditional must have run-time types, too.

$$\begin{array}{lcl}\mathcal{E}[\![\mathrm{if0}^D\, e_1\, e_2\, e_3]\!]\varphi & = & \mathrm{if0}\,(\mathcal{E}[\![e_1]\!]\varphi)\,(\mathcal{E}[\![e_2]\!]\varphi)\,(\mathcal{E}[\![e_3]\!]\varphi) \\ \mathcal{T}[\![\mathrm{if0}^D\, e_1\, e_2\, e_3 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]\end{array}$$

3.6 Fixpoints

Thus far, all of our translated terms live within a strongly normalizing type theory. If we wish to translate the fixpoint combinator, we give up strong normalization and require a partial type theory instead [10]. However, if we only admit fixpoint operators at run-time, all specialization-time computations remain strongly normalizing, the extraction function is terminating, and type checking remains decidable.

The translation of the run-time constructs is straightforward.

$$\begin{array}{lcl}\mathcal{E}[\![\mathrm{fix}^D\, x.e : \tau]\!]\varphi & = & \mathrm{fix}\,\lambda x : \mathcal{C}[\![\tau]\!].e \\ \mathcal{T}[\![\mathrm{fix}^D\, x.e : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]\end{array}$$

Specialization-time fixpoints are slightly more involved. The basic insight is that the type transformation must take the fixpoint of a constructor function. Once that is understood, the expression translation follows immediately. $\mu\ldots$ is the fixpoint operator that works on constructors. Figure 7 shows the definition.

3.6.1 Example

As an example, consider a contrived identity function at specialization time.

$$\mathrm{fix}^S\, i.\lambda^S n.\mathrm{if0}^S\, n\, 0\, (2 +^S i@^S(n -^S 1))$$

It has type $\tau = \mathrm{int}^S \stackrel{S}{\to} \mathrm{int}^S$ and its type translation yields

$$\mu\, i : \overline{\mathrm{int}} \to \overline{\mathrm{int}}.\lambda t : \overline{\mathrm{int}}.\overline{\mathrm{if0}}\,t\,\overline{0}\,(\overline{2}\,\overline{+}\,i(t\,\overline{-}\,\overline{1}))$$

1. $(\lambda t : K.\lambda x : B.M)TN \to M\{t, x := T, N\}$
   where K : kind
2. $(\lambda t : K.M)T \to M\{t := T\}$
   where K : kind
3. $\overline{\text{if0}}\,\overline{0}\,M\,N \to M$
4. $\overline{\text{if0}}\,\overline{n}\,M\,N \to N$ if $n \neq 0$
5. **choose** $\overline{n}\ e \to n$
6. $\mu\, t : K.M \to M\{t := \mu\, t : K.M\}$

Figure 8: Extraction

$$\diamond \rightsquigarrow \diamond; \varphi_0$$

$$\frac{A \rightsquigarrow \ , \ ; \varphi}{A, x :^D \tau \rightsquigarrow \ , \ , x : \mathcal{C}[\![\tau]\!]; \varphi[x \mapsto \mathcal{C}[\![\tau]\!]]}$$

$$\frac{A \rightsquigarrow \ , \ ; \varphi}{A, x :^S \tau \rightsquigarrow \ , \ , t : \mathcal{K}[\![\tau]\!], x : \mathbf{type}(\tau)(t); \varphi[x \mapsto t]}$$

Figure 9: Mapping between assumptions

There is nothing to do for this function at run time, hence the expression translation yields an uninteresting term.

$$\mu\, i : \mathbf{type}(\tau)(\mathcal{T}[\![\text{fix}^S\ x.e]\!]\varphi).\lambda t : \overline{\text{int}}.\lambda n : t.\overline{\text{if0}}\, t\, \top\, \top$$

3.7 Extraction

The extraction function is easy enough to describe: reduce all type-indexed constructs. The remaining expression is the result of specialization. More formally, extraction performs the reductions in Fig. 8 exhaustively in the translated expression.

4 Formal properties

There are two basic results. First, we have to check whether the translation of a well-formed expression type-checks in the type theory. Second, we have to check that the extraction function simulates static reduction in Lambdamix.

To establish the typing property, we need to map Lambdamix assumptions to assumptions of the type theory. This translation requires the annotated $x :^b \tau$ pairs. The function $A \rightsquigarrow \ , \ ; \varphi$ defined in Fig. 9 maps a Lambdamix assumption A to an assumption $,$ in the type theory and to a suitable environment φ for the translation functions. The symbol $\diamond$ stands for the empty assumption and φ_0 for the empty environment.

The following lemmas are both proved by induction on the derivation of $A \vdash_{\text{lm}} e : \tau$.

Lemma 3 *Suppose* $A \vdash_{lm} e : \tau$ *and* $A \rightsquigarrow \ , \ ; \varphi$.
Then $, \ \vdash_{tt} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$.

Lemma 4 *Suppose* $A \vdash_{lm} e : \tau$ *and* $A \rightsquigarrow \ , \ ; \varphi$.
Then $, \ \vdash_{tt} \mathcal{E}[\![e]\!]\varphi : \mathbf{type}(\tau)(\mathcal{T}[\![e]\!]\varphi)$.

Theorem 1 *Suppose* $e \in$ 2Expr *closed,* $\tau \in$ 2Type, *and* $\vdash_{lm} e : \tau$.
Then $\vdash_{tt} \mathcal{E}[\![e]\!]\varphi_0 : \mathbf{type}(\tau)(\mathcal{T}[\![e]\!]\varphi_0)$.

Proof Immediate from Lemma 4. □

Running Lambdamix on an annotated expression is equivalent to type checking the translated expression and applying the extraction function.

Theorem 2 *Suppose* $e \in$ 2Expr *closed,* $\tau \in$ 2Type, *and* $\vdash_{lm} e : \tau$. *Let* e' *be the result of the extraction transformation from Sec. 3.7 applied to* $\mathcal{E}[\![e]\!]\varphi_0$.
Then $\mathcal{S}[\![e]\!]\rho_0 = e'$ *up to* α*-conversion.*

Proof (sketch) Using the alternative formulation of Lambdamix in terms of reductions (see Sec. 2.2), we show that reduction in the type theory restricted to the rules from Sec. 3.7 simulates static reduction in Lambdamix.

More precisely, if $e_1 \Rightarrow e_2$ in Lambdamix then $\mathcal{E}[\![e_1]\!]\varphi_0 \Rightarrow_{tt} \mathcal{E}[\![e_2]\!]\varphi_0$. Therefore, if $e_1 \stackrel{*}{\Rightarrow} e_2$ in Lambdamix then $\mathcal{E}[\![e_1]\!]\varphi_0 \stackrel{*}{\Rightarrow}_{tt} \mathcal{E}[\![e_2]\!]\varphi_0$, by induction on the length of the reduction sequence. It is easy to see that if e_2 is a normal form then so is $\mathcal{E}[\![e_2]\!]\varphi_0$. Using the Lemma 1, $\mathcal{S}[\![e_1]\!]\rho_0 = \mathcal{S}[\![e_2]\!]\rho_0$ in particular.

Using the correctness of the binding-time analysis [38], since $\vdash_{\text{lm}} e_1 : \tau$ so does $\vdash_{\text{lm}} e_2 : \tau$. Furthermore, if e_2 is a normal form then it consists solely of run-time constructs.

Finally, induction on e_2 yields that $\mathcal{S}[\![e_2]\!]\rho_0 = \mathcal{E}[\![e_2]\!]\varphi_0$, as desired. □

5 Partial evaluation and the phase distinction

There is a deep relation between specialization and the "phase distinction" in type theory, and our work simulates one with the other. We demonstrate that Cardelli's original intuition of the phase distinction [4] was too restrictive, whereas later work on module languages by Harper and others [24] provides a perfect match. This latter work provides the starting point for extending Lambdamix to embody a module language, too.

5.1 The phase distinction — originally

Cardelli [4] argued for structuring a type system such that arbitrary computations might be performed at compile-time during type-checking while retaining a strict distinction between compile-time and run-time. He proposes [4, page 5]:

> **Phase Distinction Requirement**
> If A is a compile-time term and B is a subterm of A, then B must also be a compile-time term.

This requirement is too restrictive to model specialization. With Lambdamix, compile-time terms may have run-time subterms and vice versa. As examples, consider the following Lambdamix terms:

- $(\lambda^S x.x)@^S(\lambda^D y.y)$, a compile-time term with a run-time subterm;
- $\lambda^D x.(\lambda^S y.y)@^S x$, a run-time term with a compile-time subterm.

Instead of the phase distinction requirement, Lambdamix imposes the well-formedness criterion which restricts a compile-time term embedded into a run-time term to have a *run-time type*, which means that every component value of a run-time value is also a run-time value.

5.2 The phase distinction and modules

Our translation from Lambdamix to type theory corresponds closely to the translation of a higher-order module calculus into a structure calculus with explicit phase separation from earlier work by Harper, Mitchell, and Moggi [24][3] (hereafter HMM, for short). Since their work is concerned with making the phase separation of the module language explicit, their translation considers all lambda abstractions (functors) and applications (functor applications) as compile-time constructs. They do not consider the run-time constructs at all.

Looking at the translation of compile-time constructs [24, Table 7], the cases for variables (s), lambda abstraction ($\lambda s : S.M$) and application ($M_1 M_2$) correspond exactly to our definitions for specialization-time functions (see Sec. 3.3). In these cases, the type translation $\mathcal{T}$ yields their compile-time part and the expression translation $\mathcal{E}$ yields their run-time part. Furthermore, our function **type** gives the same result as the run-time part of their translation for dependent products (the type of a functor: $\prod s : S_1 . S_2$). Our function $\mathcal{K}$ provides the compile-time part.

While HMM does not consider compile-time integers and conditionals, a recent paper by Crary, Harper, and Puri considers fixpoints in the guise of recursive modules [12]. Their work includes a phase-splitting transformation (Fig. 4 of [12]) which is identical (modulo syntax) to our treatment of the specialization-time fixpoint (cf. Fig. 7).

The close connection leads to the question if we can specify the elaboration of the module language using Lambdamix or a suitable extension thereof. To this end, we consider a simple example, freely borrowing ML-style syntax.

```
signature M =
  sig
    type t
    val v : t * t
  end

functor IdM (structure m : M) =
  struct
    type t = m.t
    val  v = m.v
  end
```

In HMM's module calculus (which is based on MacQueen's ideas [31]) the signature `M` is modeled as a dependent sum type $\sum t : *.t \times t$, the elements of which are structures like $\langle \mathrm{int}, \langle 3, 4\rangle\rangle$ or $\langle \mathrm{bool}, \langle \mathrm{true}, \mathrm{false}\rangle\rangle$. The functor `IdM` is expressed as a dependent function $\lambda m : \mathtt{M}.\langle \mathrm{fst}(m), \mathrm{snd}(m)\rangle$ of type $\prod m : \mathtt{M}.\mathtt{M}$.

There are the following correspondences:

functor	specialization-time function
signature	partially static dependent sum type (a two-level type)
structure	partially static pair
type declaration	specialization-time value
value declaration	run-time value

Consequently, the annotated versions of the terms of the above example read:

- signature `M`: $\sum^S t : *.t \times^D t$;
- typical structure (element): $\langle \mathrm{int}, \langle \mathrm{lift}\ 3, \mathrm{lift}\ 4\rangle^D\rangle^S$;
- functor `IdM`: $\lambda^S m : \mathtt{M}.\langle \mathrm{fst}^S(m), \mathrm{snd}^S(m)\rangle^S$;
- the functor's type: $\prod^S m : \mathtt{M}.\mathtt{M}$

The annotation ^{S} of the dependent sum's type means that the specializer can access the first and second component of the underlying pair. Thus, the annotated dependent sum generalizes partially static pairs. In the same way, the specialization-time dependent product generalizes ordinary specialization-time functions.

[3] This was suggested to the author by Moggi in September 1998.

5.3 The phase distinction — extended

Consequently, it is now easy to extend Lambdamix to embrace a module calculus. The intention is to regard the types manipulated by the module calculus as specialization-time values. Since they are actually types of pure run-time values their embedding into 2Type must annotate all type constructors with ^{D}.

Figure 11 shows the extended syntax. It adds the introduction and elimination of pairs and also the types int and $e \to e$ to the previous set of expressions. The set of types now encompasses annotated versions of dependent sums and products, as well as the kind $*$ (the "type" of types) and the variable x. The function Top extracts the top-level annotation, again. Its definition on dependent sum and product types is obvious. A variable x is assumed to have annotation D. The embedding of the types at the expression level into 2Type motivates this choice. Once an application rule substitutes an expression-level type e for x in $\tau \in 2\mathrm{Type}$, it will carry the annotation D everywhere. Since $\mathrm{Top}(x\{x := e\}) = \mathrm{Top}(e) = D$ it must be that $\mathrm{Top}(x) = D$, too. Similar reasoning leads to $\mathrm{Top}(*) = D$.

Figure 12 defines additional well-formedness rules for Extended Lambdamix. The well-formedness judgement is now $A \vdash \tau : b$ for a type assumption A, a two-level type τ, and a binding time b. Obviously, $A \vdash \tau : b$ implies $\mathrm{Top}(\tau) = b$.

Any type variable x as well as $*$ are well-formed 2Type expressions of binding time D. A dependent sum is well-formed, if either the top-level binding times work out in the usual way or if the bound variable is a type (i.e., a run-time value) and the binding time of the second component is greater than or equal to the binding time of the sum constructor. Well-formedness for dependent product types works in the same way. Specializing the rules for the module level constructs yields the derived rules in Fig. 10.

$$\frac{A\{x :^b *\} \vdash \tau_2 : b_2 \quad b \le b_2}{A \vdash \sum^b x : *.\tau_2 : b}$$

$$\frac{A\{x :^b *\} \vdash \tau_2 : b_2 \quad b \le b_2}{A \vdash \prod^b x : *.\tau_2 : b}$$

$$\frac{A \text{ wft}}{A\{x : *\} \text{ wft}}$$

Figure 10: Derived well-formedness rules

Finally, Figure 13 shows the typing rules for the new expressions. The rules (b-int) and (b-fun) prescribe the type $*$ for the expressions int and $e_1 \to e_2$ (that is, they are types at the expression level), provided that e_1 and e_2 also have type $*$. The rules (b-pair), (b-fst), and (b-snd) are just the usual elimination rules for weak sums, augmented with annotations and well-formedness as appropriate. The rules

(b-dabs) and (b-dapp) generalize rules (b-abs) and (b-app) from Fig. 3.

The operational semantics does not change at all. In fact, existing implementations of specializers can be used with this extended well-formedness criterion.

5.4 Extended translation

Extending the translation is simple, now using HMM as a guideline. We need to provide expression and type translations for each of the new expressions as well as kind translations for the new two-level types.

For the new expressions denoting types, this is particularly easy, since these types denote run-time values.

$$\begin{array}{lcl}
\mathcal{E}[\![\text{int}]\!]\varphi & = & \text{int} \\
\mathcal{E}[\![e_1 \to e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi \to \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{T}[\![\text{int}]\!]\varphi & = & \text{TYPE} \\
\mathcal{T}[\![e_1 \to e_2]\!]\varphi & = & \text{TYPE} \\
\mathcal{K}[\![*]\!] & = & \text{kind}
\end{array}$$

Function types are now generalized to dependent product types. However, the expression and type translation of abstraction and application does not change with respect to Sec. 3.3 and Sec. 3.4. The only changes occur in the $\mathcal{K}$, $\mathcal{C}$, and **type** functions (see Fig. 14).

$$\begin{array}{lcl}
\mathcal{K}[\![\prod^D x : \tau_2 \,.\, \tau_1]\!] & = & \text{TYPE} \\
\mathcal{K}[\![\prod^S x : \tau_2 \,.\, \tau_1]\!] & = & \prod x : \mathcal{K}[\![\tau_2]\!] \,.\, \mathcal{K}[\![\tau_1]\!] \\
\mathcal{C}[\![\prod^D x : \tau_1 \,.\, \tau_2]\!] & = & \prod x : \mathcal{C}[\![\tau_1]\!] \,.\, \mathcal{C}[\![\tau_2]\!]
\end{array}$$

Since pairs were not part of the language before, we introduce all the translations directly for the more general dependent sum types. The run-time case is entirely trivial and the translation to constructors can fall back to the $\mathcal{C}$ function defined earlier, again by exploiting the well-formedness criterion.

$$\begin{array}{lcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^D]\!]\varphi & = & \langle \mathcal{E}[\![e_1]\!]\varphi, \mathcal{E}[\![e_2]\!]\varphi\rangle \\
\mathcal{E}[\![\text{fst}^D(e)]\!]\varphi & = & \text{fst}\,(\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{E}[\![\text{snd}^D(e)]\!]\varphi & = & \text{snd}(\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{T}[\![\langle e_1, e_2\rangle^D : \sum^D x : \tau_1 \,.\, \tau_2]\!]\varphi & = & \mathcal{C}[\![\sum^D x : \tau_1 \,.\, \tau_2]\!] \\
\mathcal{T}[\![\text{fst}^D(e) : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\text{snd}^D(e) : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{C}[\![\sum^D x : \tau_1 \,.\, \tau_2]\!] & = & \sum x : \mathcal{C}[\![\tau_1]\!] \,.\, \mathcal{C}[\![\tau_2]\!] \\
\mathcal{K}[\![\sum^D x : \tau_1 \,.\, \tau_2]\!] & = & \text{TYPE}
\end{array}$$

The case for specialization-time pairs is not much more involved. In fact, most of the translation is identical, since even if the information is provided in the type level, there must still be expressions of these types present.

$$\begin{array}{lcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^S]\!]\varphi & = & \langle \mathcal{E}[\![e_1]\!]\varphi, \mathcal{E}[\![e_2]\!]\varphi\rangle \\
\mathcal{E}[\![\text{fst}^S(e)]\!]\varphi & = & \text{fst}\,(\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{E}[\![\text{snd}^S(e)]\!]\varphi & = & \text{snd}(\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{T}[\![\langle e_1, e_2\rangle^S]\!]\varphi & = & \langle \mathcal{T}[\![e_1]\!]\varphi, \mathcal{T}[\![e_2]\!]\varphi\rangle \\
\mathcal{T}[\![\text{fst}^S(e)]\!]\varphi & = & \text{fst}\,(\mathcal{T}[\![e]\!]\varphi) \\
\mathcal{T}[\![\text{snd}^S(e)]\!]\varphi & = & \text{snd}(\mathcal{T}[\![e]\!]\varphi) \\
\mathcal{K}[\![\sum^S x : \tau_1 \,.\, \tau_2]\!] & = & \sum x : \mathcal{K}[\![\tau_1]\!] \,.\, \mathcal{K}[\![\tau_2]\!]
\end{array}$$

This translation differs from HMM. In HMM, every structure is a compile-time object with the first component a type (a compile-time object) and the second component a run-time value. Their translation takes advantage of this fact and maps pairs as follows:

$$\begin{array}{lcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^{HMM}]\!]\varphi & = & \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{T}[\![\langle e_1, e_2\rangle^{HMM}]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \\
\mathcal{E}[\![\text{snd}^{HMM}(e)]\!]\varphi & = & \mathcal{E}[\![e]\!]\varphi \\
\mathcal{T}[\![\text{fst}^{HMM}(e)]\!]\varphi & = & \mathcal{T}[\![e]\!]\varphi
\end{array}$$

with the remaining cases undefined.

Figure 14 shows the definition of the **type** function, extended to the new constructs.

The extended typing rules of the target language are well-known and therefore omitted.

5.5 Discussion

We claim that this kind of calculus may give rise to more powerful module languages. In addition to being able to perform type substitution (which is really the heart of HMM's module language), such a module calculus can perform arbitrary computations at compile/specialization time. The two-level annotations could be chosen in such a way that the specialization-time computation is sure to terminate, for example, by ruling out the specialization-time fixpoint operator or by replacing it with a terminating iteration construct, e.g., primitive recursion.

Another degree of flexibility comes with the ability to also express run-time modules and functors. This facility is interesting in concert with a model that distinguishes between compile-time, link-time, and run-time computation. Here, we might want to leave some module elaborations to link-time or even to run-time. Extending our calculus to a multi-level calculus [19] might yield a more accurate picture in this case.

5.6 Further work

An alternative idea that might be worth pursuing is to have a primitive type `type` of representations of types like `int`, `int -> int`, etc, as specialization-time data. In this scenario, the pair $\langle \texttt{int}, 5\rangle$ would have type $\sum \texttt{t} : \texttt{type} \,.\, \text{Set}(\texttt{t})$ where "Set" maps the representation `t` of a type to the type itself. Similar constructs are well-known in type theories with universes [37].

A more general approach might allow for annotated types in the module language instead of restricting the types that are manipulated at specialization-time to types of run-time values. Furthermore, the extended well-formedness judgement of Fig. 13 suggests that b plays the role of a kind and that the judgement should be revised to $A \vdash \tau : *^b$. Alas, we leave that for further work.

6 Related work

Type theory finds many of its uses as a basis for formal reasoning about mathematics but also to analyze and verify programs. It is the basis of a number of automatic and semi-automatic systems for automated deduction and proof construction [17, 8, 9, 37]. Some systems support sophisticated program-extraction techniques that apply to completed proofs. These extraction mechanisms are geared to remove the "logical" parts from the proofs while leaving the algorithmic parts intact. For example, Paulin-Mohring's extraction framework for the Calculus of Constructions [39] removes all applications to and abstractions over objects of

$$
\begin{array}{llll}
\text{expressions} & e & ::= & \ldots \mid \langle e, e\rangle^b \mid \mathrm{fst}^b(e) \mid \mathrm{snd}^b(e) \mid \mathrm{int} \mid e \to e \\
\text{types} & \tau & ::= & \ldots \mid \sum^b x : \tau \,.\, \tau \mid \prod^b x : \tau \,.\, \tau \mid x \mid * \\
\text{top function} & & & \mathrm{Top}(\sum^b x : \tau_1 \,.\, \tau_2) = b \\
 & & & \mathrm{Top}(\prod^b x : \tau_1 \,.\, \tau_2) = b \\
 & & & \mathrm{Top}(x) = D \\
 & & & \mathrm{Top}(*) = D
\end{array}
$$

Figure 11: Syntax of Extended Lambdamix

$$
\frac{A \text{ wft}}{A \vdash \mathrm{int}^b : b} \qquad \frac{A \text{ wft} \quad x :^b * \text{ in } A}{A \vdash x : D} \qquad \frac{A \text{ wft}}{A \vdash * : D}
$$

$$
\frac{A \vdash \tau_1 : b_1 \quad A\{x :^b \tau_1\} \vdash \tau_2 : b_2 \quad b \le b_1 \quad b \le b_2}{A \vdash \sum^b x : \tau_1 \,.\, \tau_2 : b}
$$

$$
\frac{A \vdash \tau_1 : b_1 \quad A\{x :^b \tau_1\} \vdash \tau_2 : b_2 \quad b \le b_1 \quad b \le b_2}{A \vdash \prod^b x : \tau_1 \,.\, \tau_2 : b}
$$

$$
\diamond \text{ wft} \qquad \frac{A \text{ wft} \quad A \vdash \tau : b_1 \quad b \le b_1}{A\{x :^b \tau\} \text{ wft}}
$$

Figure 12: Well-formedness rules for types and type assumptions of Extended Lambdamix

$$
\text{(b-int)}\ \frac{A \text{ wft}}{A \vdash_{\mathrm{lm}} \mathrm{int} : *}
$$

$$
\text{(b-fun)}\ \frac{A \vdash_{\mathrm{lm}} e_1 : * \quad A \vdash_{\mathrm{lm}} e_2 : *}{A \vdash_{\mathrm{lm}} e_1 \to e_2 : *}
$$

$$
\text{(b-pair)}\ \frac{A \vdash_{\mathrm{lm}} e_1 : \tau_1 \quad A\{x :^b \tau_1\} \vdash_{\mathrm{lm}} e_2 : \tau_2 \quad A \vdash \sum^b x : \tau_1 \,.\, \tau_2 : b}{A \vdash_{\mathrm{lm}} \langle e_1, e_2\rangle^b : \sum^b x : \tau_1 \,.\, \tau_2}
$$

$$
\text{(b-fst)}\ \frac{A \vdash_{\mathrm{lm}} e : \sum^b x : \tau_1 \,.\, \tau_2}{A \vdash_{\mathrm{lm}} \mathrm{fst}^b(e) : \tau_1}
$$

$$
\text{(b-snd)}\ \frac{A \vdash_{\mathrm{lm}} e : \sum^b x : \tau_1 \,.\, \tau_2}{A \vdash_{\mathrm{lm}} \mathrm{snd}^b(e) : \tau_2\{x := \mathrm{fst}^b(e)\}}
$$

$$
\text{(b-dabs)}\ \frac{A\{x :^b \tau_2\} \vdash_{\mathrm{lm}} e : \tau_1 \quad A \vdash \prod^b x : \tau_2 \,.\, \tau_1 : b}{A \vdash_{\mathrm{lm}} \lambda^b x.e : \prod^b x : \tau_2 \,.\, \tau_1}
$$

$$
\text{(b-dapp)}\ \frac{A \vdash_{\mathrm{lm}} e_1 : \prod^b x : \tau_2 \,.\, \tau_1 \quad A \vdash_{\mathrm{lm}} e_2 : \tau_2}{A \vdash_{\mathrm{lm}} e_1 @^b e_2 : \tau_1\{x := e_2\}}
$$

Figure 13: Typing rules of Extended Lambdamix

$$
\begin{array}{lcl}
\mathbf{type}(\prod^D x : \tau_2 \,.\, \tau_1)(c : \mathrm{TYPE}) & = & c \\
\mathbf{type}(\prod^S x : \tau_2 \,.\, \tau_1)(c : \prod x : \mathcal{K}[\![\tau_2]\!] \,.\, \mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!] \,.\, \prod x : \mathbf{type}(\tau_2)(t) \,.\, \mathbf{type}(\tau_1)(ct) \\
\mathbf{type}(\sum^D x : \tau_2 \,.\, \tau_1)(c : \mathrm{TYPE}) & = & c \\
\mathbf{type}(\sum^S x : \tau_2 \,.\, \tau_1)(c : \sum x : \mathcal{K}[\![\tau_2]\!] \,.\, \mathcal{K}[\![\tau_1]\!]) & = & \sum x : \mathcal{K}[\![\tau_2]\!] \,.\, \mathbf{type}(\tau_1)(\mathrm{snd}(c)) \\
\mathbf{type}(*)(c : \mathrm{kind}) & = & c \\
\mathbf{type}(x)(c : \mathrm{TYPE}) & = & c
\end{array}
$$

Figure 14: Extended **type** function

type Prop. This is different from our Lambdamix extraction function which removes the specialization-time information by reducing away all type manipulation.

Type theory has attracted some attention as a semantic metalanguage for defining the semantics of programming languages. Reynolds [41] gives a type-theoretic interpretation of Idealized Algol. Harper and Mitchell [23] do the same for a fragment of Standard ML. Harper and Stone [25] define a type theoretic semantics for Standard ML. Crary [11] defines the semantics of a higher-order kernel programming language with recursion, records, and modules using Nuprl [8]. There is no other work that defines the semantics of specialization using type theory. The closest is probably Davies's and Pfenning's work [16, 15] that connects modal and temporal logics with specialization.

There are also a number of investigations of the phase distinction in type theory [4, 5]. The idea of duplicating certain values at the type level (and their types at the kind level) is inspired from this work. Cardelli [5] calls these types and kinds *lifted* versions of the original values and types.

Harper, Mitchell, and Moggi [24] have specified the phase distinction in the ML module language extended with higher-order modules. They define and prove correct a staging transformation that separates the compile-time parts of a module from the run-time parts. Although this transformation is motivated from category theory, similar transformations are well-known in the partial evaluation community [33]. Moggi [34] has strengthened the connection between module languages and the two-level languages of the Nielsons [36] and Lambdamix [22] by showing that both have similar semantic models in terms of indexed categories. Our work may be seen as a further clue to the close connections between these two areas. Specifically our $\mathcal{E}$ and $\mathcal{T}$ are closely connected to the compile-time and run-time components defined in Harper, Mitchell, and Moggi's work [24]. Hence, the latter work is also an important contribution to partial evaluation.

In his thesis, Launchbury [30] explains a connection between partial evaluation and dependent types in his description of the type of a specializer. In his framework, the source program has type $\sum_{s\in S} D(s) \to R$, where S is the type of specialization-time values, $D(s)$ is the type of run-time values which depends on the particular value of $s \in S$, and R is the type of the result. Then the partial evaluator applied to the source program has a dependent product type: $\prod_{s\in S} D(s) \to R$. This typing seems to correspond to our translation of unfoldable (specialization-time) functions, where the additional type argument signifies the static part s and the value argument of type s only carries the dynamic part of the value.

Nielson and Nielson [36] investigate a two-level lambda calculus with slightly different properties, in particular the well-formedness criterion differs from the one considered here. It should in principle be possible to construct a type-theoretic semantics for their calculus along the same lines as done here, but we leave that for further investigation.

The whole approach is inspired by Hughes's work on type specialization [28]. Indeed, the translation to type theory presented in this paper is applicable to type specialization in a large degree. Unfortunately, the translation of run-time functions does not quite work out because of the function $\mathcal{C}[\![.]\!]$. This function is not well-defined for type specialization because type specialization does not require two-level types to be well-formed.

7 Conclusion

We have shown a type-theoretical semantics of Lambdamix specialization. The target language is a fairly powerful type theory with subkinding and some special operators. Exploiting the phase distinction in type theory, all specialization-time computation is done during type checking.

Our translation also extends to a dependently typed source language. Thus, it provides a foundation for specialization for dependently typed programming languages [1] and for module languages. Even the existing specializers should remain usable; it is only a question of supplying an appropriate binding-time analysis.

Acknowledgements

Many thanks to Gilles Barthe and John Hatcliff for discussions on specialization for languages with dependent types. Also, thanks to the referees for their comments.

References

[1] Lennart Augustsson. Cayenne—a language with dependent types. In Paul Hudak, editor, *Proc. International Conference on Functional Programming 1998*, Baltimore, USA, September 1998. ACM Press, New York.

[2] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[3] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Tim S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992.

[4] Luca Cardelli. Phase distinctions in type theory. Unpublished Manuscript, January 1988.

[5] Luca Cardelli. Types for data oriented languages. In *Conf. on Extending Database Technology*, pages 1–15. Springer-Verlag, 1988. LNCS 303.

[6] Felica Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[7] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[8] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall International, 1986.

[9] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[10] Karl Crary. Admissibility of fixpoint induction over partial types. In *CADE1998*, 1998.

[11] Karl Crary. Programming language semantics in foundational type theory (extended version). Technical Report CS TR98-1666, Cornell University, 1998.

[12] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? Available from `http://www.cs.cmu.edu/~crary/papers/1998/recmod/recmod.ps.gz`, 20 October 1998.

[13] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, volume 1110 of *Lecture Notes in Computer Science*, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[14] Olivier Danvy and Kristoffer Rose. Higher-order rewriting and partial evaluation. In *Rewriting Techniques and Applications: 9th International Conference, RTA '98*, Lecture Notes in Computer Science, Tsukuba, Japan, April 1998. Springer-Verlag.

[15] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[16] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg, Fla., January 1996. ACM Press.

[17] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[18] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[19] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.

[20] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 282–287, Nice, France, 1990. ACM Press.

[21] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[22] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, January 1991.

[23] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[24] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990. ACM Press.

[25] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrift*. MIT Press, 1998. (To appear).

[26] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[27] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[28] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [13], pages 183–215.

[29] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[30] John Launchbury. *Projection Factorisations in Partial Evaluation*, volume 1 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1991.

[31] David B. MacQueen. Using dependent types to express modular structure. In *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg, Florida, 1986. ACM.

[32] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[33] Torben Æ. Mogensen. Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 14–25, London, GB, 1989.

[34] Eugenio Moggi. A categorical account of two-level languages. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[35] Eugenio Moggi. Functor categories and two-level languages. In M. Nivat and A. Arnold, editors, *Foundations of Software Science and Computation Structures, FoSSaCS'98*, Lecture Notes in Computer Science, Lisbon, Portugal, April 1998.

[36] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[37] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf Type Theory. An Introduction*. Oxford University Press, 1990.

[38] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–364, July 1993.

$$
\begin{array}{lcl}
\mathcal{T}[\![k]\!]\varphi & = & \overline{k} \\
\mathcal{T}[\![e_1 +^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \,\overline{+}\, \mathcal{T}[\![e_2]\!]\varphi \\
\mathcal{T}[\![e_1 +^D e_2]\!]\varphi & = & \text{int} \\
\mathcal{T}[\![\text{lift } e]\!]\varphi & = & \text{int} \\
\mathcal{T}[\![x]\!]\varphi & = & \varphi(x) \\
\mathcal{T}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{T}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi) \\
\mathcal{T}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi & = & \mathcal{C}[\![\tau_2 \xrightarrow{D} \tau_1]\!] \\
\mathcal{T}[\![e_1 @^D e_2 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\text{if0}^D\, e_1\, e_2\, e_3 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\text{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\text{if0}}\,(\mathcal{T}[\![e_1]\!]\varphi)\,(\mathcal{T}[\![e_2]\!]\varphi)\,(\mathcal{T}[\![e_3]\!]\varphi) \\
\mathcal{T}[\![\text{fix}^D\, x.e : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\text{fix}^S\, x.e : \tau]\!]\varphi & = & \mu\, c : \mathcal{K}[\![\tau]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto c]
\end{array}
$$

Figure 16: Translation to constructors

$$
\begin{array}{lcl}
\mathcal{K}[\![\text{int}^S]\!] & = & \overline{\text{int}} \\
\mathcal{K}[\![\text{int}^D]\!] & = & \text{INT} \\
\mathcal{K}[\![\tau_2 \xrightarrow{S} \tau_1]\!] & = & \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!] \\
\mathcal{K}[\![\tau_2 \xrightarrow{D} \tau_1]\!] & = & \text{TYPE}
\end{array}
$$

Figure 17: Translation to kinds

$$\vdash_{\text{tt}} \diamond \; env$$

$$\frac{\vdash_{\text{tt}} \Gamma, \quad env \quad , \; \vdash_{\text{tt}} M : N \quad x \notin \Gamma, \quad}{\vdash_{\text{tt}} \Gamma, \quad , x : M \;\; env}$$

Figure 19: Environment construction

kinds	$\overline{\text{int}}$: kind
	INT : kind
	TYPE : kind
subkinding	$\overline{\text{int}} <:$ TYPE
	INT $<:$ TYPE
constructors	$\overline{i} : \overline{\text{int}}$ *(for all integers i)*
	int : INT
	$\to$: TYPE $\to$ TYPE $\to$ TYPE
	$\overline{+}$: $\overline{\text{int}} \to \overline{\text{int}} \to \overline{\text{int}}$
values	i : int *(for all integers i)*
	$\top : \overline{i}$ *(for all integers i)*
rule set	(TYPE, TYPE, TYPE)
	(kind, TYPE, TYPE)
	(kind, kind, kind)

Figure 20: Typing axioms

[39] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989. ACM Press.

[40] Didier Rémy. Extension of ML type system with a sorted equational theory on types. Rapports de Recherche 1766, INRIA, October 1992.

[41] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland.

A Notation

FV(M) denotes the set of free variables of expression M. $M\{x := N\}$ denotes the result of the capture-avoiding substitution of N for x in M. $\rho[x \mapsto t]$ denotes the updating of a finite map (an environment).

B Summary of the translations

This section contains an overview of the translations defined in Sec. 3 of this work. Figure 15 contains the expression translation, Figure 16 contains the type translation, and Figure 17 contains the kind translation.

C Typing rules of the target language

The typing rules prove judgements of the following forms:

1. Assumption Γ, is well-formed $\vdash_{\text{tt}} \Gamma, \; env$ (see Fig. 19) with the empty assumption $\diamond$.
2. In assumption Γ, expression M is an N: $\Gamma, \; \vdash_{\text{tt}} M : N$ (see Fig. 21). The formation rules are parameterized over a set of rules (Fig. 20) in the style of pure type systems [2, 3].

As customary, we write $D \to E$ instead of $\prod x : D.E$ if $x \notin \text{FV}(E)$.

The use of subkinding (see Fig. 20) merely avoids an explosion of the rule set. Without subkinding, we would have to replace (TYPE, TYPE, TYPE) by (s_1, s_2, TYPE) where $s_1, s_2 \in \{\overline{\text{int}}, \text{INT}, \text{TYPE}\}$.

$$
\begin{array}{lcl}
\mathcal{E}[\![k]\!]\varphi & = & \top \\
\mathcal{E}[\![e_1 +^S e_2]\!]\varphi & = & \top \\
\mathcal{E}[\![e_1 +^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi + \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{E}[\![\text{lift } e]\!]\varphi & = & \mathbf{choose}\ (\mathcal{T}[\![e]\!]\varphi)\ (\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{E}[\![x]\!]\varphi & = & x \\
\mathcal{E}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\lambda x : \mathbf{type}(\tau_2)(t).\mathcal{E}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{E}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi)(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi & = & \lambda x : \mathcal{C}[\![\tau_2]\!].\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{C}[\![\tau_2]\!]] \\
\mathcal{E}[\![e_1 @^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![\text{if0}^D\ e_1\ e_2\ e_3]\!]\varphi & = & \text{if0}\ (\mathcal{E}[\![e_1]\!]\varphi)\ (\mathcal{E}[\![e_2]\!]\varphi)\ (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{E}[\![\text{if0}^S\ e_1\ e_2\ e_3]\!]\varphi & = & \overline{\text{if0}}\ (\mathcal{T}[\![e_1]\!]\varphi)\ (\mathcal{E}[\![e_2]\!]\varphi)\ (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{E}[\![\text{fix}^D\ x.e : \tau]\!]\varphi & = & \text{fix}\ \lambda x : \mathcal{C}[\![\tau]\!].e \\
\mathcal{E}[\![\text{fix}^S\ x.e : \tau]\!]\varphi & = & \mu\, \lambda x : \mathbf{type}(\tau)(\mathcal{T}[\![\text{fix}^S\ x.e]\!]\varphi).\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{T}[\![\text{fix}^S\ x.e]\!]\varphi].
\end{array}
$$

Figure 15: Translation to expressions

$$
\begin{array}{lcl}
\mathbf{type} : \prod \tau : 2\text{Type}\,.\,\mathcal{K}[\![\tau]\!] \to \text{TYPE} & & \\
\mathbf{type}(\text{int}^S)(c : \overline{\text{int}}) & = & c \\
\mathbf{type}(\text{int}^D)(c : \text{INT}) & = & \text{int} \\
\mathbf{type}(\tau_2 \xrightarrow{S} \tau_1)(c : \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!]\,.\,\mathbf{type}(\tau_2)(t) \to \mathbf{type}(\tau_1)(ct) \\
\mathbf{type}(\tau_2 \xrightarrow{D} \tau_1)(c : \text{TYPE}) & = & c
\end{array}
$$

Figure 18: Mapping constructors to types

$$\frac{\vdash_{\text{tt}} \Gamma,\ env}{\Gamma, \Delta \vdash_{\text{tt}} x : A}\ x : A \text{ in } \Gamma$$

$$\frac{\vdash_{\text{tt}} \Gamma,\ env}{\Gamma \vdash_{\text{tt}} A : B}\ A : B \text{ is an axiom}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : B \quad B <: C}{\Gamma \vdash_{\text{tt}} A : C}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : s_1 \quad \Gamma, x : A \vdash_{\text{tt}} B : s_2}{\Gamma \vdash_{\text{tt}} \prod x : A\,.\,B : s_3}\ (s_1, s_2, s_3) \text{ is a rule}$$

$$\frac{\Gamma, x : A \vdash_{\text{tt}} M : B \quad \Gamma \vdash_{\text{tt}} \prod x : A\,.\,B : s}{\Gamma \vdash_{\text{tt}} \lambda x : A.M : \prod x : A\,.\,B}$$

$$\frac{\Gamma \vdash_{\text{tt}} M : \prod x : A\,.\,B \quad \Gamma \vdash_{\text{tt}} N : A}{\Gamma \vdash_{\text{tt}} MN : B\{x := N\}}$$

$$\frac{\Gamma \vdash_{\text{tt}} M : A \quad \Gamma \vdash_{\text{tt}} A' : s \quad A =_{\beta\delta} A'}{\Gamma \vdash_{\text{tt}} M : A'}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : \text{int} \quad \Gamma \vdash_{\text{tt}} M : B \quad \Gamma \vdash_{\text{tt}} N : B}{\Gamma \vdash_{\text{tt}} \text{if0}\ A\ M\ N : B}$$

$$\frac{\Gamma \vdash_{\text{tt}} T : \overline{\text{int}} \quad \Gamma \vdash_{\text{tt}} M : A \quad \Gamma \vdash_{\text{tt}} N : B}{\Gamma \vdash_{\text{tt}} \overline{\text{if0}}\ T\ M\ N : \overline{\text{if0}}\ T\ A\ B}$$

Figure 21: Typing rules

Program Adaptation based on Program Specialization

Charles Consel
University of Rennes I / Irisa
Campus Universitaire de Beaulieu
F-35042 Rennes Cedex, France
Home page: `http://www.irisa.fr/compose/consel`
E-mail: `Charles.Consel@irisa.fr`

Program development often has to address two conflicting goals: on the one hand, a program must adapt to a wide range of situations (e.g., large set of problems, varying usage patterns, and evolving hardware features); and, on the other hand, the same program must be as small and fast to execute as one dedicated to a specific situation.

In this talk, we show how program specialization can reconcile these conflicting goals by enabling programs to be highly adaptable without loss of efficiency.

We present various strategies of adaptation ranging from parameterized programs to interpreters, and show how program specialization can uniformly be used to map an adaptable program into an efficient one.

Combining Program and Data Specialization

Sandrine Chirokoff Charles Consel

Compose project, IRISA / INRIA - Université de Rennes 1,
Campus Universitaire de Beaulieu, 35042 Rennes cedex, France.
http://www.irisa.fr/compose

October 1998

Abstract

Program and data specialization have always been studied separately, although they are both aimed at processing early computations. On the one hand, program specialization encodes the result of early computations into a new program and, on the other hand, data specialization encodes the result of early computations in data structures.

In this paper, we present an extension of the Tempo specializer, which performs *both* program and data specialization. We show how these two strategies can be integrated in a unique specializer. This new kind of specializer provides the programmer with complementary strategies which widen the scope of specialization. We illustrate the benefits and limitations of these strategies and their *combination* on a variety of programs.

1 Introduction

Program and data specialization are both aimed at performing computations which depend on early values. However, they differ in the way the result of early computations are encoded: on the one hand, program specialization encodes these results in a residual program, and on the other hand, data specialization encodes these results in data structures.

More precisely, program specialization performs a computation when it only relies on early data, and inserts the textual representation of its result in the residual program when it is surrounded by computations depending on late values. In essence, it is because a new program is being constructed that early computations can be encoded in it. Furthermore, because a new program is being constructed it can be pruned, that is, the residual program only corresponds to the control flow which could not be resolved given the available data. As a consequence, program specialization optimizes the control flow since fewer control decisions need to be taken. However, because it requires a new program to be constructed, program specialization can lead to code explosion if the size of the specialization values is large. For example, this situation can occur when a loop needs to be unrolled and the number of iterations is high. Not only does code explosion cause code size problems, but it also degrades the execution time of the specialized program dramatically because of instruction cache misses.

The dual notion to specializing programs is specializing data. This strategy consists of splitting the execution of program into two phases. The first phase, called the *loader*, performs the early computations and stores their results in a data structure called a *cache*. Instead of generating a program which contains the textual representation of values, data specialization generates a program to perform the second phase: it only consists of the late computations and is *parameterized* with respect to the result of early computations, that is, the cache. The corresponding program is named the *reader*. Because the reader is parameterized with respect to the cache, it is shared by all specializations. This strategy fundamentally contrasts with program specialization because it decouples the result of early computations and the program which exploits it. As a consequence,

as the size of the specialization problem increases, only the cache parameter increases, not the program. In practice, data specialization can handle problem sizes which are far beyond the reach of program specialization, and thus opens up new opportunities as demonstrated by Knoblock and Ruf for graphics applications [7, 4]. However, data specialization, by definition, does not optimize the control flow: it is limited to performing the early computations which are expensive enough to be worth caching. Because the reader is valid for any cache it is passed, an early control decision leading to a costly early computation needs to be part of the loader as well as the reader: in the loader, it decides whether the costly computation much be cached; in the reader, the control decision determines whether the cache needs to be looked up. In fact, data specialization does not apply to programs whose bottlenecks are limited to control decisions. A typical example of this situation is interpreters for low-level languages: the instruction dispatch is the main target of specialization. For such programs, data specialization can be completely ineffective.

Perhaps the apparent difference in the nature of the opportunities addressed by program and data specialization has led researchers to study these strategies in isolation. As a consequence, no attempt has ever been made to integrate both strategies in a specializer; further, there exist no experimental data to assess the benefits and limitations of these specialization strategies.

In this paper, we study the relationship between program and data specialization with respect to their underlying concepts, their implementation techniques and their applicability. More precisely, we study program and data specialization when they are applied separately, as well as when they are combined (Section 2). Furthermore, we describe how a specializer can integrate both program and data specialization: what components are common to both strategies and what components differ. In practice, we have achieved this integration by extending a program specializer, named Tempo, with the phases needed to perform data specialization (Section 3). Finally, we assess the benefits and limitations of program and data specialization based on experimental data collected by specializing a variety of programs exposing various features (Section 4).

2 Concepts of Program and Data Specialization

In this section, the basic concepts of both program and data specialization are presented. The limitations of each strategy are identified and illustrated by an example. Finally, the combination of program and data specialization is introduced.

2.1 Program Specialization

The partial evaluation community has mainly been focusing on specialization of programs. That is, given some inputs of a program, partial evaluation generates a residual program which encodes the result of the early computations which depend on the known inputs. Although program specialization has successfully been used for a variety of applications (*e.g.,* operating systems [10, 11], scientific programs [8, 12], and compiler generation [2, 6]), it has shown some limitations. One of the most fundamental limitations is code explosion which occurs when the size of the specialization problem is large. Let us illustrate this limitation using the procedure displayed on the left-hand side of Figure 1. In this example, stat is considered static, whereas dyn and d are dynamic. Static constructs are printed in boldface. Assuming the specialization process unrolls the loop, variable i becomes static and thus the gi procedures (*i.e.,* g1, g2 and g3) can be fully evaluated. Even if the gi procedures correspond to non-expensive computations, program specialization still optimizes procedure f in that it simplifies its control flow: the loop and one of the conditionals are eliminated. A possible specialization of procedure f is presented on the right-hand side of Figure 1.

However, beyond some number of iterations, the unrolling of a loop, and the computations it enables, do not pay for the size of the resulting specialized program; this number depends on the processor features. In fact, as will be shown later, the specialized program can even get slower than the unspecialized program. The larger the size of the residual loop body, the earlier this phenomenon happens.

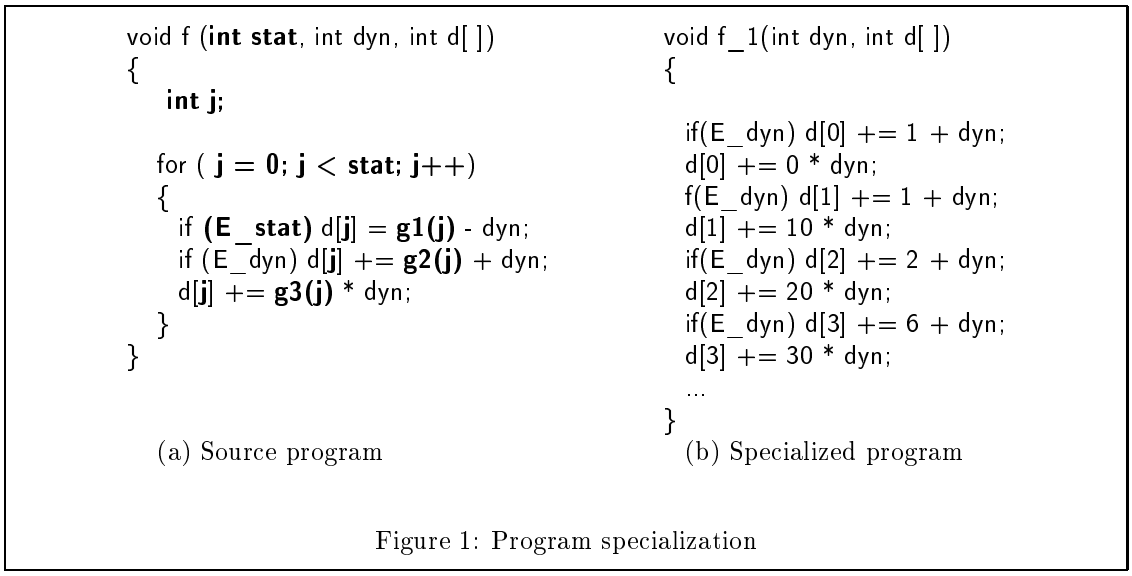

Figure 1: Program specialization

For domains like graphics and scientific computing, some applications are beyond the reach of program specialization because the specialization opportunities rely on very large data or iteration bounds which would cause code explosion if loops traversing these data were unrolled. In this situation, data specialization may apply.

2.2 Data Specialization

In late eighties, an alternative to program specialization, called data specialization, was introduced by Barzdins and Bulyonkov [1] and further explored by Malmkjær [9]. Later, Knoblock and Ruf studied data specialization for a subset of C and applied it to a graphics application [7].

Data specialization is aimed at encoding the results of early computations in data structures, not in the residual program. The execution of a program is divided into two stages: a loader first executes the early computations and saves their result in a cache. Then, a reader performs the remaining computations using the result of the early computations contained in the cache. Let us illustrate this process by an example displayed in Figure 2. On the left-hand side of this figure, a procedure f is repeatedly invoked in a loop with a first argument (c) which does not vary (and is thus considered early); its second argument, the loop index (k) varies at each iteration. Procedure f is also passed a different vector at each iteration, which is assumed to be late. Because this procedure is called repeatedly with the same first argument, data specialization can be used to perform the computations which depend on it. In this context, many computations can be performed, namely the loop test, E_stat and the invocation of the gi procedures. Of course, caching an expression assumes that its execution cost exceeds the cost of a cache reference. Measurements have shown that caching expressions which are too simple (*e.g.* a variable occurrence or simple comparisons) actually cause the resulting program to slow down.

In our example, let us assume that, like the loop test, the cost of expression E_stat is not expensive enough to be cached. If, however, the gi procedures are assumed to consist of expensive computations their invocations need to be examined as potential candidate for caching. Since the first conditional test E_stat is early, it can be put in the loader so that whenever it evaluates to true the invocation of procedure g1 can be cached; similarly, in the reader, the cache is looked up only if the conditional test evaluates to true. However, the invocation of procedure g2 cannot be cached according to Knoblock and Ruf's strategy, since it is under dynamic control and thus caching its result would amount to performing speculative evaluation [7]. Finally, the invocation of procedure g3 needs to be cached since it is unconditionally executed and its argument is early. The resulting loader and reader for procedure f are presented on the right-hand side of Figure 2, as well as their invocations.

```
extern int w[N][M];

...

for (k = 0; k < MAX; k++)
   f (c, k, w[k]);

...

void f (int stat, int dyn, int d[ ])
{
   int j;

   for (j = 0; j < stat; j++)
   {
     if (E_stat) d[j] = g1(j) - dyn;
     if (E_dyn) d[j] += g2(j) + dyn;
     d[j] += g3(j) * dyn;
   }
}
```

(a) Source program

```
extern int w[N][M];
struct data_cache { int val1;
                    int val3;} cache[MAX];
f_load (c, cache);
for (k = 0; k < MAX; k++)
  f_read (c, k, w[k], cache);
...
void f_load (stat,cache[ ])
      int stat;
      struct data_cache cache[ ];
{
  int j;
  for (j = 0; j < stat; j++) {
    if (E_stat) cache[j].val1 = g1(j);
    cache[j].val3 = g3(j);
  }
}
void f_read (stat, dyn, d[ ], cache[ ])
      int stat, dyn, d[ ];
      struct data_cache cache[ ];
{
  int j;
  for (j = 0; j < stat; j++) {
    if (E_stat) d[j] = cache[j].val1 - dyn;
    if (E_dyn) d[j] += s2(j) + dyn;
    d[j] += cache[j].val3 * dyn;
  }
}
```

(b) Specialized program

Figure 2: Data specialization

To study the limitations of data specialization consider a program where computations to be cached are not expensive enough to amortize the cost of memory reference. In our example, assume the gi procedures correspond to such computations. Then, only the control flow of procedure f remains a target for specialization.

2.3 Combining Program and Data Specialization

We have shown the benefits and limitations of both program and data specialization. The main parameters to determine which strategy fits the specialization opportunities are the cost of the early computations and the size of the specialization problem. Obviously, within the same program (or even a procedure), some fragments may require program specialization and others data specialization. As a simple example consider a procedure which consists of two nested loops. The innermost loop may require few iterations and thus allow program specialization to be applied. Whereas, the outermost loop may iterate over a vector whose size is very large; this may prevent program specialization from being applied, but not data specialization from exploiting some opportunities.

Concretely performing both program and data specialization can be done in a simple way. One approach consists of doing data specialization first, and then applying the program specializer on either the loader or the reader, or both. The idea is that code explosion may not be an issue in one of these components; as a result, program specialization can further optimize the loader or

the reader by simplifying its control flow or performing speculative specialization. For example, a reader may consist of a loop whose body is small; this situation may thus allow the loop to be unrolled without causing the residual program to be too large. Applying a program specializer to both the reader and the loader may be possible if the fragments of the program, which may cause code explosion, are made dynamic.

Alternatively, program specialization can be performed prior to data specialization. This combination requires program specialization to be applied selectively so that only fragments which do not cause code explosion are specialized. Then, the other fragments offering specialization opportunities can be processed by data specialization.

As is shown in Section 4, in practice combining both program and data specialization allows better performance than pure data specialization and prevents the performance gain from dropping as quickly as in the case of program specialization as the problem size increases.

3 Integrating Program and Data Specialization

We now present how Tempo is extended to perform data specialization. To do so let us briefly describe its features which are relevant to both data specialization and the experiments presented in the next section.

3.1 Tempo

Tempo is an off-line program specializer for C programs. As such, specialization is preceded by a preprocessing phase. This phase is aimed at computing information to guide the specialization process. The main analyses of Tempo's preprocessing phase are an alias analysis, a side-effect analyses, a binding-time analysis and an action analysis. The first two analyses are needed because of the imperative nature of the C language, whereas the binding-time analysis is typical of any off-line specializer. The action analysis is more unusual: it computes the specialization actions (*i.e.,* the program transformations) to be performed by the specialization phase.

The output of the preprocessing phase is a program annotated with specialization actions. Given some specialization values, this annotated program can be used by the specialization phase to produce a residual program at compile time, as is traditionally done by partial evaluators. In addition, Tempo can specialize a program at run time. Tempo's run-time specializer is based on templates which are efficiently compiled by standard C compilers [3, 12].

Tempo has been successfully used for a variety of applications ranging from operating systems [10, 11] to scientific programs [8, 12].

3.2 Extending Tempo with Data Specialization

Tempo includes a binding-time analysis which propagates binding times forward and backward. The forward analysis aims at determining the static computations; it propagates binding times from the definitions to the uses of variables. The backward analysis performs the same propagation in the opposite direction; when uses of a variable are both static and dynamic, its definition is annotated *static&dynamic*. This annotation indicates that the definition should be evaluated both at specialization time and run time. This process, introduced by Hornof *et al.*, allows a binding-time analysis to be more accurate; such an analysis is said to be *use sensitive* [5]. When a definition is static&dynamic and occurs in a control construct (*e.g.,* `while`), this control construct becomes static&dynamic as well. The specialized program is the code where constructs and expressions annoted static are evaluated at specialization time and its result are introduced in the residual code and where constructs and expressions annoted dynamic or static&dynamic are rebuilt in the residual code.

To perform data specialization an analysis is inserted between the forward analysis and backward analysis. In essence, this new phase identifies the *frontier terms*, that is, static terms occurring in a dynamic (or static&dynamic) context. If the cost of the frontier term is below

a given threshold (defined as a parameter of the data specializer), it is forced to dynamic (or static&dynamic).

Furthermore, because data specialization does not perform speculative evaluation, static computations which are under dynamic control are made dynamic.

Once these adjustments are done, the backward phase of the binding-time analysis then determines the final binding times of the program. Later in the process, the static computations are included in the loader and the dynamic computations in the reader; the frontier terms are cached.

The rest of our data specializer is the same as Knoblock and Ruf's.

4 Performance Evaluation

In this section, we compare the performance obtained by applying different specialization strategies on a set of programs. This set includes several scientific programs and a system program.

4.1 Overview

Machine and Compiler. The measurements presented in this paper were obtained using a Sun UltraSPARC 1 Model 170 with 448 mega bytes of main memory, running Sun-OS version 5.5.1. Times were measured using the Unix system call `getrusage` and include both "user" and "system" times.

Figure 3 displays the speedups and the size increases of compiled code obtained for different specialization strategies. For each benchmark, we give the program invariant used for specialization and an approximation of its time complexity. The code sources are included in the appendices. All the programs were compiled with `gcc -O2`. Higher degrees of optimization did not make a difference for the programs used in this experiment.

Specialization strategies. We evaluate the performance of five different specialization methods. The speedup is the ratio between the execution times of the specialized program and the original one. The size increases is the ratio between the size of the specialized program and the original one. The data displayed in Figure 3 correspond to the behavior of the following specialization strategies:

- `PS-CT`: the program is program specialized at compile time.
- `PS-RT`: the program is program specialized at run time.
- `DS`: the program is data specialized.
- `DS + PS-CT`: the program is data specialized and program specialized at compile time. The loops which manipulate the cache (for data specialization) are kept dynamic to avoid code explosion.
- `DS + PS-RT`: the program is data specialized and program specialized at run time. As in the previous strategy, the loops which manipulate the cache are kept dynamic to avoid code explosion.

Source programs. We consider a variety of source programs: a one-dimensional fast Fourier transformation (FFT), a Chebyshev approximation, a Romberg integration, a Smirnov integration, a cubic spline interpolation and a Berkeley packet filter (BPF). Given the specialization strategies available, these programs can be classified as follows.

Control flow intensive. A program which mainly exposes control flow computations; data flow computations are inexpensive. In this case, program specialization can improve performance whereas data specialization does not because there is no expensive calculations to cache.

Data flow intensive. A program which is only based on expensive data flow computations. As a result, program specialization at compile time as well as data specialization can improve the performance of such program.

Control and data flow intensive. A program which contains both control flow computations and expensive data flow computations. Such program is a good candidate for program specialization at compile time when applied to small values, and well-suited for data specialization when applied to large values.

We now analyze the performance of five specialization methods in turn on the benchmark programs.

4.2 Results

Data specialization can be executed at compile time or at run time. At run time, the loader of the cache is executed before the execution of the specialized program, while at compile time, the cache is constructed before the compilation. The cache is then used by the specialized program during the execution. For all programs, data specialization yields a greater speedup than program specialization at run time. The combination of these two specialization strategies does not make a better result.

In this section, we characterize different opportunities of specialization to illustrate our method in the three categories of program.

4.2.1 Program Specialization

We analyze two programs where performance is better with program specialization: the Berkeley packet filter (BPF), which interprets a packet with respect to an interpreter program, and the cubic spline interpolation, which approximates a function using a third degree polynomial equation.

Characteristics: For the BPF, the program consists exclusively of conditionals whose tests and branches contain inexpensive expressions. For the cubic spline interpolation, the program consists of small loops whose small body can be evaluated in part. Concretely, a program which mainly depends on the control flow graph and whose leaves contain few calculations but partially reducible, is a good candidate for program specialization. By program specialization, the control flow graph is reduced and some calculations are eliminated. Since there is no static calculation expensive enough to be efficiently cached by data specialization, the specialized program is mostly the same as the original one. For this kind of programs, only program specialization gives significant improvements: it reduces the control flow graph and it produces a small specialized program.

Applications: The BPF (Appendix F) is specialized with respect to a program (of size n). It mainly consists of the conditionals; its time complexity is linear in the size of the program and it does not contain expensive data computations. As the program does not contain any loop, the size of the specialized program is mostly the same as the original one. In Figure 3-F, program specialization at compile time and at run time yields a good speedup, whereas data specialization only improves performance marginally. The combination of program and data specialization does not improve the performance further.

The cubic spline interpolation (Appendix E) is specialized with respect to the number of points (n) and their x-coordinates. It contains three singly nested loops; its time complexity is $O(n)$. In the first two loops, more than half of the computations of each body can be completely evaluated or cached by specialization, including real multiplications and divisions. Nevertheless, there is no expensive calculation to cache, and data specialization does not improve performance significantly. The unrolled loop does not really increase the code size because of the small complexity of the program and the small body of the loop. As a consequence, for each number of points n, the

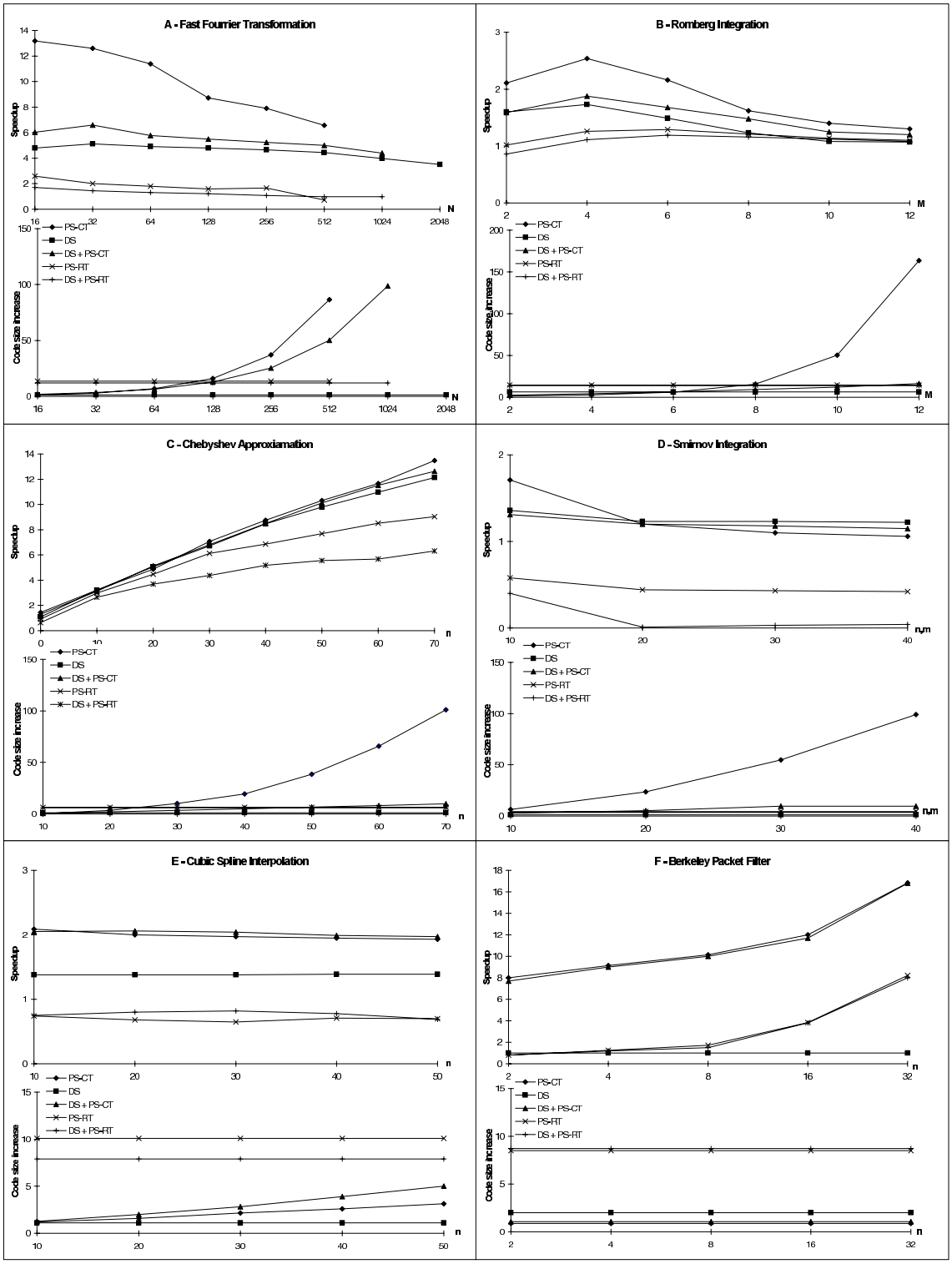

Figure 3: Program, data and combined specializations

speedup of each specialization barely changes. In Figure 3-E, program specialization at compile time produces a good speedup, whereas program specialization at run time does not improve performance. Data specialization obtains a minor speedup because the cached calculations are not expensive.

4.2.2 Program Specialization or Data Specialization

We now analyze two programs where performance is identical to program specialization or data specialization: the polynomial Chebyshev, which approximates a continuous function in a known interval, and the Smirnov integration, which approximates the integral of a function on an interval using estimations.

Characteristics: These two programs only contain loops and expensive calculations in doubly nested loops. As for the cubic spline interpolation (Section 4.2.1), more than half of the computations of each body loop can be completely evaluated or cached by specialization. In contrast with cubic spline interpolation, the static calculations in Chebyshev and Smirnov are very expensive and allow data specialization to yield major improvements. For the combined specialization, data specialization is applied to the innermost loop and program specialization is applied to the rest of the program. For this kind of programs, program and data specialization both give significant improvements. However, for the same speedup, the code size of the program produced by program specialization is a hundred times larger than the specialized program using data specialization.

Applications: The Chebyshev approximation (Appendix C) is specialized with respect to the degree (n) of the generated polynomial. This program contains two calls to the trigonometric function *cos*: one of them in a singly nested loop and the other call in a doubly nested loop. Since this program mainly consists of data flow computations, program specialization and data specialization obtain similar speedups (see Figure 3-C).

The Smirnov integration (Appendix D) is specialized with respect to the number of iterations (n, m). The program contains a call to the function *fabs* which returns the absolute value of its parameter. This function is contained in a doubly nested loop and the time complexity of this program is $O(m^n)$. As in the case of Chebyshev, program and data specialization produce similar speedups (see Figure 3-D).

4.2.3 Combining Program Specialization and Data Specialization

Finally, we analyze two programs where performance improves using program specialization when values are small, and data specialization when values are large: the FFT and the Romberg integration. The FFT converts data from the time domain to a frequency domain. The Romberg integration approximates the integral of a function on an interval using estimations.

Characteristics: These two programs contain several loops and expensive data flow computations in doubly nested loops; however more than half of the computations of each loop body cannot be evaluated. Beyond some number of iterations, when the program specialization unrolls these loops, it increases the code size of the specialized program and then degrades performance. The specialized program becomes slower because of its code size. Furthermore, beyond some problem size, the specialization process cannot produce the program because of its size. In contrast, data specialization only caches the expensive calculations, does not unroll loops, and improves performance. The result is that the code size of the program produced by program specialization is a hundred times larger than the specialized program using data specialization, for a speedup gain of 20%. The combined specialization delays the occurrence of code explosion. Data specialization is applied to the innermost loop, which contains the cache computations, and program specialization is applied to the rest of the program.

Applications: The FFT (Appendix A) is specialized with respect to the number of data points (N). It contains ten loops with several degrees of nesting. One of these loops, with complexity $O(N^2)$, contains four calls to trigonometric functions, which can be evaluated by program specialization or cached by data specialization. Due to the elimination of these expensive library calls, program specialization and data specialization produce significant speedups (see Figure 3-A). However, in the case of program specialization, code unrolling degrade performance. In contrast, data specialization produces a stable speedup regardless of the number of data points. When N is smaller than 512, data specialization does not obtain a better result in comparison to program specialization. However, when N is greater than 512, program specialization becomes impossible to apply because of the specialization time and the size of the residual code. In this situation, data specialization still gives better performance than the unspecialized program. Because this program also contains some conditionals, combined specialization, where the innermost loop is not unrolled, improves performance better than data specialization alone.

The Romberg integration (Appendix B) is specialized with respect to the number of iterations (M) used in the approximation. The Romberg integration contains two calls to the costly function *intpow*. It is called twice: once in a singly nested loop and another time in a doubly nested loop. Because both specialization strategies eliminate these expensive library calls, the speedup is consequently good. As for FFT, loop unrolling causes the program specialization speedup to decrease, whereas the data specialization speedup still remains the same, even when M increases (Figure 3-B).

5 Conclusion

We have integrated program and data specialization in a specializer named Tempo. Importantly, data specialization can re-use most of the phases of an off-line program specializer.

Because Tempo now offers both program and data specialization, we have experimentally compared both strategies and their combination. This evaluation shows that, on the one hand program specialization typically gives better speed-up than data specialization for small problem size. However, as the problem size increases, the residual program may become very large and often slower than the unspecialized program. On the other hand, data specialization can handle large problem size without much performance degradation. This strategy can, however, be ineffective if the program to be specialized mainly consists of control flow computations. The combination of both program and data specialization is promising: it can produce a residual program more efficient than with data specialization alone, without dropping in performance as dramatically as program specialization, as the problem size increases.

Acknowledgments

We thank Renaud Marlet for thoughtful comments on earlier versions of this paper, as well as the Compose group for stimulating discussions.

A substantial amount of the research reported in this paper builds on work done by the authors with Scott Thibault on Berkeley packet filter and Julia Lawall on Fast Fourier Transformation.

References

[1] G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk, 1988.

[2] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[3] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[4] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[5] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[6] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[7] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5). Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[8] J.L. Lawall. Faster Fourier transforms via automatic program specialization. Publication interne 1192, IRISA, Rennes, France, May 1998.

[9] K. Malmkjaer. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.

[10] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[11] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125, Amsterdam, The Netherlands, June 1997. ACM Press.

[12] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

A Fast Fourrier Transformation

```
#define PI  3.14159265358979323846

int fft(np, x, y)
   int np ;
   double x[2];
   double y[2];
{
   double *px,*py;
   int i,j,k,m,n;
   int i0,i1,i2,i3;
   int is,id;
   int n1,n2,n4;
   double  a,e,a3;
   double  cc1,ss1,cc3,ss3;
   double  r1,r2;
   double  s1,s2,s3;
   double  xt;

   px = x - 1;
   py = y - 1;
   i = 2;
   m = 1;
   while (i < np)
   {
      i = i+i;
      m = m+1;
   };
   n = i;
   if (n != np)
   {
      for (i = np+1; i <= n; i++)
      {
         *(px + i) = 0.0F;
         *(py + i) = 0.0F;
      };
      printf("\nuse %d point fft",n);
   }

   n2 = n+n;
   for (k = 1;  k <= m-1; k++ )
   {
      n2 = n2 / 2;
      n4 = n2 / 4;
      e = 2.0F * (float)PI / n2;
      a = 0.0F;
      for (j = 1; j<= n4 ; j++)
      {
         a3 = 3.0F*a;
         cc1 = cos(a);
         ss1 = sin(a);
         cc3 = cos(a3);
         ss3 = sin(a3);
         a = j*e;
         is = j;
         id = 2*n2;
         while ( is < n )
         {
            for (i0 = is; i0 <= n-1;)
            {
               i1 = i0 + n4;
               i2 = i1 + n4;
               i3 = i2 + n4;

               r1 = *(px+i0) - *(px+i2);
               *(px+i0) = *(px+i0) + *(px+i2);
               r2 = *(px+i1) - *(px+i3);
               *(px+i1) = *(px+i1) + *(px+i3);
               s1 = *(py+i0) - *(py+i2);
               *(py+i0) = *(py+i0) + *(py+i2);
               s2 = *(py+i1) - *(py+i3);
               *(py+i1) = *(py+i1) + *(py+i3);

               s3 = r1 - s2;
               r1 = r1 + s2;
               s2 = r2 - s1;
               r2 = r2 + s1;

               *(px+i2) = r1*cc1 - s2*ss1;
               *(py+i2) = -s2*cc1 - r1*ss1;
               *(px+i3) = s3*cc3 + r2*ss3;
               *(py+i3) = r2*cc3 - s3*ss3;

               i0 = i0 + id;
            }
            is = 2*id - n2 + j;
            id = 4*id;
         }
      }
   }

/*
-------Last stage, length=2 butterfly---------
*/
   is = 1;
   id = 4;
   while ( is < n)
   {
      for (i0 = is; i0 <= n;)
      {
         i1 = i0 + 1;

         r1 = *(px+i0);
         *(px+i0) = r1 + *(px+i1);
         *(px+i1) = r1 - *(px+i1);

         r1 = *(py+i0);
         *(py+i0) = r1 + *(py+i1);
         *(py+i1) = r1 - *(py+i1);

         i0 = i0 + id;
      }
      is = 2*id - 1;
      id = 4 * id;
   }

/*
----------Bit reverse counter------------
*/
   j = 1;
   n1 = n - 1;
   for (i = 1; i <= n1; i++)
   {
      if (i < j)
      {
         xt = *(px+j);
         *(px+j) = *(px+i);
         *(px+i) = xt;

         xt = *(py+j);
         *(py+j) = *(py+i);
         *(py+i) = xt;
      }
      k = n / 2;
      while (k < j)
      {
         j = j - k;
         k = k / 2;
      }
      j = j + k;
   }
   return(n);

}
```

B Romberg Integration

```
void romberg(float (*r)[50], float a, float b, int M)
{
  int n, m, i, max;
  float h, s;

  h = b - a;
  r[0][0] = (f(a) + f(b)) * h / 2.0;

  for (n = 1; n <= M; n++) {
    h = h / 2.0;
    s = 0.0;

    max = int_pow(2, n - 1);
    for (i = 1; i <= max; i++) {
      s = s + f(a + (float)(2.0 * i - 1) * h);
    }

    r[n][0] = r[n-1][0]/2.0 + h * s;

    for(m = 1; m <= n; m++) {
      r[n][m] = r[n][m-1] + (float)(1.0/(int_pow(4,m)-1))
                * (r[n][m-1] - r[n-1][m-1]);
    }
  }
}

int int_pow(int base, int expon)
{
  int accum = 1;

  while (expon > 0) {
    accum *= base;
    expon--;
  }

  return(accum);
}
```

C Chebyshev Approximation

```
#define MAX1 100
#define PI  3.14159265358979323846

void cheb(float c[MAX1], int n, float xa, float xb)
{
  int k, j;
  float xm, xp, sm;
  float f[MAX1];

  xp = (xb + xa) / 2;
  xm = (xb - xa) / 2;

  for(k = 1; k <= n; k++) {
    f[k] = func(xp + xm * cos(PI * (k - 0.5) / n));
  }

  for(j = 0; j <= n-1; j++) {
    sm = 0.0;
    for(k = 1; k <= n; k++) {
      sm = sm + f[k] * cos(PI * j * (k - 0.5) / n);
    }
    c[j] = (2.0 / n) * sm;
  }
  return;
}
```

D Smirnov Integration

```
double smirnov(int m, int n, double D, double *u)
  {
    double c;
    double W;
    int i;
    int j;
    double temp;

    c = (double)(m * n) * D - 1.0 ;
    for (j = 0; j <= n; j++)
      {
        u[j] = 1.0;
        if (c < (double)(m * j))
          {
            u[j] = 0.0;
          }
      }
    for (i = 1; i <= m; i++)
      {
        double *suif_tmp0;

        W = (double)i / (double)(i + n) ;
        suif_tmp0 = u;
        *suif_tmp0 = *suif_tmp0 * W;
        if (c < (double)(n * i))
          {
            *u = 0.0;
          }
        for (j = 1; j <= n; j++)
          {
            u[j] = W * u[j] + u[j - 1]+1;
            temp=(double)fabs(n * i - m * j);
            if (c < temp)
              {
                u[j] = 0.0;
              }
          }
      }
    return 1.0 - u[n];
  }
```

E Cubic Spline Interpolation

```
#define MAX 100

void csi(int n, float x[MAX], float y[MAX], float z[MAX])
{
  float h[MAX],b[MAX],u[MAX],v[MAX];

  int i;

  for (i = 0; i<= n-1; i=i+1){
    h[i] = x[i+1]-x[i];
    b[i] = (6/h[i])*(y[i+1]-y[i]);
  }

  u[1] = 2*(h[0]+h[1]);
  v[1] = b[1]-b[0];
  for (i = 2; i <= n-1; i=i+1){
    u[i] = 2*(h[i]+h[i-1])-h[i-1]*h[i-1]/u[i-1];
    v[i] = b[i]-b[i-1]-h[i-1]*v[i-1]/u[i-1];
  }
  z[n] = 0;

  i=n-1;
  if ( i>=1 )
    do {
      z[i] = (v[i]-h[i]*z[i+1])/u[i];
      i=i-1;
    } while ( i>=1 );
  z[0] = 0;
}
```

F Berkeley Packet Filter

```
u_int bpf(pc, p, wirelen, buflen)
   register struct bpf_insn *pc;
   register u_char *p;
   u_int wirelen;
   register u_int buflen;
{
   register u_int32 A, X;
   register int k;
   int32 mem[BPF_MEMWORDS];
   u_char returned=FALSE;

   if (pc == 0)
      /*
       * No filter means accept all.
       */
      return (u_int)-1;
   A = 0;
   X = 0;
   --pc;
   while (!returned) {
      ++pc;
      switch (pc->code) {

      default:
         returned=TRUE;
         abort();
         break;

      case BPF_RET|BPF_K:
         returned=TRUE;
         return (u_int)pc->k;
         break;

      case BPF_RET|BPF_A:
         returned=TRUE;
         return (u_int)A;
         break;

      case BPF_LD|BPF_W|BPF_ABS:
         k = pc->k;
         if (k + sizeof(int32) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_LONG(&p[k]);
         break;

      case BPF_LD|BPF_H|BPF_ABS:
         k = pc->k;
         if (k + sizeof(short) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_SHORT(&p[k]);
         break;

      case BPF_LD|BPF_B|BPF_ABS:
         k = pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         A = p[k];
         break;

      case BPF_LD|BPF_W|BPF_LEN:
         A = wirelen;
         break;

      case BPF_LDX|BPF_W|BPF_LEN:
         X = wirelen;
         break;

      case BPF_LD|BPF_W|BPF_IND:
         k = X + pc->k;
         if (k + sizeof(int32) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_LONG(&p[k]);
         break;

      case BPF_LD|BPF_H|BPF_IND:
         k = X + pc->k;
         if (k + sizeof(short) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_SHORT(&p[k]);
         break;

      case BPF_LD|BPF_B|BPF_IND:
         k = X + pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         A = p[k];
         break;

      case BPF_LDX|BPF_MSH|BPF_B:
         k = pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         X = (p[pc->k] & 0xf) << 2;
         break;

      case BPF_LD|BPF_IMM:
         A = pc->k;
         break;

      case BPF_LDX|BPF_IMM:
         X = pc->k;
         break;

      case BPF_LD|BPF_MEM:
         A = mem[pc->k];
         break;

      case BPF_LDX|BPF_MEM:
         X = mem[pc->k];
         break;

      case BPF_ST:
         mem[pc->k] = A;
         break;

      case BPF_STX:
         mem[pc->k] = X;
         break;

      case BPF_JMP|BPF_JA:
         pc += pc->k;
         break;

      case BPF_JMP|BPF_JGT|BPF_K:
         pc += (A > pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGE|BPF_K:
         pc += (A >= pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JEQ|BPF_K:
         pc += (A == pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JSET|BPF_K:
         pc += (A & pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGT|BPF_X:
         pc += (A > X) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGE|BPF_X:
```

```
   pc += (A >= X) ? pc->jt : pc->jf;
   break;

case BPF_JMP|BPF_JEQ|BPF_X:
   pc += (A == X) ? pc->jt : pc->jf;
   break;

case BPF_JMP|BPF_JSET|BPF_X:
   pc += (A & X) ? pc->jt : pc->jf;
   break;

case BPF_ALU|BPF_ADD|BPF_X:
   A += X;
   break;

case BPF_ALU|BPF_SUB|BPF_X:
   A -= X;
   break;

case BPF_ALU|BPF_MUL|BPF_X:
   A *= X;
   break;

case BPF_ALU|BPF_DIV|BPF_X:
        if (X == 0) {
      returned=TRUE;
      return 0;
   }
   A /= X;
   break;

case BPF_ALU|BPF_AND|BPF_X:
   A &= X;
   break;

case BPF_ALU|BPF_OR|BPF_X:
   A |= X;
   break;

case BPF_ALU|BPF_LSH|BPF_X:
   A <<= X;
   break;

case BPF_ALU|BPF_RSH|BPF_X:
   A >>= X;
   break;

case BPF_ALU|BPF_ADD|BPF_K:
   A += pc->k;
   break;

case BPF_ALU|BPF_SUB|BPF_K:
   A -= pc->k;
   break;

case BPF_ALU|BPF_MUL|BPF_K:
   A *= pc->k;
   break;

case BPF_ALU|BPF_DIV|BPF_K:
   A /= pc->k;
   break;

case BPF_ALU|BPF_AND|BPF_K:
   A &= pc->k;
   break;

case BPF_ALU|BPF_OR|BPF_K:
   A |= pc->k;
   break;

case BPF_ALU|BPF_LSH|BPF_K:
   A <<= pc->k;
   break;

case BPF_ALU|BPF_RSH|BPF_K:
   A >>= pc->k;
   break;

case BPF_ALU|BPF_NEG:
   A = -A;
   break;

case BPF_MISC|BPF_TAX:
   X = A;
   break;

case BPF_MISC|BPF_TXA:
   A = X;
   break;
   }
  }
  return 0;
}
```

Certifying Compilation and Run-time Code Generation

Luke Hornof Trevor Jim

Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19103
{hornof,tjim}@cis.upenn.edu

Abstract

A certifying compiler takes a source language program and produces object code, as well as a "certificate" that can be used to verify that the object code satisfies desirable properties, such as type safety and memory safety. Certifying compilation helps to increase both compiler robustness and program safety. Compiler robustness is improved since some compiler errors can be caught by checking the object code against the certificate immediately after compilation. Program safety is improved because the object code and certificate alone are sufficient to establish safety: even if the object code and certificate are produced on an unknown machine by an unknown compiler and sent over an untrusted network, safe execution is guaranteed as long as the code and certificate pass the verifier.

Existing work in certifying compilation has addressed statically generated code. In this paper, we extend this to code generated at run time. Our goal is to combine certifying compilation with run-time code generation to produce programs that are both verifiably safe and extremely fast. To achieve this goal, we present two new languages with explicit run-time code generation constructs: Cyclone, a type safe dialect of C, and TAL/T, a type safe assembly language. We have designed and implemented a system that translates a safe C program into Cyclone, which is then compiled to TAL/T, and finally assembled into executable object code. This paper focuses on our overall approach and the front end of our system; details about TAL/T will appear in a subsequent paper.

1 Introduction

1.1 Run-time specialization

Specialization is a program transformation that optimizes a program with respect to invariants. This technique has been shown to give dramatic speedups on a wide range of applications, including aircraft crew planning programs, image shaders, and operating systems [4, 11, 17]. *Run-time* specialization exploits invariants that become available during the execution of a program, generating optimized code on the fly. Opportunities for run-time specialization occur when dynamically changing values remain invariant for a period of time. For example, networking software can be specialized to a particular TCP connection or multicast tree.

Run-time code generation is tricky. It is hard to correctly write and reason about code that generates code; it is not obvious how to optimize or debug a program that has yet to be generated. Early examples of run-time code generation include self-modifying code, and ad hoc code generators written by hand with a specific function in mind. These approaches proved complicated and error prone [14].

More recent work has applied advanced programming language techniques to the problem. New source languages have been designed to facilitate run-time code generation by providing the programmer with high-level constructs and having the compiler implement the low-level details [15, 21, 22]. Program transformations based on static analyses are now capable of automatically translating a normal program into a run-time code generating program [6, 10, 12]. And type systems can check run-time code generating programs at compile time, ensuring that certain bugs will not occur at run time (provided the compiler is correct) [22, 25].

These techniques make it easier for programmers to use run-time code generation, but they do not address the concerns of the compiler writer or end user. The compiler writer still needs to implement a correct compiler—not easy even for a language without run-time code generation. The end user would like some assurance that executables will not crash their machine, even if the programs generate code and jump to it—behavior that usually provokes suspicion in security-concious users. We will address both of these concerns through another programming language technique, *certifying compilation*.

1.2 Certifying compilation

A certifying compiler takes a source language program and produces object code and a "certificate" that may help to show that the object code satisfies certain desirable properties [16, 18]. A separate component called the *verifier* examines the object code and certificate and determines whether the object code actually satisfies the properties. A wide range of properties can be verified, including memory safety (unallocated portions of memory are not accessed), control safety (code is entered only at valid entry points), and various security properties (e.g., highly classified data does appear on low security channels). Often, these properties are corollaries of type safety in an appropriate type system for the object code.

In this paper we will describe a certifying compiler for *Cyclone*, a high-level language that supports run-time code generation. Cyclone is compiled into *TAL/T*, an assembly language that supports run-time code generation. Cyclone and TAL/T are both type safe; the certificates of our system are the type annotations of the TAL/T output, and the verifier is the TAL/T type checker.

As compiler writers, we were motivated to implement Cyclone as a certifying compiler because we believe the approach enhances compiler correctness. For example, we were forced to develop a type system and operational semantics for TAL/T. This provides a formal framework for reasoning about object code that generates object code at run time. Eventually, we hope to prove that the compiler transforms type correct source programs into type correct object programs, an important step towards proving correctness for the compiler. In the meantime, we use the verifier to type check the output of the compiler, so that we get immediate feedback when our compiler introduces type errors. As others have noted [23, 24], this helps to identify and correct compiler bugs quickly.

We also wanted a certifying compiler to address the safety concerns of end users. In our system, type safety only depends on the certificate and the object code, and not on the method by which they are produced. Thus the end user does not have to rely on the programmer or the Cyclone compiler to ensure safety. This makes our system usable as the basis of security-critical applications like active networks and mobile code systems.

1.3 The Cyclone compiler

The Cyclone compiler is built on two existing systems, the Tempo specializer [19] and the Popcorn certifying compiler [16]. It has three phases, shown in Fig. 1.

The first phase transforms a type safe C program into a Cyclone program that uses run-time code generation. It starts by applying the static analyses of the Tempo system to a C program and context information that specifies which function arguments are invariant. The Tempo front end produces an action-annotated program. We added an additional pass to translate the action-annotated program into a Cyclone run-time specializer.

The second phase verifies that the Cyclone program is type safe, and then compiles it into TAL/T. To do this, we modified the Popcorn compiler of Morrisett et al.; Popcorn compiles a type safe dialect of C into TAL, a typed assembly language. We extended the front end of Popcorn to handle Cyclone programs, and modified its back end so that it outputs TAL/T. TAL/T is TAL extended with instructions for manipulating *templates*, code fragments parameterized by holes, and their corresponding types. This compilation phase not only transforms high-level Cyclone constructs into low-level assembly instructions, but also transforms Cyclone types into TAL/T types.

The third phase first verifies the type safety of the TAL/T program. The type system of TAL/T ensures that the templates are combined correctly and that holes are filled in correctly. This paper describes our overall approach and the front end in detail, but the details of TAL/T will appear in a subsequent paper. Finally, the TAL/T program is assembled and linked into an executable.

This three phase design offers a very flexible user interface since it allows programs to be written in C, Cyclone, or TAL/T. In the simplest case, the user can simply write a C program (or reuse an existing program) and allow the system to handle the rest. If the user desires more explicit control over the code generation process, he may write (or modify) a Cyclone program. If very fine-grain control is desired, the user can fine-tune a TAL/T program produced by Cyclone, or can write one by hand. Note that, since verification is performed at the TAL/T level, the same program safety properties are guaranteed in all three of these cases.

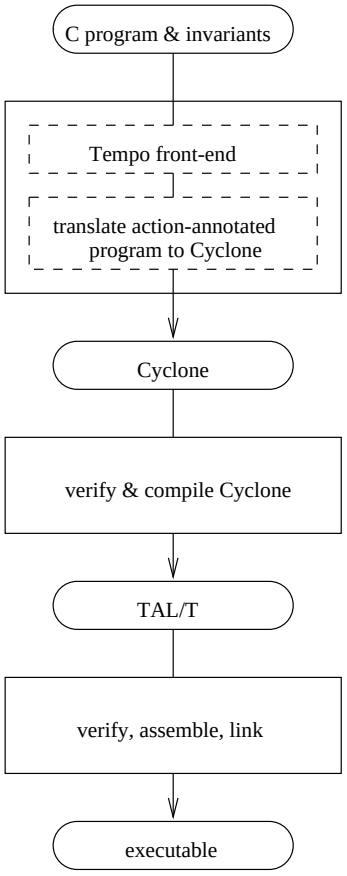

Figure 1: Overview of the Cyclone compiler

1.4 Example

We now present an example that illustrates run-time code generation and the phases of our Cyclone compiler. Fig. 2 shows a modular exponentiation function, `mexp`, written in standard C. Its arguments are a base value, an exponent, and a modulus. Modular exponentiation is often used in cryptography; when the same key is used to encrypt or decrypt several messages, the function is called repeatedly with the same exponent and modulus. Thus `mexp` can benefit from specialization.

To specialize the function with respect to a given exponent and modulus, the user indicates that the two arguments are *invariant*: the function will be called repeatedly with the same values for the invariant arguments. In Fig. 2, invariant arguments are shown in *italics*. A static analysis propagates this information throughout the program, producing an *action-annotated* program. Actions describe how each language construct will be treated during specialization. Constructs that depend only on invariants can be evaluated during specialization; these constructs are displayed in *italics* in the second part of the figure.

To understand how run-time specialization works, it is

C code (invariant arguments in *italics*)

```
int mexp(int base, int exp, int mod)
{
  int s, t, u;

  s = 1; t = base; u = exp;

  while (u != 0) {
    if ((u&1) != 0)
      s = (s*t) % mod;
    t = (t*t) % mod;
    u >>= 1;
  }
  return(s);
}
```

Action-annotated code (*italicized* constructs can be evaluated)

```
int mexp(int base, int exp, int mod)
{
  int s, t, u;

  s = 1; t = base; u = exp;

  while (u != 0) {
    if ((u & 1) != 0)
      s = (s*t) % mod;
    t = (t*t) % mod;
    u >>= 1;
  }
  return(s);
}
```

Specialized source code (`exp = 10`, `mod = 1234`)

```
int mexp_sp(int base)
{
  int s, t;

  s = 1; t = base;

  t = (t*t) % 1234;
  s = (s*t) % 1234;
  t = (t*t) % 1234;
  t = (t*t) % 1234;
  s = (s*t) % 1234;
  t = (t*t) % 1234;

  return(s);
}
```

Figure 2: Specialization at the source level

helpful to first consider how specialization could be achieved entirely within the source language. In our example, the specialized function `mexp_sp` of Fig. 2 is obtained from the action-annotated `mexp` when the exponent is 10 and the modulus is 1234. Italicized constructs of `mexp`, like the while loop, can be evaluated (note that the loop test depends only on the known arguments). Non-italicized constructs of `mexp` show up in the source code of `mexp_sp`. These constructs can only be evaluated when `mexp_sp` is called, because they depend on the unknown argument.

We can think of `mexp_sp` as being constructed by cutting and pasting together fragments of the source code of `mexp`. These fragments, or *templates*, are a central idea we used in designing Cyclone. Cyclone is a type safe dialect of C extended with four constructs that manipulate templates: `codegen`, `cut`, `splice`, and `fill`. Using these constructs, it is possible to write a Cyclone function that generates a specialized version of `mexp` at run time.

```
int (int) mexp_gen(int exp, int mod)
{
  int u;

  u = exp;

  return codegen(
    int mexp_sp(int base) {
    int s, t;

    s = 1; t = base;

    cut
      while (u != 0) {
        if ((u & 1) != 0)
          splice s = (s*t) % fill(mod);
        splice t = (t*t) % fill(mod);
        u >>= 1;
      }
    return(s);
    });
}
```

Figure 3: A run-time specializer written in Cyclone

In fact, our system can automatically generate a Cyclone run-time specializer from an action-annotated program. Fig. 3 shows the Cyclone specializer produced from the action-annotated modular exponentiation function of Fig. 2. The function `mexp_gen` takes the two invariant arguments of the original `mexp` function and returns the function `mexp_sp`, a version of `mexp` specialized to those arguments. In the figure we have italicized code that will be evaluated when `mexp_gen` is called. Non-italicized code is template code that will be manipulated by `mexp_gen` to produce the specialized function. The template code will only be evaluated when the specialized function is itself called.

In our example, the `codegen` expression begins the code generation process by allocating a region in memory for the new function `mexp_sp`, and copying the first template into the region. This template includes the declarations of the function, its argument `base`, and local variables `s` and `t`, and also the initial assignments to `s` and `t`. Recall that this template code is *not* evaluated during the code generation process, but merely manipulated.

The `cut` statement marks the end of the template and introduces code (italicized) that will be evaluated during code generation: namely, the while loop. The while test and body, including the conditional statement, `splice` statements, and shift/assignment statement, will all be evaluated. After the while loop finishes, the template following the `cut` statement (containing `return(s)`) will be added to the code generation region.

Evaluating a `splice` statement causes a template to be appended to the code generation region. In our example, each time the first `splice` statement is executed, an assignment to `s` is appended. Similarly, each time the second `splice` statement is executed, an assignment to `t` is appended. The effect of the while loop is thus to add some number of assignment statements to the code of `mexp_sp`; exactly how many, and which ones, is determined by the arguments of `mexp_gen`.

A `fill` expression can be used within a template, and it

marks a *hole* in the template. When `fill(`e`)` is encountered in a template, e is evaluated at code generation time to a value, which is then used to fill the hole in the template. In our example, `fill` is used to insert the known modulus value into the assignment statements.

After code generation is complete, the newly generated function `mexp_sp` is returned as the result of `codegen`. It takes the one remaining argument of `mexp` to compute its result.

Cyclone programs can be evaluated symbolically to produce specialized source programs, like the one in Fig. 2; this is the basis of the formal operational semantics we give in the appendix. In our implementation, however, we compile Cyclone source code to object code, and we compile source templates into object templates. The Cyclone object code then manipulates object templates directly.

Our object code, TAL/T, is an extension of TAL with instructions for manipulating object templates. Most of the TAL/T instructions are x86 machine instructions; the new template instructions are `CGSTART`, `CGDUMP`, `CGFILL`, `CGHOLE`, `TEMPLATE_START`, and `TEMPLATE_END`. For example, the Cyclone program in Fig. 3 is compiled into the TAL/T program shown in Fig. 4. (We omitted some instructions to save space, and added source code fragments in comments to aid readability.)

The beginning of `_mexp_gen` contains x86 instructions for adding the local variable `u` to the stack and assigning it the value of the argument `exp`. Next, `CGSTART` is used to dynamically allocate a code generation region, and the first template is dumped (copied) into the region with the `CGDUMP` instruction. Next, the body of the loop is unrolled. Each Cyclone `splice` statement is compiled into a `CGDUMP` instruction, followed by instructions for computing hole values and a `CGFILL` instruction for filling in the hole. At the end of the `_mexp_gen` function, a final `CGDUMP` instruction outputs code for the last template.

Next comes the code for each of the four templates. The first template allocates stack space for local variables `s` and `t` and assigns values to them. The second and third templates come from the statements contained within the Cyclone `splice` instructions, i.e., the multiplications, mods, and assignments. The final template contains the code for `return(s)`. Each `CGHOLE` instruction introduces a placeholder inside a template, filled in during specialization as described above.

1.5 Summary

We designed a system for performing type safe run-time code generation. It has the following parts:

- C to action-annotated program translation
- Action-annotated program to Cyclone translation
- Cyclone language design
- Cyclone type system
- Cyclone verifier
- Cyclone to TAL/T compiler
- TAL/T language design
- TAL/T type system
- TAL/T verifier

```
_mexp_gen:
    PUSH    0                   ; int u = 0
    MOV     EAX,[ESP+8]         ; int u = exp;
    MOV     [ESP+0],EAX
    CGSTART                     ; codegen(...
    CGDUMP  ECX,cdgn_beg$18 ; (dump 1st template)
    JMP     whiletest$21        ; while ...
whilebody$20:
    :                           ; if ((u & 1) != 0)
    :
    CMP     EAX,ECX
    JE      ifend$24
    CGDUMP  ECX,splc_beg$25     ; (dump 2nd template)
    MOV     EAX,[ESP+12]        ; mod
    CGFILL  ECX,splc_beg$25,hole$29,EAX
ifend$24:
    CGDUMP  ECX,splc_beg$31     ; (dump 3rd template)
    MOV     EAX,[ESP+12]        ; mod
    CGFILL  ECX,splc_beg$26,hole$34,EAX
    MOV     ECX,1               ; u = u >> 1;
    MOV     EAX,[ESP+0]
    SAR     EAX,CL
    MOV     [ESP+0],EAX
whiletest$21:
    :
    :
    CMP     EAX,ECX             ; ... (u != 0)
    JNE     whilebody$20
whileend$22:
    CGDUMP  ECX,cut_beg$33      ; (dump 4th template)
    CGEND   EAX
    ADD     ESP,4               ; (return spec. fun.)
    RETN



TEMPLATE_START cdgn_beg$18,cdgn_beg$19
cdgn_beg$18                     ; (1st template)
    PUSH    0                   ; int s = 0;
    :
    MOV     EAX,[ESP+12]        ; t = base;
    MOV     [ESP+0],EAX
TEMPLATE_END cdgn_beg$19

TEMPLATE_START splc_beg$25,splc_end$26
splc_beg$25:                            ; (2nd template)
    CGHOLE  EAX,splc_beg$25,hole$29  ; fill(...)
    :
    MOV     [ESP+4],EAX             ; s = mod(s*t,...));
TEMPLATE_END splc_end$26

TEMPLATE_START splc_beg$31,splc_end$32
splc_beg$31:                            ; (3rd template)
    CGHOLE  EAX,splc_beg$31,hole$34  ; fill(...)
    :
    MOV     [ESP+0],EAX             ; t = mod(t*t,...));
TEMPLATE_END splc_end$32

TEMPLATE_START cut_beg$36,cut_end$37
cut_beg$36:                             ; (4th template)
    MOV     EAX,[ESP+4]             ; return s;
    ADD     ESP,8
    RETN
TEMPLATE_END cut_end$37
```

Figure 4: TAL/T code

- TAL/T to assembly translation
- Assembler/Linker

For some parts, we were able to reuse existing software. Specifically, we used Tempo for action-annotated program generation, Microsoft MASM for assembling, and Microsoft Visual C++ for linking. Other parts extend existing work. This was the case for the Cyclone language, type system, verifier, compiler, and the TAL/T language. Some components needed to be written from scratch, including the translation from an action-annotated program into a Cyclone program, and the definition of the new TAL/T instructions in terms of x86 instructions.

We've organized the rest of the paper as follows. In Section 2, we present the Cyclone language and its type system. In Section 3, we give a brief description of TAL/T; due to limited space we defer a full description to a later paper. We give implementation details and initial impressions about performance in Section 4. We discuss related work in Section 5, and future work in Section 6. Our final remarks are in Section 7.

2 Cyclone

2.1 Design decisions

Cyclone's `codegen`, `cut`, `splice`, and `fill` constructs were designed to express a template-based style of run-time code generation cleanly and concisely. We made some other design decisions based on Cyclone's relationship to the C programming language, and on implementation concerns.

First, because a run-time specializer is a function that returns a function as its result, we need higher order types in Cyclone. In C, higher order types can be written using pointer types, but Cyclone does not have pointers. Therefore, we introduce new notation for higher order types in Cyclone. For example:

```
int (float,int) f(int x) { ... }
```

This is a Cyclone function `f` that takes an int argument `x`, and returns a function taking a float and an int and returning an int. When `f` is declared and not defined, we use

```
int (int) (float,int) f;
```

Note that the type of the first argument appears to the left of the remaining arguments. This is consistent with the order the arguments would appear in C, using pointer types.

A second design decision concerns the extent to which we should support nested `codegen`'s. Consider the following example.

```
int (float) (int) f(int x) {
  return(codegen(
    int (int) g(float y) {
      return(codegen(
        int h(int z) {
        ... body of h ...
        }));
    }));
}
```

Here `f` is a function that generates a function `g` using `codegen` when called at run time. In turn, `g` will generate a function `h` each time it is called. Nested `codegen`'s are thus used to generate code that generates code. The first version of Tempo did not support code that generates code (though it has recently been extended to do so), and some other systems, such as 'C [20, 21], also prohibit it. We decided to permit it in Cyclone, because it adds little complication to our type system or implementation. Nested `codegen`'s are not generated automatically in Cyclone, because of the version of Tempo that we use, but the programmer can always write them explicitly.

A final design decision concerns the extent to which Cyclone should support lexically scoped bindings. In the last example, the function `h` is nested inside of two other functions, `f` and `g`. In a language with true lexical scoping, the arguments and local variables of these outer functions would be visible within the inner function: `f`, `x`, `g`, and `y` could be used in the body of `h`.

We decided that we would *not* support full lexical scoping in Cyclone. Our scoping rule is that in the body of a function, only the function itself, its arguments and local variables, and top-level variables are visible. This is in keeping with C's character as a low-level, machine- and systems-oriented language: the operators in the language are close to those provided by the machine, and the cost of executing a program is not hidden by high-level abstractions. We felt that closures and lambda lifting, the standard techniques for supporting lexical scoping, would stray too far from this. If lexical scoping is desired, the programmer can introduce explicit closures. Or, lexical scoping can be achieved using the Cyclone features, for example, if `y` is needed in the body of `h`, it can be accessed using `fill(y)`.

2.2 Syntax and typing rules

Now we formalize a core calculus of Cyclone. Full Cyclone has, in addition, structures, unions, arrays, void, break and continue, and for and do loops.

We use x to range over variables, c to range over constants, and b to range over base types. There is an implicit signature assigning types to constants, so that we can speak of "the type of c." Figure 5 gives the grammars for programs p, modifiers m, types t, declarations d, sequences D of declarations, function definitions F, statements s, and expressions e.

We write $t \bullet m$ for the type of a function from m to t: if $t = b\ m_1\ \cdots\ m_n$, then $t \bullet m = b\ m\ m_1\ \cdots\ m_n$. If $D = t_1\ x_1, \ldots, t_k\ x_n$, then $\widetilde{D}$ is defined to be the modifier $(t_1, \ \ldots\ , t_n)$, so that a function definition $t\ x(D)\ s$ declares x to be of type $t \bullet \widetilde{D}$.

We sometimes consider a sequence $D = t_1\ x_1, \ldots, t_n\ x_n$ of declarations to be a finite function from variables to types: $D(x_i) = t_i$ if $1 \leq i \leq n$. This assumes that the x_i are distinct; we achieve this by alpha conversion when necessary, and by imposing some standard syntactic restrictions on Cyclone programs (the names of a function and its formal parameters must be distinct, and global variables have distinct names).

We define type environments E to support Cyclone's scoping rules:

$$
\begin{array}{rcl}
E & ::= & \texttt{outermost(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}\ D_{\text{global}} \\
& | & \texttt{frame(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}\ E \\
& | & \texttt{hidden(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}\ E
\end{array}
$$

Informally, a type environment is a sequence of hidden and visible *frames*, followed by an outermost frame that gives

$$E_{\text{vis}} = \begin{cases} D_2, D_1, t \bullet \widetilde{D_1}\ x, D_3 & \text{if } E = \texttt{outermost}(t\ x(D_1);\ D_2);\ D_3 \\ E'_{\text{vis}} & \text{if } E = \texttt{hidden}(t\ x(D);\ D'); E' \\ D', D, t \bullet \widetilde{D}\ x, E'_{\text{vis0}} & \text{if } E = \texttt{frame}(t\ x(D);\ D'); E' \end{cases}$$

$$E_{\text{vis0}} = \begin{cases} D_3 & \text{if } E = \texttt{outermost}(t\ x(D_1);\ D_2);\ D_3 \\ E'_{\text{vis0}} & \text{if } E = \texttt{hidden}(t\ x(D);\ D'); E' \\ E'_{\text{vis0}} & \text{if } E = \texttt{frame}(t\ x(D);\ D'); E' \end{cases}$$

$$\texttt{rtype}(E) = \begin{cases} t & \text{if } E = \texttt{outermost}(t\ x(D_1);\ D_2);\ D_3 \\ \texttt{rtype}(E') & \text{if } E = \texttt{hidden}(t\ x(D);\ D'); E' \\ t & \text{if } E = \texttt{frame}(t\ x(D);\ D'); E' \end{cases}$$

$$E + d = \begin{cases} \texttt{outermost}(t\ x(D_1);\ d\texttt{,}D_2);\ D_3 & \text{if } E = \texttt{outermost}(t\ x(D_1);\ D_2);\ D_3 \\ \texttt{hidden}(t\ x(D);\ D'); E' + d & \text{if } E = \texttt{hidden}(t\ x(D);\ D'); E' \\ \texttt{frame}(t\ x(D);\ d\texttt{,}D'); E' & \text{if } E = \texttt{frame}(t\ x(D);\ D'); E' \end{cases}$$

Figure 6: Cyclone environment functions

Programs	p	::=	$\cdot$
		|	d; p
		|	F p
Modifiers	m	::=	(t_1, ... , t_n)
Types	t	::=	$b\ m_1 \cdots m_n$
Declarations	d	::=	$t\ x$
Decl. sequences	D	::=	d_1, ... , d_n
Function defns.	F	::=	$t\ x(D)\ s$
Statements	s	::=	e;
		|	$d = e$;
		|	{ $s_1 \cdots s_n$ }
		|	`if` (e) s_1 `else` s_2
		|	`while` (e) s
		|	`return` e;
		|	`splice` s
		|	`cut` s
Expressions	e	::=	x
		|	c
		|	e_0(e_1, ...,e_n)
		|	x = e
		|	`codegen`(F)
		|	`fill`(e)

Figure 5: The grammar of core Cyclone

the type of a top level function, the types of its local variables, and the types of global variables. The non-outermost frames contain the type of a function that will be generated at run time, and types for the parameters and local variables of the function. If E is a type environment, we write E_{vis} for the *visible declarations* of E; E_{vis} is defined in Figure 6. Informally, the definition says that the declarations of the first non-hidden frame and the global declarations are visible, and all other declarations are not visible. Note that E_{vis} is a sequence of declarations, so we may write $E_{\text{vis}}(x)$ for the type of x in E.

Figure 6 also defines two other important operations on environments: $\texttt{rtype}(E)$ is the return type for the function of the first non-hidden frame, and $E + d$ is the environment obtained by adding declaration d to the local declarations of the first non-hidden frame.

The typing rules of Cyclone are given in Figure 7. The interesting rules are those for `codegen`, `cut`, `splice`, and `fill`.

A `codegen` expression starts the process of run time code generation. To type $\texttt{codegen}(t\ x(D)\ s)$ in an environment E, we type the body s of the function in an environment $\texttt{frame}(t\ x(D);\ \cdot);\ E$. This makes the function x and its parameters D visible in the body, while any enclosing function, parameters, and local variables will be hidden.

An expression $\texttt{fill}(e)$ should only appear within a template. Our typing rule ensures this by looking at the environment: it must have the form $\texttt{frame}(t\ x(D);\ D');\ E$. If so, the expression $\texttt{fill}(e)$ is typed if e is typed in the environment $\texttt{hidden}(t\ x(D);\ D');\ E$. That is, the function being generated with `codegen`, as well as its parameters and local variables, are hidden when computing the value that will fill the hole. This is necessary because the parameters and local variables will not become available until the function is called; they will not be available when the hole is filled.

The rules for `cut` and `splice` are similar. Like `fill`, `cut` can only be invoked within a template, and it changes `frame` to `hidden` for the same reason as `fill`. `Splice` is the dual of `cut`; it changes a frame `hidden` by `cut` back into a visible `frame`. Thus `splice` introduces a template, and `cut` interrupts a template.

$\boxed{D \vdash p}$ (p is a well-formed program)

$$D \vdash \cdot$$

$$\frac{D \vdash p}{D \vdash t\ x;\ p} \quad D(x) = t$$

$$\frac{D \vdash p, \quad D(x) = t \bullet \widetilde{D'} \qquad \texttt{outermost}(t\ x(D');\ \cdot\ );\ D \vdash s}{D \vdash t\ x(D')\ s\ p}$$

$\boxed{E \vdash s}$ (s is a well-formed statement)

$$\frac{E \vdash e : t}{E \vdash e;}$$

$$\frac{E \vdash e : t}{E \vdash t\ x = e;}$$

$$\frac{}{E \vdash \{\ \}}$$

$$\frac{E \vdash d = e; \quad E + d \vdash \{\ s_1\ \cdots\ s_n\ \}}{E \vdash \{\ d = e;\ s_1\ \cdots\ s_n\ \}}$$

$$\frac{E \vdash s_0,\ E \vdash \{\ s_1\ \cdots\ s_n\ \}}{E \vdash \{\ s_0\ s_1\ \cdots\ s_n\ \}} \quad s_0 \neq d = e;$$

$$\frac{E \vdash e : \texttt{int}, \quad E \vdash s_1, \quad E \vdash s_2}{E \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

$$\frac{E \vdash e : \texttt{int}, \quad E \vdash s}{E \vdash \texttt{while}\ (e)\ s}$$

$$\frac{E \vdash e : t}{E \vdash \texttt{return}\ e;} \quad \texttt{rtype}(E) = t$$

$$\frac{\texttt{frame}(t\ x(D);\ D'); E \vdash s}{\texttt{hidden}(t\ x(D);\ D'); E \vdash \texttt{splice}\ s}$$

$$\frac{\texttt{hidden}(t\ x(D);\ D'); E \vdash s}{\texttt{frame}(t\ x(D);\ D'); E \vdash \texttt{cut}\ s}$$

$\boxed{E \vdash e : t}$ (e has type t)

$$E \vdash x : t \quad \text{if } E_{\text{vis}}(x) = t$$

$$E \vdash c : t \quad \text{where } t \text{ is the type of the constant } c$$

$$\frac{E \vdash e_0 : t \bullet (t_1, \ \ldots, \ t_n),\ E \vdash e_1 : t_1,\ \ldots,\ E \vdash e_n : t_n}{E \vdash e_0(e_1, \ \ldots, e_n) : t}$$

$$\frac{E \vdash x : t,\ E \vdash e : t}{E \vdash x\ \texttt{=}\ e : t}$$

$$\frac{\texttt{frame}(t\ x(D);\ \cdot); E \vdash s}{E \vdash \texttt{codegen}(t\ x(D)\ s) : t \bullet \widetilde{D}}$$

$$\frac{\texttt{hidden}(t\ x(D);\ D'); E \vdash e : t}{\texttt{frame}(t\ x(D);\ D'); E \vdash \texttt{fill}(e) : t}$$

Figure 7: Typing rules of Cyclone

An operational semantics for Cyclone and safety theorem are given in an appendix.

3 TAL/T

The output of the Cyclone compiler is a program in TAL/T, an extension of the Typed Assembly Language (TAL) of Morrisett et al. [16]. In designing TAL/T, our primary concern was to retain the low-level, assembly language character of TAL. Most TAL instructions are x86 machine instructions, possibly annotated with type information. The exceptions are a few macros, such as `malloc`, that would be difficult to type in their expanded form; each macro expands to a short sequence of x86 instructions. Since each instruction is simple, the trusted components of the system—the typing rules, the verifier, and the macros—are also simple. This gives us a high degree of confidence in the correctness and safety of the system.

TAL already has instructions that are powerful enough to generate code at run time: `malloc` and move are sufficient. The problem with this approach is in the types. If we `malloc` a region for code, what is its type? Clearly, by the end of the code generation process, it should have the type of TAL code that can be jumped to. But at the start of code generation, when it is not safe to jump to, it must have a different type. Moreover, the type of the region should change as we move instructions into it. The TAL type system is not powerful enough to show that a sequence of `malloc` and move instructions results in a TAL program that can safely be jumped to.

Our solution, TAL/T, is an extension of TAL with some types and macros for manipulating templates. Since this paper focuses on Cyclone and the front end of system, we will only sketch the ideas of TAL/T here. Full details will appear in a subsequent paper.

In TAL, a procedure is just the label or address of a sequence of TAL instructions. A procedure is called by jumping to the label or address. The type of a procedure is a precondition that says that on entry, the x86 registers should contain values of particular types. For example, if a procedure is to return it will have a precondition saying that a return address should be accessible through the stack pointer when it is jumped to.

In TAL/T, a template is also the label of a sequence of instructions. Unlike a TAL procedure, however, a template is not meant to be jumped to. For example, it might need to be concatenated with another template to form a TAL procedure. Thus the type of a template includes a postcondition as well as a precondition. Our typing rules for the template instructions of TAL/T will ensure that before a template is dumped into a code generation region, its precondition matches the postcondition of the previous template dumped. Also, a template may have holes that need to be filled; the types of these holes are also given in the type of the template.

The type of a code generation region is very similar to that of a template: it includes types for the holes that remain to be filled in the region, the precondition of the first template that was dumped, and the postcondition of the last template that was dumped. When all holes have been filled and a template with no postcondition is dumped, the region will have a type consisting of just a precondition, i.e., the type of a TAL procedure. At this point code generation is finished and the result can be jumped to.

```
int f(int x) {
  return(codegen(
    int g(int y) {
      return y+1;
    })(x));
}

int h(int x)(int) {
  return(codegen(
    int k(int y) {
      return(fill(f(x)))
    }));
}
```

Figure 8: An example showing that two `codegen` expressions can be executing at once. When called, `h` starts generating `k`, but stops in the middle to call `f` which generates `g`.

Now we give a brief description of the new TAL/T macros. This is intended to be an informal description showing that each macro does not go beyond what is already in TAL—the macros are low level, and remain close to machine code.

The macros manipulate an implicit stack of code generation regions. Each region in the stack is used for a function being generated by a `codegen`. The stack is needed because it is possible to have two `codegen` expressions executing at once (for an example, see Figure 8).

- `cgstart` initiates run-time code generation by allocating a new code generation region. This new region is pushed onto the stack of code generation regions and becomes the "current" region. The `cgstart` macro is about as complicated as `malloc`.
- `cgdump` r, L copies the template at label L into the current code generation region. After execution, the register r points to the copy of the template, and can be used to fill holes in the copy. `Cgdump` is our most complicated macro: its core is a simple string-copy loop, but it must also check that the current code generation region has enough room for a copy of the template. If there is not enough room, `cgdump` allocates a new region twice the size of the old region, copies the contents of the old region plus the new template to the new region, and replaces the old region with the new on the region stack. This is the most complex TAL/T instruction, consisting of roughly twenty x86 instructions.
- `cghole` r, L_{template}, L_{hole} is a move instruction containing a hole. It should be used in a template with label L_{template}, and declares the hole L_{hole}.
- `cgfill` r_1, L_{template}, L_{hole}, r_2 fills the hole of a template; it is a simple move instruction. Register r_1 should point to a copy of the template at label L_{template}, which should have a hole with label L_{hole}. Register r_2 contains the value to put in the hole.
- `cgfillrel` fills the hole of a template with a pointer into a second template; like `cgfill` it expands to a simple move instruction. It is needed for jumps between templates.

```
int f() {
  return(codegen(
    int g(int i) {
      cut { return 4; }
      return(i+1);
    })(7));
}
```

Figure 9: An example that shows the need for `cgabort`. When called, the function `f` starts generating function `g` but aborts in the middle (it returns 4).

- `cgabort` aborts a code generation; it pops the top region off the region stack. It is needed when the run-time code generation of a function stops in the middle, as in the example of Figure 9.
- `cgend` r finalizes the code generation process: the current region is popped off the region stack and put into register r. TAL can then jump to location r.

4 Implementation Status

We now describe some key aspects of our implementation. As previously mentioned, some components were written from scratch, while others were realized by modifying existing software.

4.1 Action-annotated program to Cyclone

We translate Tempo action-annotated programs into run-time specializers written in Cyclone. Using the Tempo front end, this lets us automatically generate a Cyclone program from a C program.

An action-annotated program distinguishes two kinds of code: normal code that will be executed during specialization, indicated in italics in Fig. 2; and template code that will emitted during specialization (non-italicized code). The annotated C program is translated into a Cyclone program that uses `codegen`, `cut`, `splice`, and `fill`. Since italicized constructs will be executed during code generation, they will occur outside `codegen`, or within a `cut` statement or a `fill` expression. Non-italicized constructs will be placed within a `codegen` expression or `splice` statement.

Our algorithm operates in two modes: "normal" mode translates constructs that should be executed at code generation time and "template" mode translates constructs that will be part of a template. The algorithm performs a recursive descent of the action-annotated abstract syntax, keeping track of which mode it is in. It starts off in "normal" mode and produces Cyclone code for the beginning of the run-time specializer: its arguments (the invariants) and any local variables and initial statements that are annotated with italics. When the first non-italic construct is encountered, a `codegen` expression is issued, putting the translation into "template" mode. The rest of the program is translated as follows.

An italic statement or expression must be translated in "normal" mode. Therefore, if the translation is in "template" mode, we insert `cut` (if we are processing a statement)

or `fill` (if we are processing an expression) and switch into "normal" mode. Similarly, a non-italic statement should be translated in "template" mode; here we insert `splice` and switch modes if necessary. It isn't possible to encounter a non-italic expression within an italic expression.

Another step needs to be taken during this translation since specialization is *speculative*, i.e., both branches of a conditional statement can be optimistically specialized when the conditional test itself cannot be evaluated. This means that during specialization, the store needs to be saved prior to specializing one branch and restored before specializing the other branch. Therefore, we must introduce Cyclone statements to save and restore the store when translating such a conditional statement. This is the same solution used by Tempo [6].

4.2 Cyclone to TAL/T

To compile Cyclone to TAL/T, we extended an existing compiler, the Popcorn compiler of Morrisett et al. Popcorn is written in Caml, and it compiles a type safe dialect of C into TAL, a typed assembly language [16]. Currently, Popcorn is a very simple, stack based compiler, though it is being extended with register allocation and more sophisticated optimizations.

The Popcorn compiler works by performing a traversal of the abstract syntax tree, emitting TAL code as it goes. It uses an environment data structure of the following form:

```
type env = { local_env: (id * int) list;
             args_on_stack: int }
```

The environment maintains the execution state of each function as it is compiled. The field `local_env` contains each variable identifier and its corresponding stack offset. Arguments are pushed onto the stack prior to entry to the function body; the field `args_on_stack` records the number of arguments, so they can be popped off the stack upon exiting the function.

To compile Cyclone we needed to extend the environment datatype: first, because Cyclone switches between generating normal code and template code, and second, because Cyclone has nested functions. Therefore, we use environments with the same structure as the environments used in Cyclone's typing rules:

```
type cyclone_env =
    Outermost of env * (id list)
  | Frame of env * cyclone_env
  | Hidden of env * cyclone_env
```

That is, environments are sequences of type frames for functions. A frame can either be outermost, normal, or hidden. Once we have this type of environment, visible bindings are defined as they are for E_{vis} in Section 2.

An `Outermost` frame contains the local environment for a top-level function as well as the global identifiers. A `Frame` is used when compiling template code. A new `Frame` environment is created each time `codegen` is encountered. A `Frame` becomes `Hidden` to switch back to "normal" mode when a `cut` or `fill` is encountered.

Popcorn programs are compiled by traversing the abstract syntax tree and translating each Popcorn construct into the appropriate TAL instructions; the resulting sequence of TAL instructions is the compiled program. Compiling a Cyclone program, however, is more complicated; it is performed in two phases. The first phase alternates between generating normal and template TAL/T instructions and a second phase rearranges the instructions to put them in their proper place. In order for the instructions to be rearranged in the second phase, the first phase interleaves special *markers* with the TAL/T instructions:

```
type marker =
    M_TemplateBeg of id * id
  | M_TemplateEnd
  | M_Fill of id * exp
```

These markers are used to indicate which instructions are normal, which belong within a template, and which are used to fill holes. `M_TemplateBeg` takes two arguments, the beginning and ending label of a template, and is issued at the beginning of a template (when `codegen` or `splice` is encountered, or `cut` ends). Similarly, `M_TemplateEnd` is issued at the end of a template (at the end of a `codegen` or `splice`, or the beginning of a `cut`). Note that between corresponding `M_TemplateBeg` and `M_TemplateEnd` markers, other templates may begin and end. Therefore, these markers can be nested. When a hole is encountered, a `M_Fill` marker is issued. The first argument of `M_Fill` is a label for the hole inside the template. The second argument is the Cyclone source code expression that should fill the hole.

The following example shows how the `cut` statement is compiled.

```
fun compile_stmt stmt cyclone_env =
 match stmt of
  Cut s ->
   match cyclone_env with
     Outermost _ -> raise Error
   | Frame(env,cyclone_env') ->
       cg_fill_holes (Hidden(env,cyclone_env'));
       compile_stmt s (Hidden(env,cyclone_env'));
       emit_mark(M_TemplateBeg(id_new "a", id_new "b"));
   | Hidden _ -> raise Error
```

The function `compile_stmt` takes a Cyclone statement and an environment, and emits TAL/T instructions as a side-effect. The first thing to notice is that a cut can only occur when the compiler is in "template" mode, in which case the environment begins with `Frame`. A `cut` statement ends a template. Therefore, `cg_fill_holes` is called, which emits a `M_TemplateEnd` marker, and emits TAL/T code to dump the template and fill its holes. Filling holes must be done using a "normal" environment, and therefore the first frame becomes `Hidden`. Next, `compile_stmt` is called recursively to compile the statement `s` within the cut. Since the statement `s` should also be compiled in normal mode, it also keeps the first frame `Hidden`. Finally, a `M_TemplateBeg` marker is emitted so that the compilation of any constructs following the cut will occur within a new template.

The second phase of the code generation uses the markers to rearrange the code. The TAL/T instructions issued between a `M_TemplateBeg` and a `M_TemplateEnd` marker are extracted and made into a template. The remaining, normal instructions are concatenated to make one function; hole filling instructions are inserted after the instruction which dumps the template that contains the hole. The example in Fig. 4 shows a TAL/T program after the second phase is completed; the normal code includes instructions to dump templates and fill holes, and is followed by the templates.

4.3 TAL/T to executable

TAL is translated into assembly code by expanding each TAL macro into a sequence of x86 instructions. Similarly, the new TAL/T macros expand into a sequence of x86 and TAL instructions. A description of each TAL/T macro is given in Section 3. The resulting x86 assembly language program is assembled with Microsoft MASM and linked with the Microsoft Visual C++ linker.

4.4 Initial Impressions

We have implemented our system and have started testing it on programs to assess its strengths and weaknesses. Since there is currently a lot of interest in specializing interpreters, we decided to explore this type of application program. A state-of-the-art program specializer such as Tempo typically achieves a speedup between 2 and 20, depending on the interpreter and program interpreted. To see how our system compares, we took a bytecode interpreter available in the Tempo distribution and ran it through our system.

Preliminary results show that Cyclone achieves a speedup of over 3. This is encouraging, since this is roughly the speedup Tempo achieves on similar programs. A more precise comparison of the two systems still needs to be done, however. On the other hand, in our initial implementation, the cost of generating code is higher than in Tempo. One possible reason is that for safety, we allocate our code generation regions at run time, and perform bounds checks as we dump templates. The approach taken by Tempo, choosing a maximum buffer size at compile time and allocating a buffer of that size, is faster but not safe.

5 Related Work

Propagating types through all stages of a compiler, from the front end to the back end, has been shown to aid robust compiler construction: checking type safety after each stage quickly identifies compiler bugs [23, 24]. Additionally, Necula and Lee have shown that proving properties at the assembly language level is useful for safe execution of untrusted mobile code [18]. So far, this approach has been taken only for statically generated code. Our system is intended to achieve these same goals for dynamically generated code.

Many of the ideas in Cyclone were derived from the Tempo run-time specializer [7, 12, 13]. We designed Cyclone and TAL/T with a template-based approach in mind, and we use the Tempo front end for automatic template identification. Another run-time specializer, DyC, shares some of the same features, such as static analyses and a template-like back end [5, 9, 10]. There are, however, some important differences between Cyclone and these systems. We have tried to make our compiler more robust than Tempo and DyC, by making Cyclone type safe, and by using types to verify the safety of compiled code. Like Tempo and DyC, Cyclone can automatically construct specializers, but in addition, Cyclone also gives the programmer explicit control over run-time code generation, via the `codegen`, `cut`, `splice`, and `fill` constructs. It is even possible for us to hand-tweak the specializers produced by the Tempo front end with complete type safety. Like DyC, we can perform optimizations such as inter-template code motion, since we are writing our own compiler. Tempo's strategy of using an unmodified, existing compiler limits the optimizations that it can perform.

ML-box, Meta-ML, and 'C are all systems that add explicit code generation constructs to existing languages. ML-box and Meta-ML are type safe dialects of ML [15, 25, 22], while 'C is an unsafe dialect of C [20, 21]. All three systems have features for combining code fragments that go beyond what we provide in Cyclone. For example, in 'C it is possible to generate functions that have n arguments, where n is a value computed at run time; this is not possible in Cyclone, ML-box, or Meta-ML. On the other hand, 'C cannot generate a function that generates a function; this can be done in Cyclone (using nested `codegens`), and also in ML-box and Meta-ML. An advantage we gain from not having sophisticated features for manipulating code fragments is simplicity: for example, the Cyclone type system does not need a new type for code fragments. The most fundamental difference, however, is that the overall system we present will provide type safety not only at the source level, but also at the object level. This makes our system more robust and makes it usable in a proof carrying code system.

6 Future Work

In this paper we presented a framework for performing safe and robust run-time code generation. Our compiler is based on a simple, stack-based, certifying compiler written by Morrisett et al. They are extending the compiler with register allocation and other standard optimizations, and we expect to merge Cyclone with their improvements.

We are interested in studying template-specific optimizations. For example, because templates appear explicitly in TAL/T, we plan to study inter-template optimizations, such as code motion between templates. Performing inter-template optimizations is more difficult in a system, like Tempo, based on an existing compiler that is not aware of templates.

We are also interested in analyses that could statically bound the size of the dynamic code generation region. This would let us allocate exactly the right amount of space when we begin generating code for a function, and would let us eliminate bounds checks during template dumps.

We would like to extend the front end of Tempo so that it takes Cyclone, and not just C, as input. This would mean extending the analyses of Tempo to handle Cyclone, which is an n-level language like ML-box. Additionally, we may implement the analysis of Glück and Jørgensen [8] to produce n-level Cyclone from C or Cyclone.

7 Conclusion

We have designed a programming language and compiler that combines dynamic code generation with certified compilation. Our system, Cyclone, has the following features.

Robust dynamic code generation Existing dynamic code generation systems only prove safety at the source level. Our approach extends this to object code. This means that bugs in the compiler that produce unsafe run-time specializers can be caught at compile time, before the specializer itself is run. This is extremely helpful because of the complexity of the analyses and transformations involved in dynamic code generation.

Flexibility and Safety Cyclone produces dynamic code generators that exploit run-time invariants to produce optimized programs. The user interface is flexible, since the final executable can be generated from a C program, a Cyclone program, or TAL/T assembly code. Type safety is statically verified in all three cases.

Safe execution of untrusted, dynamic, mobile code generators This approach can be used to extend a proof-carrying code system to include dynamic code generation. Since verification occurs prior to run time, there is no run-time cost incurred for the safety guarantees. Sophisticated optimization techniques can be employed in the certifying compiler. The resulting system could produce mobile code that is not only safe, but potentially extremely fast.

Acknowledgements. We were able to implement Cyclone quickly because we worked from the existing Tempo and TAL implementations. We'd like to thank Charles Consel and the Tempo group, and Greg Morrisett and the TAL group, for making this possible. The paper was improved by feedback from Julia Lawall and the anonymous referees.

References

[1] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[2] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, 12–13 June 1997.

[3] *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[4] L. Augustsson. Partial evaluation in aircraft crew planning. In ACM [2], pages 127–136.

[5] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In ACM [1], pages 149–159. *SIGPLAN Notices* 31(5), May 1996.

[6] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Feb. 1996.

[7] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[8] R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Comput. Sci.* To appear.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In ACM [2], pages 163–178.

[11] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[12] L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications.* PhD thesis, Université de Rennes I, June 1997.

[13] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In ACM [2], pages 63–73.

[14] D. Keppel, S. Eggers, and R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science, University of Washington, Seattle, WA, 1991.

[15] M. Leone and P. Lee. Lightweight run-time code generation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Orlando, Florida, 25 June 1994. University of Melbourne, Australia, Department of Computer Science, Technical Report 94/9.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, California, 19–21 Jan. 1998.

[17] G. Muller, R. Marlet, E. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[18] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In ACM [3], pages 333–344. *SIGPLAN Notices* 33(5), May 1998.

[19] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[20] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[21] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Trans. Prog. Lang. Syst.* To appear.

[22] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In ACM [2], pages 203–217.

[23] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Dec. 1997. CMU–CS–97–108.

[24] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In ACM [1], pages 181–192. *SIGPLAN Notices* 31(5), May 1996.

[25] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In ACM [3], pages 224–235. *SIGPLAN Notices* 33(5), May 1998.

A Operational semantics of Cyclone

In this appendix, we define a small-step operational semantics for Cyclone based on evaluation contexts.

It is convenient to use a modified grammar for Cyclone in the operational semantics; this grammar is summarized in Figure 10. We have extended the Cyclone expressions with a new form, $\langle\!\langle s \rangle\!\rangle_t$. This is an expression consisting of a statement that should finish computing by evaluating `return` on a value of type t. We have introduced a new class, f, of variables called *function names*. Function names are distinct from the usual variables, x, and are intended to be used in the semantics only—they are not present in user programs. *Values*, v, are constants or function names, but not variables x. *Heaps*, H, are sequences of definitions, for both normal variables and for function names. The variables defined in a heap must be distinct; in our semantics we implicitly use alpha conversion when necessary to achieve this. If a heap H defines variables $z_1, \ldots, z_n$ with types $t_1, \ldots, t_n$, then $\widetilde{H}$ is defined to be the sequence of declarations $t_1\ z_1, \ldots, t_n\ z_n$. It is sometimes convenient to use $\widetilde{H}$ where we would use D, although, strictly speaking, some of the z_i will be function names, contrary to the definition of D.

Our new syntax requires some new typing rules, presented below. We assume that there is a distinguished function name, `main`, and for declarations D we define $\hat{D}$ to be the environment `outermost(int main(); ·);` D. The name `main` is not permitted anywhere else. Additionally, we extend Cyclone type environments as follows:

$$E ::= \cdots \mid \texttt{return(}t\texttt{);}\ E$$

This marks an expected return type, and is used in typing the new construct, $\langle\!\langle s \rangle\!\rangle_t$. The environment functions, like `rtype`(E), are extended in the obvious way.

$\boxed{E \vdash e : t}$ (e has type t)

$$\frac{\texttt{return(}t\texttt{);}\ E \vdash s}{E \vdash \langle\!\langle s \rangle\!\rangle_t : t}$$

$$E \vdash f : t \quad \text{if } E_{\text{vis}}(f) = t$$

$\boxed{D \vdash H}$ (H is a well-formed heap)

$$D \vdash \cdot$$

$$\frac{\hat{D} \vdash v : t, \quad D \vdash H}{D \vdash H, d = v} \quad d = (t\ x) \in D$$

$$\frac{\texttt{outermost(}t\ f\texttt{(}D'\texttt{);}\ \cdot\ \texttt{);}\ D \vdash s \quad D \vdash H}{D \vdash H, t\ f\texttt{(}D'\texttt{)}\ s} \text{ where } D(f) = t \bullet \widetilde{D'}$$

$\boxed{\vdash H, s}$ (H, s is a well-formed program)

$$\frac{\widetilde{H} \vdash H, \quad \hat{\widetilde{H}} \vdash s}{\vdash H, s}$$

The evaluation contexts for Cyclone are given in Figure 11. We need about four times the number of contexts as usual, because Cyclone distinguishes statements from expressions, and because we have both normal evaluation and evaluation under templates. For example, the context SE[·] becomes a statement when its hole is filled by an expression, and the context SS[·] becomes a statement when its hole is filled by a statement. Similarly, the context TES[·] becomes an expression when its hole is filled by a statement; the "T" contexts are meant to be used within templates. Fortunately, the variations between the different classes of evaluation context are small.

Our rewriting semantics is defined by identifying "redexes," which are the syntactic subterms where rewriting will take place. We have two kinds of redexes, expression redexes and statement redexes. The expression redexes are:

- x
- $v_0\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}$
- $x = v$
- $\langle\!\langle \{\} \rangle\!\rangle_t$
- `codegen(`$t\ x\texttt{(}D\texttt{)}\ \overline{s}$`)`
- `fill(`v`)`

The statement redexes are:

- v`;`
- $d = v$`;`
- `{ {}` $s \cdots$ `}`
- `if (`v`)` $s_1\ s_2$
- `while (`e`)` s
- `return` v`;`
- `splice` s
- `cut {}`

The evaluation contexts are used to specify which redex in a program should be evaluated first; that is, they give the evaluation order. The following lemma (easily proved by induction) shows that our evaluation contexts and redexes exactly capture all of the cases we will need to consider to define a complete, deterministic evaluation relation.

Lemma. For any statement s, exactly one of the following cases holds:

- there is a unique context SS[·] such that $s = \text{SS}[s']$, where s' is a statement redex;
- there is a unique context SE[·] such that $s = \text{SE}[e]$, where e is an expression redex; or

$$
\begin{array}{rcl@{\qquad}rcl}
m & ::= & (t_1, \ \ldots \ , \ t_n) & t & ::= & b\ m_1 \cdots m_n \\
d & ::= & t\ x & D & ::= & d_1, \ldots, d_n \\
v & ::= & c \mid f & H & ::= & \cdot \mid H, d = v \mid H, t\ f(D)\ s \\
s & ::= & e; & \overline{s} & ::= & \overline{e}; \\
 & \mid & d = e; & & \mid & d = \overline{e}; \\
 & \mid & \{\ s_1\ \cdots\ s_n\ \} & & \mid & \{\ \overline{s_1}\ \cdots\ \overline{s_n}\ \} \\
 & \mid & \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 & & \mid & \texttt{if}\ (\overline{e})\ \overline{s_1}\ \texttt{else}\ \overline{s_2} \\
 & \mid & \texttt{while}\ (e)\ s & & \mid & \texttt{while}\ (\overline{e})\ \overline{s} \\
 & \mid & \texttt{return}\ e; & & \mid & \texttt{return}\ \overline{e}; \\
 & \mid & \texttt{splice}\ s & & \mid & \texttt{splice}\ \overline{s} \\
 & \mid & \texttt{cut}\ s & & & \\
e & ::= & x & \overline{e} & ::= & x \\
 & \mid & c & & \mid & c \\
 & \mid & f & & \mid & f \\
 & \mid & e_0(e_1, \ \ldots, e_n) & & \mid & \overline{e_0}(\overline{e_1}, \ \ldots, \overline{e_n}) \\
 & \mid & x\ \texttt{=}\ e & & \mid & x\ \texttt{=}\ \overline{e} \\
 & \mid & \langle\!\langle s \rangle\!\rangle_t & & \mid & \langle\!\langle \overline{s} \rangle\!\rangle_t \\
 & \mid & \texttt{codegen}(t\ x(D)\ s) & & \mid & \texttt{codegen}(t\ x(D)\ s) \\
 & \mid & \texttt{fill}(e) & & &
\end{array}
$$

Figure 10: Grammars for Cyclone's operational semantics

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{SE}[\cdot] & ::= & & \mathrm{SS}[\cdot] & ::= & [\cdot] \\
 & & \mathrm{EE}[\cdot]; & & \mid & \mathrm{ES}[\cdot]; \\
 & \mid & d = \mathrm{EE}[\cdot]; & & \mid & d = \mathrm{ES}[\cdot]; \\
 & \mid & \{\ \mathrm{SE}[\cdot]\ s_1 \cdots s_n\ \} & & \mid & \{\ \mathrm{SS}[\cdot]\ s_1 \cdots s_n\ \} \\
 & \mid & \texttt{if}\ (\mathrm{EE}[\cdot])\ s_1\ s_2 & & \mid & \texttt{if}\ (\mathrm{ES}[\cdot])\ s_1\ s_2 \\
 & \mid & \texttt{return}\ \mathrm{EE}[\cdot]; & & \mid & \texttt{return}\ \mathrm{ES}[\cdot]; \\
 & \mid & \texttt{cut}\ \mathrm{SE}[\cdot] & & \mid & \texttt{cut}\ \mathrm{SS}[\cdot] \\
\mathrm{EE}[\cdot] & ::= & [\cdot] & \mathrm{ES}[\cdot] & ::= & \\
 & \mid & \mathrm{EE}[\cdot](e_1\ , \ \ldots\ , \ e_n) & & & \mathrm{ES}[\cdot](e_1\ , \ \ldots\ , \ e_n) \\
 & \mid & v(v', \ \ldots\ , \ \mathrm{EE}[\cdot]\ , \ e, \ \ldots) & & \mid & v(v', \ \ldots\ , \ \mathrm{ES}[\cdot]\ , \ e, \ \ldots) \\
 & \mid & x = \mathrm{EE}[\cdot] & & \mid & x = \mathrm{ES}[\cdot] \\
 & \mid & \langle\!\langle \mathrm{SE}[\cdot] \rangle\!\rangle_t & & \mid & \langle\!\langle \mathrm{SS}[\cdot] \rangle\!\rangle_t \\
 & \mid & \texttt{codegen}\ (t\ x(D)\ \mathrm{TSE}[\cdot]) & & \mid & \texttt{codegen}\ (t\ x(D)\ \mathrm{TSS}[\cdot]) \\
 & \mid & \texttt{fill}(\mathrm{EE}[\cdot]) & & \mid & \texttt{fill}(\mathrm{ES}[\cdot]) \\
\mathrm{TSE}[\cdot] & ::= & \mathrm{TEE}[\cdot]; & \mathrm{TSS}[\cdot] & ::= & \mathrm{TES}[\cdot]; \\
 & \mid & d = \mathrm{TEE}[\cdot]; & & \mid & d = \mathrm{TES}[\cdot]; \\
 & \mid & \{\ \overline{s}\ \cdots\ \mathrm{TSE}[\cdot]\ s'\ \cdots\ \} & & \mid & \{\ \overline{s}\ \cdots\ \mathrm{TSS}[\cdot]\ s'\ \cdots\ \} \\
 & \mid & \texttt{if}\ (\mathrm{TEE}[\cdot])\ s_1\ s_2 & & \mid & \texttt{if}\ (\mathrm{TES}[\cdot])\ s_1\ s_2 \\
 & \mid & \texttt{if}\ (\overline{e})\ \mathrm{TSE}[\cdot]\ s_2 & & \mid & \texttt{if}\ (\overline{e})\ \mathrm{TSS}[\cdot]\ s_2 \\
 & \mid & \texttt{if}\ (\overline{e})\ \overline{s}\ \mathrm{TSE}[\cdot] & & \mid & \texttt{if}\ (\overline{e})\ \overline{s}\ \mathrm{TSS}[\cdot] \\
 & \mid & \texttt{while}\ (\mathrm{TEE}[\cdot])\ s & & \mid & \texttt{while}\ (\mathrm{TES}[\cdot])\ s \\
 & \mid & \texttt{while}\ (\overline{e})\ \mathrm{TSE}[\cdot] & & \mid & \texttt{while}\ (\overline{e})\ \mathrm{TSS}[\cdot] \\
 & \mid & \texttt{return}\ \mathrm{TEE}[\cdot]; & & \mid & \texttt{return}\ \mathrm{TES}[\cdot]; \\
 & \mid & \texttt{cut}\ \mathrm{SE}[\cdot] & & \mid & \texttt{cut}\ \mathrm{SS}[\cdot] \\
\mathrm{TEE}[\cdot] & ::= & \mathrm{TEE}[\cdot](e_1\ , \ \ldots\ , \ e_n) & \mathrm{TES}[\cdot] & ::= & \mathrm{TES}[\cdot](e_1\ , \ \ldots\ , \ e_n) \\
 & \mid & \overline{e_0}(\overline{e_1}, \ \ldots\ , \ \mathrm{TEE}[\cdot]\ , \ e_n, \ \ldots) & & \mid & \overline{e_0}(\overline{e_1}, \ \ldots\ , \ \mathrm{TES}[\cdot]\ , \ e_n, \ \ldots) \\
 & \mid & x = \mathrm{TEE}[\cdot] & & \mid & x = \mathrm{TES}[\cdot] \\
 & \mid & \langle\!\langle \mathrm{TSE}[\cdot] \rangle\!\rangle_t & & \mid & \langle\!\langle \mathrm{TSS}[\cdot] \rangle\!\rangle_t \\
 & \mid & \texttt{fill}(\mathrm{EE}[\cdot]) & & \mid & \texttt{fill}(\mathrm{ES}[\cdot])
\end{array}
$$

Figure 11: Evaluation contexts for Cyclone's operational semantics

- $s = \{\}$.

The constructs `return`, `cut`, `splice`, and `fill` may result in errors depending on the context in which they execute. The following definition makes the relevant conditions on the contexts precise.

Definition.

- A context SS[·] is in *return mode* if its hole does not occur within $\langle\langle \cdot \rangle\rangle_t$.
- A context SS[·] is in *codegen mode* if

$$\text{SS}[\cdot] = \text{SE}[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \text{SS}'[\cdot]\texttt{)}]$$

  where the hole of SS′[·] does not occur within `cut`.
- A context SE[·] is in *codegen mode* if

$$\text{SE}[\cdot] = \text{SE}_1[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \text{SE}_2[\cdot]\texttt{)}]$$

  where the hole of $\text{SE}_2[\cdot]$ does not occur within `cut`.

Finally, the rewriting relation $\rightarrow$ is defined by the rewrite rules in Figure 12. The relation rewrites a heap and statement H, s. By the previous lemma and a case analysis of the rewrite rules, we can see that there are four disjoint possibilities:

- $H, s \rightarrow H', s'$ for some heap and statement H', s';
- $H, s \rightarrow \text{error}$;
- $H, s \rightarrow \text{error}_{\text{return}}$; or
- s is the statement $\{\}$, in which case evaluation is halted (there is no H', s' such that $H, s \rightarrow H', s'$, and H, s does not rewrite to error or $\text{error}_{\text{return}}$).

There are two kinds of error. Most errors are prevented by the type system; these are simply lumped together under "error." These errors include applying a function to the wrong number of arguments, trying to apply a non-function, using undefined variables, etc. There is also an error, "$\text{error}_{\text{return}}$," that is not prevented by our type system. It occurs when a function completes without executing a `return` statement. In our implementation, when this happens execution halts and the error is reported to the user. The full implementation has a few more errors of this sort, including array bound, stack overflow, and out of memory errors.

The safety theorem is stated as follows.

Theorem (Safety). If $\vdash H, s$, then there is no H', s' such that $H, s \rightarrow^* H', s' \rightarrow \text{error}$.

The theorem is proved by showing that types are preserved by each rule rewriting $H, s \rightarrow H', s'$, and by showing that if a rule rewrites $H, s \rightarrow \text{error}$, then H, s is not well-typed.

$$H, \mathrm{SE}[x] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{if } H = H_1, t\ x = v, H_2 \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[v_0\texttt{(}v_1\texttt{, ..., }v_n\texttt{)}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[\langle\!\langle \{d_1 = v_1\texttt{; } \cdots\texttt{; } d_n = v_n\texttt{; } s\} \rangle\!\rangle_t] \\ \qquad \text{if } v_0 = f \text{ and } H = H_1, t\ \ f\texttt{(}d_1, \ldots, d_n\texttt{)}\ \ s, H_2 \\ \text{error} \quad \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[x = v] \;\rightarrow\; \begin{cases} H_1, t\ x = v, H_2, \mathrm{SE}[v] & \text{if } H = H_1, t\ x = v', H_2 \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[\langle\!\langle \{\} \rangle\!\rangle_t] \;\rightarrow\; \mathrm{error}_{\mathrm{return}}$$

$$H, \mathrm{SE}[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \overline{s}\texttt{)}] \;\rightarrow\; H, t\ \ f\texttt{(}D\texttt{)}\ \ \overline{s}[x := f], \mathrm{SE}[f] \quad \text{where } f \text{ is a fresh function name}$$

$$H, \mathrm{SE}[\texttt{fill(}v\texttt{)}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{where SE}[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[v\texttt{;}] \;\rightarrow\; H, \mathrm{SS}[\{\}]$$

$$H, \mathrm{SS}[d = v\texttt{;}] \;\rightarrow\; H, d = v, \mathrm{SS}[\{\}] \quad \text{where the variable of } d \text{ does not appear in } H$$

$$H, \mathrm{SS}[\texttt{\{ \{\} }\ s \cdots\ \texttt{\}}] \;\rightarrow\; H, \mathrm{SS}[\texttt{\{ }\ s \cdots\ \texttt{\}}]$$

$$H, \mathrm{SS}[\texttt{if (}v\texttt{) }\ s_1\ \ s_2] \;\rightarrow\; \begin{cases} H, \mathrm{SS}[s_2] & \text{if } v = 0 \\ H, \mathrm{SS}[s_1] & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{while (}e\texttt{) }\ s] \;\rightarrow\; H, \mathrm{SS}[\texttt{if (}e\texttt{) \{ }\ s\ \texttt{while (}e\texttt{) }\ s\ \texttt{\} \{\}}]$$

$$H, \mathrm{SS}[\texttt{return }\ v\texttt{;}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{if SS}[\cdot] = \mathrm{SE}[\langle\!\langle \mathrm{SS}'[\cdot] \rangle\!\rangle_t], \text{ and SS}'[\cdot] \text{ is in return mode} \\ \mathrm{error}_{\mathrm{return}} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{splice }\ s] \;\rightarrow\; \begin{cases} H, \mathrm{SS}_1[\texttt{\{ }\ s\ \texttt{ cut SS}_2[\{\}]\ \texttt{\}}] & \text{if SS}[\cdot] = \mathrm{SS}_1[\texttt{cut SS}_2[\cdot]], \\ & \quad \text{where SS}_1[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{cut \{\}}] \;\rightarrow\; \begin{cases} H, \mathrm{SS}[\{\}] & \text{if the hole of SS}[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

Figure 12: Rewrite rules of Cyclone's operational semantics

An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework

Germán Puebla and *Manuel Hermenegildo*
Department of Computer Science
Technical University of Madrid (UPM)
{german,herme}@fi.upm.es

John P. Gallagher
Department of Computer Science
University of Bristol
john@cs.bris.ac.uk

Abstract

Information generated by abstract interpreters has long been used to perform program specialization. Additionally, if the abstract interpreter generates a multivariant analysis, it is also possible to perform multiple specialization. Information about values of variables is propagated by simulating program execution and performing fixpoint computations for recursive calls. In contrast, traditional partial evaluators (mainly) use unfolding for both propagating values of variables and transforming the program. It is known that abstract interpretation is a better technique for propagating success values than unfolding. However, the program transformations induced by unfolding may lead to important optimizations which are not directly achievable in the existing frameworks for multiple specialization based on abstract interpretation. The aim of this work is to devise a specialization framework which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation, and which can be implemented via small modifications to existing generic abstract interpreters. With this aim, we will relate top-down abstract interpretation with traditional concepts in partial evaluation and sketch how the sophisticated techniques developed for controlling partial evaluation can be adapted to the proposed specialization framework. We conclude that there can be both practical and conceptual advantages in the proposed integration of partial evaluation and abstract interpretation.

Keywords: Logic Programming, Abstract Interpretation, Partial Evaluation, Program Specialization.

1 Introduction

Partial evaluation [JGS93, DGT96] specializes programs for known values of the input. Partial evaluation of logic programs has received considerable attention [Neu90, LS91, Sah93, Gal93, Leu97] and several algorithms parameterized by different control strategies have been proposed which produce useful partial evaluations of programs. Regarding the correctness of such transformations, two conditions, defined on the set of atoms to be partially evaluated, have been identified which ensure correctness of the transformation: "closedness" and "independence" [LS91].

From a practical point of view, effectiveness, that is, finding suitable control strategies which provide an appropriate level of specialization while ensuring termination, is a crucial problem which has also received considerable attention. Much work has been devoted to the study of such control strategies in the context of "on-line" partial evaluation of logic programs [MG95, LD97, LM96]. Usually, control is divided into components: "local control," which controls the unfolding for a given atom, and "global control," which ensures that the set of atoms for which a partial evaluation is to be computed remains finite.

In most of the practical algorithms for program specialization, the above mentioned control strategies use, to a greater or lesser degree, information generated by static program analysis. One of the most widely used techniques for static analysis is abstract interpretation [CC77, CC92]. Some of the relations between abstract interpretation and partial evaluation have been identified before [GCS88, GH91, Gal92, CK93, PH95, LS96, PH97, Jon97, PGH97, Leu98]. However, the role of analysis is so fundamental that it is natural to consider whether partial evaluation could be achieved directly by a generic, top-down abstract interpretation system such as [Bru91, MH92, CV94]. With this question in mind, we present a method for generating a specialized program directly from the output (an and–or graph) of such a generic, top-down abstract interpreter. We then explore two main questions which arise. First, how much specialization can be performed by an abstract interpreter, compared to on-line partial evaluation? Second, how do the traditional problems of local and global control appear when placed in the setting of generic abstract interpretation? We conclude that there seem to be practical and conceptual advantages in using an abstract interpreter to perform program specialization.

2 Abstract Interpretation

Abstract interpretation [CC77] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to D$.

We recall some classical definitions in logic programming. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. We often use $\overline{t}$ to denote a tuple of terms. A *clause* is of the form H :- $B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. A *definite logic program*, or *program*, is a finite sequence of clauses.

Goal dependent abstract interpretation takes as input a

program P, a predicate symbol[1] p (denoting the exported predicate), and, optionally, a restriction of the run-time bindings of p expressed as an abstract substitution λ in the abstract domain D_α. Such an abstract interpretation computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$, p_i is an atom and λ_i^c and λ_i^c are, respectively, the abstract call and success substitutions. Correctness of abstract interpretation guarantees:

- The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$.
- The abstract call substitutions cover all the concrete calls which appear during execution. $\forall c$ that occurs in the concrete computation of $p\theta$ s.t. $\theta \in \gamma(\lambda)$ where p is the exported predicate and λ the description of the initial calls of p $\exists \langle p_j, \lambda_j^c, \lambda_j^s\rangle \in Analysis(P, p, \lambda, D_\alpha)$ s.t. $c = p_j\theta'$ and $\theta' \in \gamma(\lambda_j^c)$. This property is related to the closedness condition [LS91] required in partial deduction.

As usual in abstract interpretation, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$. A tuple $\langle p_j, \lambda_j^c, \bot\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions.

An analysis is said to be *multivariant on calls* if more than one triple $\langle p, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. Note that if $n = 0$ then the corresponding predicate is not needed for solving any goal in the considered class (p, λ) and is thus dead code and may be eliminated. An analysis is said to be *multivariant on successes* if more than one triple $\langle p, \lambda^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^s \neq \lambda_j^s$ for some i, j may be computed for the same predicate p and call substitution λ^c. Different analyses may be defined with different levels of multivariance [VDCM93]. However, unless the analysis is multivariant on calls, little specialization may be expected in general. Many implementations of abstract interpreters are multivariant on calls. However, most of them (such as PLAI [MH89, MH90, MH92]) are not multivariant on successes, mainly for efficiency reasons. As a result, and as we are interested in reusing existing abstract interpreters for performing partial evaluation, we will limit in principle our discussion to analyses which are multivariant on calls but not on successes. Note that this is not a strong restriction for our purposes as traditional partial evaluation is not multivariant on successes either. Also, code generation from an analysis which is multivariant on successes is not straightforward. However, multivariant successes can in fact be captured by certain abstract domains even if the analysis is not multivariant on successes, as will be discussed in Section 5. Note that for analyses not multivariant on successes when $\langle p, \lambda^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda^c, \lambda_n^s\rangle$ with $n > 1$ have been computed for the same predicate p and call substitution λ^c, the different substitutions $\{\lambda_1^s, \ldots, \lambda_n^s\}$ have to be summarized in a more general one (possibly losing accuracy) λ^s before propagating this success information. This is done by means of the *least upper bound* (lub) operator.[2]

[1] Extending the framework to sets of predicate symbols is trivial.
[2] D_α is a poset.

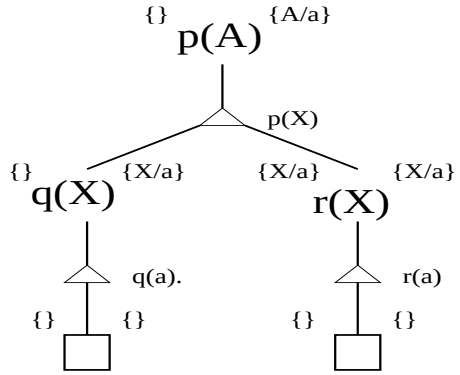

Figure 1: And–or analysis graph

In our case, in order to compute $Analysis(P, p, \lambda, D_\alpha)$, an and–or graph $AO(P, p, \lambda, D_\alpha)$ is constructed which encodes dependencies among the different triples. Such and–or graph can be viewed as a finite representation of the (possibly infinite) set of and–or trees explored by the (possibly infinite) concrete execution [Bru91]. Finiteness of the and–or graph (and thus termination of analysis) is achieved by considering abstract domains with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [CC77]. If it is clear in the context, we will often write AO instead of $AO(P, p, \lambda, D_\alpha)$ for short.

Example 2.1 Consider the simple example program below taken from [Leu97].

```
p(X):- q(X), r(X).
q(a).
r(a).
r(b).
```

We take as initial (exported predicate) the goal $p(A)$ with A unrestricted using the concrete domain as abstract domain. In this case, $Analysis(P, p(A), \{\}, D) = \{\langle p(A), \{\}, \{A/a\}\rangle, \langle q(X), \{\}, \{X/a\}\rangle, \langle r(X), \{X/a\}, \{X/a\}\rangle\}$ and Figure 1 depicts a possible and–or analysis graph. □

Due to space limitations, and given that it is now well understood, we do not describe in detail here how to build analysis and–or graphs. More details can be found in [Bru91, MH90, MH92, HPMS95, PH96]. The graph has two sorts of nodes: those which correspond to atoms (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$. As before, λ_i^c and λ_i^s are, respectively, a pair of abstract call and success substitutions for the atom p_i. For clarity, in the figures the atom p_i is superscripted with λ^c to the left and λ^s to the right of p_i respectively. For example, the or–node $\langle p(A), \{\}, \{A/a\}\rangle$ is depicted in the figure as ${}^{\{\}}p(A)^{\{A/a\}}$. And–nodes are pairs $\langle Id, H\rangle$ where Id is a unique identifier for the node and H is the head of the clause the node refers to. In the figures, they are represented as triangles and H is depicted to the right of the triangles. Note that the substitutions (atoms) labeling and–nodes are concrete whereas the substitutions labeling or–nodes are abstract. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs

Algorithm 2.2 [Code Generation] Given $Analysis(P, p, \lambda, D_\alpha)$ and $AO(P, p, \lambda, D_\alpha)$ generated by analysis for a program P and an atomic goal $\leftarrow p$ with abstract substitution $\lambda \in D_\alpha$, and a partial concretization function $part\_conc$ do:

- For each tuple $N = \langle a(\overline{t}), \lambda^c, \lambda^s \rangle \in Analysis(P, p, \lambda, D_\alpha)$ generate a distinct predicate with name $pred_N = \mathsf{name}(\langle a(\overline{t}), \lambda^c, \lambda^s \rangle)$.
- Each predicate $pred_N$ is defined by
  - $pred_N(\overline{t})$:- $fail$ if $\lambda^s = \bot$
  - $(pred_N(\overline{t}_1)$:- $b'_1)\theta_1 :: \ldots :: (pred_N(\overline{t}_n)$:- $b'_n)\theta_n$ otherwise
    where $expansion(N, AO) = O_N$ and
    $children(O_N, AO) = \langle Id_1, p_1(\overline{t}_1) \rangle :: \ldots :: \langle Id_i, p_i(\overline{t}_i) \rangle :: \ldots :: \langle Id_n, p(\overline{t}_n) \rangle$
- Each body b'_i is defined as
  - $b'_i = fail$ if $\lambda^s_{ik_i} = \bot$
  - $b'_i = (pred_{i1}(\overline{t}_{i1}), \ldots, pred_{ik_i}(\overline{t}_{ik_i}))$ otherwise
    where $pred_{ij} = \mathsf{name}(\langle a_{ij}(\overline{t}_{ij}), \lambda^c_{ij}, \lambda^s_{ij} \rangle)$, and
    $children(\langle Id_i, p(\overline{t}_i) \rangle, AO) = \langle a_{i1}(\overline{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1} \rangle :: \ldots :: \langle a_{ik_i}(\overline{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i} \rangle$.
- Each substitution θ_i is defined as
  - $\theta_i = \epsilon$ if $b'_i = fail$
  - $\theta_i = mgu(std(part\_conc(\lambda^s_{i1})), \ldots, std(part\_conc(\lambda^s_{ik_i})))$ otherwise

Figure 2: Algorithm for Code Generation

to and–nodes which represent the clauses with which the atom (possibly) unifies. And–nodes have arcs to or–nodes which represent the atoms in the body of the clause. Note that several instances of the same clause may exist in the analysis graph of a program. In order to avoid conflicts with variable names, clauses are standardized apart before adding to the analysis graph the nodes which correspond to such clause. This way, only nodes which belong to the same clause may share variables. As the head of the clause (after the standardizing renaming transformation) is stored in the and–node, we can always reconstruct (a variant of) the original clause when generating code from an and–or graph (see Section 3 below).

Intuitively, analysis algorithms are just graph traversal algorithms which given P, p, λ, and D_α build $AO(P, p, \lambda, D_\alpha)$ by adding the required nodes and computing success substitutions until a global fixpoint is reached. For a given P, p, λ, and D_α there may be many different analysis graphs. However, there is a unique *least analysis graph* which gives the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations. Each time the analysis algorithm creates a new or–node for p and λ^c and before computing the corresponding λ^s, it checks whether $Analysis(P, p, \lambda, D_\alpha)$ already contains a tuple for (a variant of) p and λ^c. If that is the case, the or–node is not expanded and the already computed λ^s stored in $Analysis(P, p, \lambda, D_\alpha)$ is used for that or–node. This is done both for efficiency and for avoiding infinite loops when analyzing recursive predicates. As a result, several instances of the same or–node may appear in AO, but only one of them is expanded. We denote by $expansion(N)$ the instance of the or–node N which is expanded. If there is no tuple for p and λ^c in $Analysis(P, p, \lambda, D_\alpha)$, the or–node is expanded, λ^s computed, and $\langle p, \lambda^c, \lambda^s \rangle$ added to $Analysis(P, p, \lambda, D_\alpha)$. Note that the success substitutions λ^s stored in $Analysis(P, p, \lambda, D_\alpha)$ are tentative and may be updated during analysis. Only when a global fixpoint is reached the success substitutions are safe approximations of the concrete success substitutions.

3 Code Generation from an And–Or Graph

The information in $Analysis(P, p, \lambda, D_\alpha)$ has long been used for program optimization. Multiple specialization is a program transformation technique which allows generating several versions $p_1, \ldots, p_n$ $n \geq 1$ for a predicate p in P. Then, we have to decide which of $p_1, \ldots, p_n$ is appropriate for each call to p. One possibility is to use run-time tests to decide which version to use. If analysis is multivariant on calls but not on successes, another possibility, as done in [Win92, PH95], is to generate code from $AO(P, p, \lambda, D_\alpha)$ instead of $Analysis(P, p, \lambda, D_\alpha)$. The arcs in $AO(P, p, \lambda, D_\alpha)$ allow determining which p_i to use at each call. Then, each version of a predicate receives a unique name and calls are renamed appropriately.

After introducing some notation, an algorithm which generates a logic program from an analysis and–or graph is presented in Figure 2 (Algorithm 2.2). Given a non-root node N, we denote by $parent(N, AO)$ the node $M \in AO$ such that there is an arc from M to N in AO, and $children(N, AO)$ is the sequence of nodes $N_1 :: \ldots :: N_n$ $n \geq 0$ such that there is an arc from N to N' in AO iff $N' = N_i$ for some i and $\forall i, j = 0, \ldots, n$ N_i is to the left of N_j in AO iff $i < j$. Note that $children(N, AO)$ may be applied both to or– and and–nodes. We assume the existence of an injective function name which given $Analysis(P, p, \lambda, D_\alpha)$ returns a unique predicate name for each tuple and $\mathsf{name}(\langle q(\overline{t}), \lambda^c, \lambda^s \rangle) = q$ iff $q(\overline{t}) = p$ (the exported predicate) and $\lambda^c = \lambda$ (the restriction on initial goals), to ensure that top-level – exported

– predicate names are preserved.

Definition 3.1 [partial concretization] A function $part\_conc : D_\alpha \rightarrow D$ is a *partial concretization* iff $\forall \lambda \in D_\alpha \; \forall \theta' \in \gamma(\lambda) \; \exists \theta''$ s.t. $\theta' = part\_conc(\lambda)\theta''$.
$part\_conc(\lambda)$ can be regarded as containing (part of) the definite information about concrete bindings that the abstract substitution λ captures. Note that different partial concretizations of an abstract substitution λ with different accuracy may be considered. For example if the abstract domain is a depth-k abstraction and $\lambda = \{X/f(f(Y)) or X/f(a)\}$, a most accurate $part\_conc(\lambda)$ is $\{X/f(Z)\}$. Note also that $part\_conc(\lambda) = \epsilon$ where ϵ is the empty substitution, is a trivially correct partial concretization of any λ.

Definition 3.2 [specialization] Let P be a definite program. Let $AO(P, p, \lambda, D_\alpha)$ be an and–or graph. We say that a program P' is a *specialization* of P w.r.t. $AO(P, p, \lambda, D_\alpha)$ and $part\_conc$ and we denote it $P' = spec(AO(P, p, \lambda, D_\alpha), part\_conc)$ iff P' can be obtained by applying Algorithm 2.2 to $AO(P, p, \lambda, D_\alpha)$ using $part\_conc$.

Basically, Algorithm 2.2 for code generation creates a different version for each different (abstract) call substitution λ^c to each predicate p_i in the original program. This is easily done by associating a version to each or–node. Note that if we always take the trivial substitution ϵ as $part\_conc(\lambda)$ for any λ (such as in [PH95]) then such versions are identical except that atoms in clause bodies are renamed to always call the appropriate version.[3] The interest in performing the proposed multiple specialization is that the new program may be subject to further optimizations, such as elimination of redundant type/mode checks, which are allowed in the multiply specialized program because now each version it to be used for a more restricted set of input values than in the original program. Additionally, in Algorithm 2.2 predicates whose success substitution is $\perp$ are directly defined as $p(\overline{t}) : -fail$, as it is known that they produce no answers. Even if the success substitution λ^s for $\langle p, \lambda^c, \lambda^s \rangle$ is not $\perp$, individual clauses for p whose success substitution is $\perp$ (useless clauses) for the considered λ^c are removed from the final program.

By *mgu* we denote, as usual, the *most general unifier* of substitutions. *std* represents the result of standardizing apart the results of $part\_conc$ in order to avoid undesired variable name clashes. Note that in Algorithm 2.2 atoms are specialized w.r.t. answers rather than calls, as is the case in traditional partial evaluation. This will in general provide further specialized (and optimized) programs as in general the success substitution (which describes answers) computed by abstract interpretation is more informative (restricted) than the call substitution. However, this cannot be done for example if the program contains calls to extra-logical predicates such as `var/1`. Other more conservative algorithms can be used for such cases and for programs with side-effects. Using Algorithm 2.2 it is sometimes possible to detect infinite failures of predicates and replace predicate definitions and/or clause bodies by `fail`, which is not possible in partial evaluation, as the number of unfolding steps must be finite. Additionally, as mentioned above, dead code, i.e., clauses not used to solve the considered goal, are removed.

Note that Algorithm 2.2 is an improvement over the code-generation phase of [PH95, PH97] in that it allows applying non-trivial partial concretizations of the abstract

[3] The program obtained in this way is $program_0$ in the notation of [PH95].

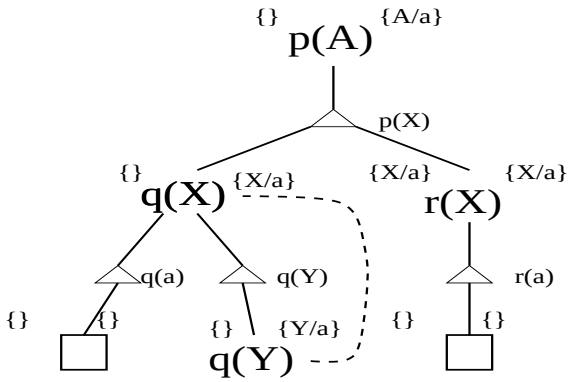

Figure 3: And–or analysis graph for a recursive program

(success) substitutions. The program obtained by Algorithm 2.2 can then be further optimized by applying the notion of *abstract executability* as presented in [PH97], which reduces an atom w.r.t. an abstract substitution.

Theorem 3.3 Let $AO(P, p, \lambda, D_\alpha)$ be an analysis and–or graph for a definite program P and an atomic goal $\leftarrow p$ with the abstract call substitution $\lambda \in D_\alpha$. Let P' be the program obtained from $AO(P, p, \lambda, D_\alpha)$ by Algorithm 2.2. Then $\forall \theta_c$ s.t. $\theta_c \in \gamma(\lambda)$

i) $p\theta_c$ succeeds in P' with computed answer θ_s iff $p\theta_c$ succeeds in P with computed answer θ_s.

ii) if $p\theta_c$ finitely fails in P then $p\theta_c$ finitely fails in P'.

Thus, both computed answers and finite failures are preserved. However, the specialized program may fail finitely while the original one loops (see Example 4.2).

4 And–Or Graphs Vs. SLD Trees

It is known [LS96] that the propagation of success information during partial evaluation is not optimal compared to that potentially achievable by abstract interpretation.

Example 4.1 Consider the program of Example 2.1. The program obtained by applying Algorithm 2.2 to the and–or graph in Figure 1 is:

```
p(a):- q(a), r(a).
q(a).
r(a).
```

Note that Algorithm 2.2 may perform some degree of specialization even if no unfolding is performed. The information in $AO(P, p, \lambda, D_\alpha)$ allows determining that the call to `r(X)` will be performed with `X=a` and thus the second clause for `r` can be eliminated. Such information can only be propagated in partial evaluation by unfolding the atom `q(X)`. □

Example 4.2 Consider again the goal and program of Example 2.1 to which a new clause `q(X):- q(X).` is added for

predicate `q`. The and–or graph for the new program is depicted in Figure 3. The dotted arc indicates that the corresponding or–nodes have equivalent abstract call substitution. However, the set of tuples in $Analysis(P', p(A), \{\}, D)$ for the current program P' is exactly the same as in Example 2.1, in spite of the more involved and–or graph in this example. The program generated for this graph by Algorithm 2.2 is the following:

```
p(a):- q(a), r(a).
q(a).
q(a):- q(a).
r(a).
```

The fact that `r(X)` will only be called with `X=a` cannot be determined by any finite unfolding rule. Note that the original program loops for the goal ← `p(b)` while the specialized one fails finitely. □

The two examples above show that and–or graphs allow a level of success information propagation not possible in traditional partial evaluation, either because the unfolding rule is not aggressive enough (Example 4.1) or because the required unfolding would be infinite (Example 4.2). This observation already provides motivation for studying the integration of full partial evaluation in an analysis/specialization framework based on abstract interpretation.

In addition, the fact that such a framework can work uniformly with abstract or concrete substitutions makes it more general than partial evaluation and may allow performing optimizations not possible in the traditional approaches to partial evaluation. An additional pragmatic motivation for this work is the availability of off-the-shelf generic abstract interpretation engines such as PLAI [MH92] or GAIA [CV94] which greatly facilitate the efficient implementation of analyses. The existence of such an abstract interpreter in advanced optimizing compilers is likely, and using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler.

5 Partial Evaluation using And–Or Graphs

We have established so far that for a given abstract interpretation of a program in a system such as PLAI (even interpretations over very simple domains such as modes) we can get some corresponding specialized source program with possibly multiple versions by applying Algorithm 2.2. Correctness of abstract interpretation ensures that the set of triples computed by analysis must cover all calls performed during execution of any instance of the given initial goal (p, λ). This condition is strongly related to the closedness condition of partial evaluation [LS91]. Furthermore there are well-understood conditions and methods for ensuring termination of an abstract interpretation.

Thus, an important conceptual advantage of formalizing partial evaluation in terms of abstract interpretation is that two of the main concerns of partial evaluation algorithms – namely, correctness and termination – are treated in a very general and flexible way by the general principles, methods, and formal results of abstract interpretation. The other important concern is the degree of specialization that is achieved, which is determined in partial evaluation by the local and global control. We now examine how these control issues appear in the setting of abstract interpretation.

5.1 Global Control in Abstract Interpretation

Effectiveness of program specialization greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which (specialized) code is to be generated. In partial evaluation, this mainly depends on the global control used. If we use the specialization framework based on abstract interpretation, the number of specialized versions depends on the number of or–nodes in the analysis graph. This is controlled by the choice of abstract domain and widening operators (if any). The finer-grained the abstract domain is, the larger the set $\mathbf{A}$ will be. In conclusion, the role of so-called global control in partial evaluation is played in abstract interpretation by our particular choice of abstract domain and widening operators (which are strictly required for ensuring termination when the abstract domain contains ascending chains which are infinite – as is the case for the concrete domain).

Note that the specialization framework we propose is very general. Depending on the kind of optimizations we are interested in performing, different domains (and widening operators) should be used and thus different $\mathbf{A}$ sets would be obtained. For example, if we are interested in eliminating redundant groundness tests, our abstract domain could in principle collapse the two atoms $p(1)$ and $p(2)$ into one $p(ground)$ since, from the point of view of the optimization, whether p is called with the value 1 or 2 is not relevant.

While the main aim of global control is to ensure termination and to avoid generating too many superfluous versions, it may often be the case that global control (or the domain) does not collapse two versions in the hope that they will lead to different optimizations. If this is not the case, a minimizing step may be performed a posteriori on the and–or graph in order to produce a minimal number of versions while maintaining all optimizations. This was proposed in [Win92], implemented in [PH95] and also discussed in [LM95]. We intend to extend the minimizing algorithm in [PH95] for the case of optimizations based on unfolding.

5.2 Local Control in Abstract Interpretation

Local control in partial evaluation determines how each atom in $\mathbf{A}$ should be unfolded. However, in traditional frameworks for abstract interpretation we usually have a choice for abstract domain and widening operators, but no choice for local control is offered. This is because by default, in abstract interpretation each or–node is related by just one (abstract) unfolding step to its children. This corresponds to a trivial local control (unfolding rule) in partial evaluation.

Unfolding is a well known program transformation technique in which an atom in the body of a clause, which can be seen as a call to a procedure, is replaced by the code of such procedure. We now introduce the notion of *node-unfolding* which is a graph transformation technique which given an and–or graph AO and an or–node N in AO builds a new and–or graph AO'. Such graph transformation mimics the effect of unfolding an atom in a program.

Definition 5.1 [clause-unfolding]
Let $A = \langle Id, H \rangle$ be an and–or node in $AO(P, p, \lambda, D_\alpha)$ s.t. $children(A, AO) = L_1 :: \ldots :: N :: \ldots L_m$ with $m \geq 1$ and $N = \langle a, \lambda^c, \lambda^s \rangle$. Let also $C = H_C$ `:-` $B_1, \ldots, B_n$ be a clause in program P whose head H_C unifies with atom a.

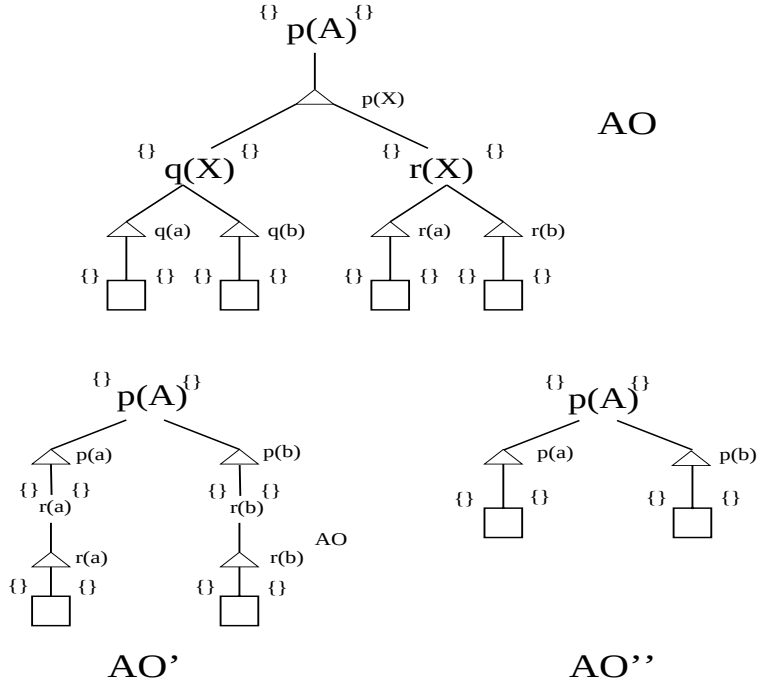

Figure 4: Example Node Unfoldings

The *clause-unfolding* of A and N w.r.t. C, denoted $cl\_unf(A, N, C)$, is the (partial) and–or graph AO' with root $A' = \langle Id, H\theta \rangle$ such that θ is a mgu of a and H_C and $children(A', AO') = L'_1 :: \ldots :: N_1 :: \ldots N_n :: \ldots L'_m$.

Each or-node N_j is of the form $\langle B_j, \lambda^c_j, \lambda^s_j \rangle$, where λ^c_j and λ^s_j have to be computed by the analysis algorithm as usual. Provided that $L_i = \langle p_i, \lambda^c_i, \lambda^s_i \rangle$ then, $L'_i = \langle p_i\theta, Aadd(\theta, \lambda^c_i), Aadd(\theta, \lambda^s_i) \rangle$ where $Aadd(\theta, \lambda)$ updates the abstract substitution λ by conjoining it with the concrete substitution θ (see for example [HPMS95]). A discussion on the effects of performing such conjoining (accuracy vs. efficiency) can be found in Section 5.3.

As usual, *mgu* denotes a most general unifier, in this case of two atoms. Clause-unfolding mimics the effect of an SLD resolution step.

Definition 5.2 [node-unfolding]

Let $N = \langle a, \lambda^c, \lambda^s \rangle$ be a non-empty or–node in AO. Let $C_1 :: \ldots :: C_n$ $n \geq 1$ be the sequence of standardized apart clauses in program P s.t. a unifies with the head of C_i. Let $parent(N, AO) = A$, and let $parent(A, AO) = GP$ with $children(GP, AO) = P_1 :: \ldots :: A :: \ldots P_i$, $i \geq 1$.

The *node-unfolding* of AO w.r.t. N, denoted *node-unfolding*(AO, N) is the and–or graph AO' obtained from AO by making $children(GP, AO')$ be $P_1 :: \ldots :: cl\_unf(A, N, C_1) :: \ldots :: cl\_unf(A, N, C_n) :: \ldots P_i$ and eliminating nodes A and N.

Node unfolding is achieved by performing clause-unfolding with all clauses in the program whose head unifies with the unfolded atom.

Theorem 5.3 [node-unfolding] Let $AO(P, p, \lambda, D_\alpha)$ be a (partially computed) and–or graph. Let *part_conc* be a partial concretization function, let $P_{AO} = spec(AO(P, p, \lambda, D_\alpha),$ *part_conc*). Let N be an or–node in AO, let $AO' =$ *node-unfolding*(AO, N), and let $P' = spec(AO', part\_conc)$. Then $\forall \theta_c$ s.t. $\theta_c \in \gamma(\lambda)$

i) $p\theta_c$ succeeds in P' with computed answer θ_s iff $p\theta_c$ succeeds in P_{AO} with computed answer θ_s.

ii) if $p\theta_c$ finitely fails in P_{AO} then $p\theta_c$ finitely fails in P'.

Theorem 5.3 guarantees correctness of node-unfolfing as it states that performing node-unfolding on an and–or graph preserves computed answers and finite failures.

Example 5.4 Reconsider the program of Example 2.1 in which an additional clause `q(b).` has been added to predicate q. The new analysis graph generated without performing any node-unfolding is shown in Figure 4 as AO, using the concrete domain as abstract domain and the *most specific generalization* (msg) as lub operator for summarizing different success substitutions into one. As discussed in Section 5.5 below, the msg is a rather crude lub operator. However, we use it for the sake of clarity of the example. AO' is an analysis graph for the same program but this time the or–node $\langle q(X), \{\}, \{\} \rangle$ has been unfolded. Finally, graph AO'' in the figure is the result of applying node-unfolding twice to AO', once w.r.t. $\langle p(a), \{\}, \{\} \rangle$ and another one w.r.t $\langle p(b), \{\}, \{\} \rangle$. The code generated by $spec(AO'', part\_conc)$ (for any *part_conc*) is the program:

```
p(a).
p(b).
```

5.3 Strategies for Local Control

Several possibilities exist in order to overcome the simplicity of the local control performed by abstract interpretation:

1. According to many authors, [Gal93, LM96] global control is much harder than local control. Thus, one possibility is to obtain AO using the traditional analysis algorithm. Subsequent unfolding $spec(AO, part\_conc)$ can be done using traditional unfolding rules in order to eliminate determinate calls or some non-recursive calls, for example. The and–or analysis graph AO may be of much help in order to detect such cases.

2. A second alternative it to use abstract domains for analysis which allow propagating enough information about the success of an or–node so as to perform useful specialization on other or-nodes. This requires that the lub operator not lose "much" information, for example by allowing sets of abstract substitutions. The advantage of this method is that no modification of the abstract interpretation framework is required. Also, as we will see in Example 5.5, it may allow specializations which are not possible by the methods proposed below (nor by traditional partial evaluation).

3. Another possibility is a simple modification to the algorithm for abstract interpretation in order to accommodate a node-unfolding rule. In this approach, if the node-unfolding rule decides that an or–node N in a graph AO should not be unfolded, then N is treated as in the traditional abstract interpretation algorithm. If the rule decides that N should be unfolded, $AO' = \mathit{node\text{-}unfolding}(AO, N)$ is computed and analysis continues for the new graph AO'. Note that in order to unfold N it is not required to know its success substitution. Thus, the graph transformation associated to an unfolding is merely structural and can be performed before or after computing the and–or graph below N. If decisions on unfolding are taken before computing the nodes below N, the unfolding rule corresponds to those used in partial evaluation: only the history of nodes higher in the top-down algorithm is available for deciding whether to unfold or not. However, we can delay this decision until the graph below N has been computed. This allows making better decisions as also the specialization history of atoms lower in the hierarchy is known.

   In the latter case, if N is not the leftmost atom in its clause and the abstract domain is downwards closed, there is a choice of whether to apply the (sequence of) success substitution(s) for N to the sibling nodes $L_1 :: \ldots :: L_k$ to the left of N (i.e., to perform left propagation) and reanalyze such nodes with the better information or not. Both alternatives are correct. The second alternative may allow better analysis and specialization, but at a higher computational cost.

4. The last possibility we propose is to first compute AO with a trivial unfolding rule (i.e., using the traditional abstract interpretation algorithm). Once analysis has finished, further unfolding may be performed if desired, as done in the first alternative proposed. However, unlike the first approach, rather than performing unfolding externally to the analysis and without modifying the analysis graph, whenever an unfolding step is performed for a node N in AO, a new analysis graph $AO' = \mathit{node\text{-}unfolding}(AO, N)$ is computed.

   This approach can be seen as an extreme case of delaying node-unfolding, not only until the graph below N has been computed (as mentioned in the third approach) but rather until a global fixpoint has been reached for the whole analysis graph. The difference with the third approach is that there, unfolding is completely integrated in abstract interpretation and the local control decisions are taken when performing analysis and as mentioned before, the only issue is whether to perform left propagation of bindings or not. The advantage over the previous approach is that unfolding is performed once the whole analysis graph has been computed. The benefits of the availability of such better information for local control still have to be explored. The disadvantage is that in order to achieve as accurate information as possible it may be required to perform reanalysis in order to propagate the improved information introduced due to the additional unfolding steps, i.e., using $\langle p_i\theta, Aadd(\theta, \lambda_i^c), Aadd(\theta, \lambda_i^s)\rangle$ instead of $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$ for the nodes to the right of the unfolded node, with the associated computational cost. This cost could however be kept quite reasonable by the use of incremental analysis techniques such as those presented in [HPMS95, PH96].

Example 5.5 Consider the following program and the goal ←`r(X)`

```
r(X) :- q(X),p(X).
q(a).
q(f(X)) :- q(X).
p(a).
p(f(X)) :- p(X).
p(g(X)) :- p(X).
```

The third clause for `p` can be eliminated in the specialized program for ←`r(X)`, provided that the call substitution for `p(X)` (i.e., the success substitution for `q(X)`) contains the information that `X=a` or `X=f(Y)`. The abstract domain has to be precise enough to capture, in this case, at least the set of principal functors of the answers.

Note that no partial evaluation algorithm based on unfolding will be able to eliminate the third clause for `p`, since an atom of form `p(X)` will be produced, no matter what local and global control is used.[4] Thus, simulating unfolding in abstract interpretation (such as methods 1, 3, and 4 above do) will not achieve this specialization either. An approach such as 2 is required. □

5.4 Abstract Domains and Widenings for Partial Evaluation

Once we have presented the relation between abstract domains and widening with global control in partial evaluation, in this section we discuss desired features for performing partial evaluation. Ideally, we would like that

- The domain can simulate the effect of unfolding, which is the means by which bindings are propagated in partial evaluation. Our abstract domain has to be capable of tracking such bindings. This suggests that domains based on term structure are required.

- In addition, the domain needs to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation in order

[4] Conjunctive partial deduction [LSdW96] can solve this problem in a completely different way.

to achieve the approach 2 for local control. A term domain whose least upper bound is based on the *msg* (most specific generalization), for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

Two examples of classes of domain which have the above desirable features are:

- The domain of type-graphs [BJ92], [GdW94], [HCC94]. Its drawback is that inter-argument dependencies are lost.
- The domain of sets of depth-k substitutions with set union as the least upper bound operator. However uniform depth bounds are usually either too imprecise (if k is too small) or generate much redundancy if larger values of k are chosen.

One way to eliminate the depth-bound k in the abstract domain it to depend on a suitable widening operator which will guarantee that the set of or–nodes remains finite. Many techniques have been developed for global control of partial evaluation. Such techniques make use of data structures which are very related to the and–or analysis graph such as *characteristic trees* [GB91], [Leu95] (related to *neighborhoods* [Tur88]), *trace-terms* [GL96], and *global trees* [MG95], and combinations of them [LM96]. Thus, it seems possible to adapt these techniques to the case of abstract interpretation and formalize them as widening operators.

6 Related Work

The integration of partial evaluation and abstract interpretation has been attempted before, both from a partial evaluation and abstract interpretation perspective. In [GCS88, Gal92] such an integration is attempted from the point of view of partial evaluation. However, the approach is only partially successful as the resulting specialization framework does not exploit the full power of abstract interpretation. Another attempt for functional rather than logic programs is presented in [CK93].

From an abstract interpretation perspective, the integration has also received considerable attention. In [GH91], abstract interpretation is used to perform multiple specialization in an ad-hoc way. Also in [GH91] the notion of *abstract executability* is presented (and later formalized in [PH97]) and applied to remove redundant builtin checks. The first complete framework for multiple specialization based on abstract interpretation is presented in [Win92]. The first implementation and experimental evaluation is presented in [PH95] together with a framework based on existing abstract interpreters. All these techniques, even though they allow important specializations often not achievable by partial evaluation, are not designed for performing unfolding, which is one of the basic optimization techniques used by partial evaluators.

On the other hand, the drawbacks of traditional partial evaluation techniques for propagating success information are identified in [LS96] and some of the possible advantages of a full integration of partial evaluation and abstract interpretation are presented in [Jon97].

To the best of our knowledge, the first framework which presents a full integration of abstract interpretation and partial evaluation is [PGH97], on which this paper is based. More recently, a different formulation of such an integration has been presented in [Leu98]. In this formulation a top-down specialization algorithm is presented which assumes the existence of an *abstract unfolding function*, possibly not based on concrete unfolding, which generalizes existing algorithms for partial evaluation. Rather strong conditions are assumed over the behaviour of the abstract unfolding function. Unfortunately, no method is given for computing interesting ones except by providing relatively simple examples based on concrete unfolding. Also, the top-down algorithm proposed suffers from the same problems as traditional partial evaluation: lack of success propagation. This problem is solved by integrating the top-down algorithm with a bottom-up abstract interpretation algorithm which approximates success patterns. Note that this alternative corresponds to alternative 3 for local control (see Section 5.2, or [PGH97]). The main difference is that in our approach a single (and already existing) top-down abstract interpretation algorithm augmented with an unfolding rule performs propagation of both the call and success patterns in an integrated fashion.

Another difference between the two approaches is that [Leu98] is capable of dealing with conjunctions and not only atoms. This allows conjunctive partial evaluation [LSdW96] but adds an additional level of complexity to the control of program specialization: in order to guarantee termination a mechanism needs to be provided for deciding when and how to split conjunctions into components. While a similar form of conjunctive partial evaluation could be easily included in our framework, there is another pragmatic reason for not doing so: in general, existing abstract interpreters (and partial evaluators) only analyze (specialize) atoms individually, and we aim at reusing as much of existing analyzers as possible, an objective which is a further difference between our work and [Leu98].

7 Conclusions

We have proposed an integration of traditional partial evaluation into standard, generic, top-down abstract interpretation frameworks. We now summarize the main conclusions which can be derived from this work. As seen in [PH95], a multiply specialized program can be associated to every abstract interpretation which is multivariant on calls. Abstract interpretation can be regarded as having the simple local control strategy of always performing one unfolding step. However, useful specialization can be achieved if the global control is powerful enough. The global control is closely related to the abstract domain which is used, since this determines the multivariance of the analysis. If the abstract domain is finite (as is often the case), global control may simply be performed by the abstraction function of the abstract domain. However, if the abstract domain is infinite (as is required for partial evaluation), global control has to be augmented with a widening operator in order to ensure termination. The strategies for global control used in partial evaluation, such as those based on characteristic trees [GB91, LD97], on global trees [MG95], and on combinations of both [LM96], are then applicable to abstract interpretation. We have discussed different alternatives for introducing more powerful local unfolding strategies in abstract interpretation, such as unfolding the specialized program derived from abstract interpretation, or incorporating unfolding into the analysis algorithm. In the latter case, it

can be proved that the set of atoms $\mathbf{A}_{AI}$ computed by abstract interpretation is as good as or better approximation of the computation than the set of atoms $\mathbf{A}_{PE}$ computed by traditional on-line partial evaluation with the corresponding global and local control, due to the better success propagation of abstract interpretation.

Acknowledgments

This work was supported in part by ESPRIT project DiSCiPl and CICYT project ELLA. Part of this work was performed while Germán Puebla was visiting the University of Bristol supported the Human Capital and Mobility Network *Logic Program Synthesis and Transformation*. The authors would like to thank the anonymous referees for useful comments.

References

[BJ92] M. Bruynooghe and G. Janssens. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[CK93] C. Consel and S.C. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[CV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*. Number 1110 in LNCS. Springer, February 1996. Dagstuhl Seminar.

[Gal92] J.P. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.

[Gal93] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GL96] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag Lecture Notes in Computer Science, 1996.

[HCC94] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

[HPMS95] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall, New York, 1993.

[Jon97] N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *Static Analysis Symposium*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.

[LD97] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accepted for Publication in New Generation Computing.

[Leu95] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1995.

[Leu97] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.

[Leu98] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.

[LM95] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. Technical Report CW 220, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1995.

[LM96] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[LS96] Michael Leuschel and De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996.

[LSdW96] M. Leuschel, D. De Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[MG95] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

[Neu90] G. Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 205–217, Leuven, Belgium, 1990. K. U. Leuven.

[PGH97] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PH97] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *VI International Workshop on Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[Sah93] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[Tur88] V. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[VDCM93] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.

Diffusion: Calculating Efficient Parallel Programs

Zhenjiang Hu, Masato Takeichi, Hideya Iwasaki

Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo, Tokyo 113, Japan
({hu,takeichi,iwasaki}@ipl.t.u-tokyo.ac.jp)

Abstract

Parallel primitives (skeletons) intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process easier. However, programmers often suffer from the difficulty to choose a combination of proper parallel primitives so as to construct efficient parallel programs. To overcome this difficulty, we shall propose a new transformation, called *diffusion*, which can efficiently decompose a recursive definition into several functions such that each function can be described by some parallel primitive. This allows programmers to describe algorithms in a more natural recursive form. We demonstrate our idea with several interesting examples. Our diffusion transformation should be significant not only in development of new parallel algorithms, but also in construction of parallelizing compilers.

Keywords: Bird Meertens Formalisms, Data Parallelism, Parallelization, Skeletal Parallel programming

1 Introduction

Data parallelism is currently one of the most successful models for programming massively parallel computers, compared with *control parallelism* that is explored from the control structures [Pra92]. To support parallel programming, this model basically consists of two parts, namely

- *a parallel data structure* to model a uniform collection of data which can be organized in a way that each of its elements can be manipulated in parallel;
- *a set of parallel primitives* on the parallel data structure to capture parallel skeletons of interest, which can be used as building blocks to write parallel programs.

For instance, in the parallel language Nesl [Ble92], the parallel data structure is sequences, and the most important parallel primitives on sequences are *apply-to-each* and *scan*; and in the BMF parallel model [Bir87, Ski94], the parallel data structure is parallel lists, and the parallel primitives are mainly *map* and *reduce*.

This parallel model not only provides the programmer an easily understandable view of a single execution stream of a parallel program, but also makes the parallelization process easier because of explicit parallelism in the parallel primitives [HS86, Kar87, HL93].

Despite these promising features, the application of current data parallel programming suffers from a problem. Because parallel programming relies on a set of parallel primitives to specify parallelism, programmers often find it hard to choose proper parallel primitives and to integrate them well in order to develop efficient parallel programs. Consider, as an example, that we want to develop an efficient parallel program for the bracket matching problem [Col95, HT99]. This is a kind of language recognition problem, determining whether the brackets of many types in a given string are correctly matched. For example, the string "$g + \{[o + o] * d\}()$" is well matched, whereas "$b\{[a)d]$" is not. The problem itself is so simple, but it is far from being that simple to develop an *efficient* parallel program in terms of the specified set of parallel primitives, say only with map, reduce and scan.

Nevertheless, a simple straightforward sequential algorithm still exists by using a stack. Opening brackets are pushed, and each closing bracket is matched against the current stack top. Failure is indicated by a mismatch, by an empty stack when a match is required, or by a nonempty stack at the end of the scan of the input. Thus we come to the following naive program.

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a:x)\ s & = & \textbf{if}\ isOpen\ a\ \textbf{then}\ bm\ x\ (push\ a\ s) \\
 & & \textbf{else if}\ isClose\ a\ \textbf{then} \\
 & & \quad noEmpty\ s\ \wedge \\
 & & \quad match\ a\ (top\ s)\ \wedge \\
 & & \quad bm\ x\ (pop\ s) \\
 & & \textbf{else}\ bm\ x\ s
\end{array}
$$

A possible way to programming with parallel primitives is to use the multi-pass programming method, well-known in the sequential functional programming community [BW88]. We start by a naive specification (without any concern of efficiency) of the problem by a composition of several passes so that each pass can be described in terms of the parallel primitives, and then we optimize it by correctness-preserving program transformation. To be more concrete, an initial naive

program in terms of the parallel primitives may be something like

$$prog \;=\; \cdots\ scan\ (\otimes)\ \circ\ reduce\ (\oplus)\ \cdots \\ scan\ (\odot)\ \circ\ map\ f\ \cdots$$

This could be quite inefficient if expensive operations of f, $\otimes$, $\oplus$ and $\odot$ are used in the parallel primitives. To make it efficient, in the sequential programming we can adopt the fusion (deforestation) transformation [Wad88, Chi92] to merge several passes into a single pass resulting in a compact recursive program; but in the parallel programming we cannot do so easily. The major difficulty lies in the primitive-closed requirement that the fusion of two primitives should give a primitive again. In fact, it is known to be hard to establish a set of efficient calculational rules for such kind of fusion transformation [Bir87], and to construct a suitable cost model to guide derivation of efficient parallel programs [Ski94].

In this paper, we shall propose a new transformation, called *diffusion*, to calculate efficient parallel programs in terms of a fixed set of parallel primitives. In contrast to the well-known *fusion* transformation, diffusion efficiently decomposes a recursive definition into several functions such that each function can be described using a parallel primitive, allowing programmers to define their algorithms in a natural recursive form. We shall adopt Bird Meertens Formalisms as our abstract parallel computation model [Bir87, Ski94]. Our main contributions are as follows.

- We propose a novel theorem for diffusion transformation (Section 3.1) in a calculational form [THT98]. Our diffusion theorem integrates the existing parallelization technique [HTC98], and generalizes the well-known homomorphism lemma [Bir87, Col95], with a nice use of scan to deal with accumulating parameters in recursive definitions.
- Our diffusion transformation can be applied to a wide class of recursive definitions. In fact, the recursive form in the diffusion theorem is very natural in its own right. Moreover, as illustrated in derivation of an efficient parallel program for bracket matching in Section 3.2, if combined with the normalization algorithm [HTC98] which is based on fusion and tupling calculation, a wider class of recursive definitions can be turned into the form to which our diffusion theorem can be applied.
- We highlight how to generalize our idea from recursive definitions on lists to those on other data structures like trees. This indicates a new way to do parallel programming efficiently over trees or more general data structures, which has been argued to be important but difficult in data parallel programming [NO94, KC98].

In summary, our diffusion theorem provides a significant guide for both parallel programming by hand and automatic parallelization by machine. If a recursive definition is in the required form to which our diffusion theorem can be applied, then it can be automatically parallelized into a set of efficient parallel primitives.

The organization of this paper is as follows. In Section 2, we review the notational conventions and some basic concepts used in this paper, and explain the existing problem of parallel programming in Bird Meertens Formalisms. We propose our idea of diffusion transformation in Section 3, and generalize the idea to be applicable to functions on data structures other than lists in Section 4. Related work and discussions are given in Section 5.

2 BMF and Parallel Computation

In this section, we briefly review the notational conventions and some basic concepts in Bird Meertens Formalisms (BMF for short) [Bir87, Ski94], and point out some related results which will be used in the rest of this paper.

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by id. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary functions by

$$(a\oplus)\,b = a \oplus b = (\oplus\,b)\,a.$$

Besides, function zip which will be used later is informally defined by:

$$zip\ [x_1, x_2, \ldots, x_n]\ [y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)].$$

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (and $[\cdot]$ for the function taking a to $[a]$), and $x +\!\!+ y$ for the concatenation of two lists x and y. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] +\!\!+ xs$.

2.1 BMF: An Architecture Independent Parallel Model

It has been argued in [Ski90] that BMF [Bir87] is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as parallel primitives suitable for parallel implementation. Three important higher order functions are *map*, *reduce* and *scan*.

Map is the operator which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

$$k * [x_1, x_2, \ldots, x_n] = [k\,x_1, k\,x_2, \ldots, k\,x_n].$$

Reduce is the operator which collapses a list into a single value by repeated application of some associative binary operator. It is written as an infix $/$. Informally, for an associative binary operator $\oplus$, we have

$$\oplus/\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Scan is the operator that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ with an unit $\iota_\oplus$, we have

$$\begin{aligned}&\oplus\!\!/\!\!/\,[x_1, x_2, \ldots, x_n]\\ &\quad = [\iota_\oplus, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n].\end{aligned}$$

It has been shown that map, reduce and scan have nice massively parallel implementations on many architectures [Ski90, Ble89]. If k and an associative $\oplus$ use $O(1)$ parallel time, then $k*$ can be implemented using $O(1)$ parallel time, and both $\oplus/$ and $\oplus\!\!/\!\!/$ can be implemented using $O(\log N)$ parallel time (N denotes the size of the list). For example, $\oplus/$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $\oplus\!\!/\!\!/$ can be found in [Ble89], though it is not so obvious.

A similar model to BMF that sounds more practical to use is the parallel functional language called Nesl [Ble92] that includes two principle parallel primitives, namely *apply-to-each* (like map) and *scan*.

2.2 Simple Diffusion: the Homomorphism Lemma

List homomorphisms (or *homomorphisms* for short) [Bir87] are those *recursive* functions on finite lists that *promote* through list concatenation. A function h satisfying the following equations is called a *list homomorphism*:

$$\begin{array}{lcl} h\ [a] & = & k\,a \\ h\ (x +\!\!+\ y) & = & h\,x \oplus h\,y \end{array}$$

where $\oplus$ is an *associative* binary operator. We write $([k, \oplus])$ for the unique function h. For example, the function sum, for summing up all elements in a list, can be defined as a homomorphism of $([id, +])$.

The relevance of homomorphisms to parallel programming is basically from the *homomorphism lemma* [Bir87]:

$$([k, \oplus]) = (\oplus/) \circ (k*)$$

saying that every list homomorphism can be *diffused* to be the composition of a reduce and a map. It follows that if we can derive list homomorphisms, then we can get corresponding parallel programs. Though being so simple, the homomorphism lemma plays an important role to bridge the gap between programs in recursive form and programs in compositional form, and it has led to surprisingly many good results [Gor96a, Gor96b, HIT97, HTC98]. The major reason is that list homomorphisms provide us a new interface to develop parallel programs.

The importance of using a recursion instead of map and reduce in parallel programming has greatly motivated us to study this simple diffusion in a more general and practical manner.

2.3 Limitation of the Simple Diffusion

The homomorphism lemma, a simple diffusion transformation, is nice, but a closer look at the lemma reveals a practical limitation in the case where the result of the application of $([k, \oplus])$ to a list returns a function instead of a basic value. Take a look at the following homomorphism h for *psums* (computing the prefix sums of a list) derived in [HTC98].

$$\begin{array}{lcl} psums & : & [Int] \to [Int] \\ psums\ x & = & s\ \textbf{where}\ (s, g) = h\ x\ 0 \\ h\ [a]\ c & = & ([c + a], a) \\ h\ (x +\!\!+\ y)\ c & = & \textbf{let}\ (s_x, g_x) = h\ x\ c \\ & & \quad\ (s_y, g_y) = h\ y\ (c + g_x) \\ & & \textbf{in}\ (s_x +\!\!+\ s_y,\ g_x + g_{yx}) \end{array}$$

Function h can be described in a more explicit homomorphic way by

$$\begin{array}{lcl} h & : & [Int] \to (Int \to [Int]) \\ h & = & ([k, \oplus]) \\ & & \textbf{where} \\ & & \quad k\ a = \lambda c.\,([c + a], a) \\ & & \quad h_x \oplus h_y = \lambda c.\,\textbf{let}\ (s_x, \underline{g_x}) = h_x\ c \\ & & \qquad\qquad\qquad\quad (s_y, g_y) = h_y\ \underline{(c + g_x)} \\ & & \qquad\qquad\quad \textbf{in}\ (s_x +\!\!+\ s_y,\ g_x + g_y) \end{array}$$

The problem lies in the definition for $\oplus$ where h_x and h_y are functions. As indicated by the underlined parts above, the computation of h_y cannot be performed until it receives g_x, one of the results from the computation of h_x. Thus, a naive implementation of such $\oplus$ may lead to a big function-closure, resulting in a sequential program. So we need to study carefully this dependency reflected in the use of accumulating parameter (like c used in h) in the initial definition (for h).

Another practical concern [Gor96a, CTT97, HTC98] is that defining a function, say h, over join lists (as we do with list homomorphism) like

$$\begin{array}{lcl} h\ [a]\ c & = & \ldots \\ h\ (x +\!\!+\ y)\ c & = & \ldots \end{array}$$

is much more difficult than doing over cons lists like

$$\begin{array}{lcl} h\ [\,]\ c & = & \ldots \\ h\ (a : x)\ c & = & \ldots \end{array}$$

Fortunately, with the technique of parallelization transformation [HTC98], we are able to allow functions to be defined in the latter easier form which will be handled later by our proposing diffusion.

3 Diffusion

Diffusion is a transformation turning a recursive definition into a composition of our higher order functions, namely *map*, *reduce* and *scan*. To be useful, our proposing diffusion transformation should satisfy the following two requirements.

- First, the diffusion transformation should be *powerful* enough to be applied to a wide class of recursive forms of interest.
- Second, the result parallel programs should be *efficient*, in the sense that if the original program uses t sequential time, then the derived parallel one should take at most $O(\log N)$ times of the sequential time. Here and after, we often use N to denote the size of the input list.

3.1 Diffusion Theorem

To meet our requirements, we should carefully choose proper recursive forms to diffuse We will be generous with paper space to show how we reach our target form in this section.

The simplest form: diffused to `reduce`

We start by considering the following most natural and simplest recursive form defined over the cons lists:

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ (a:x) & = & a\ \oplus\ h\ x. \end{array}$$

Here, by choosing suitable e and $\oplus$, we are able to define many functions of our interest. This kind of definition should be familiar enough to the functional programming community, which is known to be *catamorphism* [MFP91] or *foldr* in Haskell. To diffuse this recursive form into our parallel primitives, we require $\oplus$ to be associative, and thus have

$$h\ x = (\oplus/x)\ \oplus\ e.$$

The correctness of this simple diffusion is obvious, and the efficiency requirement is satisfied: the sequential program uses $O(N \times t_\oplus)$ sequential time while the derived version use $O(\log N \times t_\oplus)$ parallel time, where $t_\oplus$ denotes the time to compute $\oplus$.

The form with computation on elements: diffused to `reduce` and `map`

We, however, should not be satisfied with the power of the simplest form. Even for the following simple function for squaring each element of a list

$$\begin{array}{lcl} sqrs\ [\,] & = & [\,] \\ sqrs\ (a:x) & = & sqr\ a\ :\ sqrs\ x \end{array}$$

we cannot diffuse it, because the $\oplus$ here is defined by

$$a \oplus b = sqr\ a : b$$

which is not associative since a and b have the different types. Nevertheless, with the normalization algorithm given in [HTC98], we can turn the second equation of $sqrs$ to

$$sqrs\ (a:x) = \underline{[sqr\ a]}\ +\!\!+\ \underline{sqrs\ x}.$$

Here $+\!\!+$ is an associative operator used to combine a *computation* on a with the recursive part, as indicated by the underlined parts. In fact we can and should allow such computation on a. This leads to the following improved recursive form, permitting a computation on a by any function k.

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ (a:x) & = & k\ a\ \oplus\ h\ x \end{array}$$

This recursion can be diffused into the following if $\oplus$ is associative.

$$h\ x = ((\oplus/) \circ k*)\ x\ \oplus\ e$$

If e is the unit of $\oplus$, this can be reduced to

$$h = (\oplus/) \circ k*$$

which is the same as the homomorphism lemma except that h is defined as a recursion on cons lists rather than that on join lists.

The form with accumulating parameters: diffused to `reduce`, `map` and `scan`

After discussing the two recursive forms that can be diffused, we are ready to solve the problem in Section 2.3, in which h is a function whose application to a list still gives a function. To simplify our presentation, we shall focus ourselves on those the functions that have an accumulating parameter as in the following recursive form.

$$\begin{array}{lcl} h & : & [I] \to A \to O \\ h\ [\,]\ c & = & g_1\ c \\ h\ (a:x)\ c & = & k(a,c)\ \oplus\ h\ x\ (c \otimes g_2\ a) \end{array}$$

Note that h has two parameters; the first is the inductive one and the second is the accumulating one. This recursive form extends the above second one with an accumulating parameter. To be specific, the computation on element a can include the use of the accumulating parameter c, and the accumulating parameter in the recursive call can be updated with a combination of some computation on a using a binary, associative operator $\otimes$.

We need to deal with such additional accumulating parameter c. Our idea is to remove it from the recursion by precomputation. We make the accumulating parameter look like a constant by precomputing all the c, say cs, that will be used during computation of h. The trick, and also an important point here, is our use of scan, an efficient parallel primitive, to perform such precomputation in a parallel way:

$$cs = (c\otimes) * (\otimes\!\!\!/\ (g_2 * x)).$$

Now using the diffusion for the form that does not contain accumulating parameters while paying attention to the access to corresponding elements of cs, we can come up with our diffusion theorem.

Theorem 1 (Diffusion) Given a function h defined in the above recursive form, if $\oplus$ and $\otimes$ are associative and have units, then h can be diffused into the following.

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\!/(g_2 * x)) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \end{array}$$

Proof: We can prove that the newly defined h is equivalent to the original, by induction on the inductive parameter x, as shown by the following calculation.

- Base case $x = [\,]$.

$$\begin{array}{lcl} & & h\ [\,]\ c \\ & = & \quad \{\ \text{by the new definition}\ \} \\ & & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\!/(g_2 * [\,])) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \\ & = & \quad \{\ \text{map}\ \} \\ & & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\!/[\,]) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ scan } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * [\iota_\otimes] \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ map, } c \otimes \iota_\otimes = c \ \} \\
& \textbf{let } cs' +\!\!+ [c'] = [c] \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ by pattern matching: } cs' = [\,],\ c = c' \ \} \\
& \textbf{let } e = g_1\ c \\
& \quad ac = zip\ x\ [\,] \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ zip, let } \} \\
& (\oplus/\,(k * [\,])) \oplus (g_1\ c) \\
= & \{ \text{ map, } \oplus \ \} \\
& \iota_\oplus \oplus (g_1\ c) \\
= & \{ \ \iota_\oplus \text{ is a unit of } \oplus \ \} \\
& g_1\ c
\end{array}
$$

- Inductive case $x = a : x'$.

$$
\begin{array}{ll}
& h\ (a : x')\ c \\
= & \{ \text{ by the new definition } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!/(g_2 * (a : x'))) \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ map } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!/(g_2\ a : g_2 * x')) \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ definition of scan [Bir87] } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\iota_\otimes : (g_2\ a \otimes) * \\
& \qquad\qquad \otimes\!\!/(g_2 * x')) \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \text{ map, associativity of } \otimes \ \} \\
& \textbf{let } cs' +\!\!+ [c'] = c : ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \\
& \quad ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \{ \ ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \text{ is not empty } \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \\
& \quad ac = zip\ (a : x')\ (c : cs'') \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c'') \\
= & \{ \text{ zip } \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \\
& \quad ac'' = zip\ x'\ cs'' \\
& \quad ac = (a, c) : ac'' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c'') \\
= & \{ \text{ map, } \oplus \ \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \\
& \quad ac'' = zip\ x'\ cs'' \\
& \textbf{in } k(a, c) \oplus (\oplus/\,(k * ac'')) \oplus (g_1\ c'') \\
= & \{ \text{ let, associativity of } \oplus \ \} \\
& k(a, c) \oplus \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!\!/(g_2 * x')) \\
& \quad ac'' = zip\ x'\ cs'' \\
& \textbf{in } (\oplus/\,(k * ac'')) \oplus (g_1\ c'') \\
= & \{ \text{ inductive hypothesis } \} \\
& k(a, c) \oplus h\ x'\ (c \oplus g_2\ a)
\end{array}
$$

□

Note that we use the matching notation of $cs' +\!\!+ [c']$ to extract the leading part and the last element from a list. It is not difficult to check that this diffusion indeed meets our requirements as given at the beginning of this section.

To see a simple use of this theorem, consider the following function *sbp* to solve a simplified bracket matching problem: determining whether a single type (not many types) of brackets '(' and ')' are matched in a given string. It uses a counter (starting with zero) to increase upon meeting '(' and to decrease upon meeting ')':

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & c == 0 \\
sbp\ (a : x)\ c & = & \textbf{if } a == \text{'('} \textbf{ then } sbp\ x\ (c + 1) \\
& & \textbf{else if } a == \text{')'} \textbf{ then} \\
& & \quad c > 0\ \wedge\ sbp\ x\ (c - 1) \\
& & \textbf{else } sbp\ x\ c.
\end{array}
$$

Merging all recursive calls by the normalization algorithm in [CDG96] will give

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & g_1\ c \\
sbp\ (a : x)\ c & = & k(a, c)\ \wedge\ sbp\ x\ (c + g_2\ a)
\end{array}
$$

where

$$
\begin{array}{lcl}
g_1\ c & = & c == 0 \\
k(a, c) & = & \textbf{if } a == \text{'('} \textbf{ then } True \\
& & \textbf{else if } a == \text{')'} \textbf{ then } c > 0 \textbf{ else } True \\
g_2\ a & = & \textbf{if } a == \text{'('} \textbf{ then } 1 \\
& & \textbf{else if } a == \text{')'} \textbf{ then } (-1) \textbf{ else } 0
\end{array}
$$

which is in the right form that the diffusion theorem can be applied to get an efficient parallel program for *sbp*. It is worth noting that this problem was considered as a kind of hard parallelization problem in [Col95]. By using the diffusion theorem, its efficient parallel program can be obtained by a straightforward program calculation.

3.2 Diffusion Algorithm

Although we have argued that our recursive form is powerful and general, user's programs may not be exactly in our form. Therefore, we turn to find a way to transform more general programs into the form that our diffusion theorem can be applied. The diffusion algorithm is for this purpose.

We shall illustrate our algorithm by the derivation of an explicit parallel algorithm for bracket matching in terms of our parallel primitives from the naive program given in the introduction.

Step 1: Linearizing Recursive Calls

It is required by the diffusion theorem that the occurrences of the recursive call should appear once. If there are many occurrences, we need to merge them into a single one. Recall the definition of *bm* in the introduction. In the branch of $(a : x)$, there are three occurrences of the recursive call to *bm* in the right hand side. We can merge them based on the normalization algorithm [CDG96].

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a : x)\ s & = & g_1\ (a, s)\ \wedge\ bm\ x\ (g_2\ a\ s)
\end{array}
$$

where

$$
\begin{array}{lcl}
g_1\ (a, s) & = & \textbf{if } isOpen\ a \textbf{ then } True \\
& & \textbf{else if } isClose\ a \textbf{ then} \\
& & \quad noEmpty\ s\ \wedge \\
& & \quad match\ a\ (top\ s) \\
& & \textbf{else } True \\
g_2\ a\ s & = & \textbf{if } isOpen\ a \textbf{ then } push\ a\ s \\
& & \textbf{else if } isClose\ a \textbf{ then } pop\ s \textbf{ else } s.
\end{array}
$$

Step 2: Identifying Associative Operators

Central to our diffusion theorem is the use of associativity of the binary operators $\oplus$ and $\otimes$. Clearly, $\oplus$ should be an associative operator over the resulting domain of function h, while $\otimes$ is an associative operator over the resulting domain of the accumulating parameter (e.g., s for bm).

For bm, it is easy to find that $\oplus$ is $\wedge$, but not so easy to find what corresponds to $\otimes$ which is supposed to combine two stacks, satisfying

$$g_2\ a\ s = s \otimes g_2'\ a.$$

Consider the following stack we would like to use in bm:

$$Stack\ \alpha\ =\ Empty \mid Push\ \alpha\ Stack \mid Pop\ Stack$$

From this definition, we are able to systematically derive the following associative operator $\otimes$ for combining two stacks as shown in [SF93, HT99].

$$\begin{array}{lcl} s \otimes Empty & = & s \\ s \otimes (Push\ a\ s') & = & Push\ a\ (s \otimes s') \\ s \otimes (Pop\ s') & = & Pop\ (s \otimes s') \end{array}$$

Using this $\otimes$, we thus have

$$\begin{array}{lcl} g_2\ a\ s & = & s \otimes\ g_2'\ a \\ g_2'\ a & = & \textbf{if}\ isOpen\ a\ \textbf{then}\ Push\ a\ Empty \\ & & \textbf{else if}\ isClose\ a\ \textbf{then} \\ & & \quad Pop\ Empty\ \textbf{else}\ Empty. \end{array}$$

Step 3: Applying the Diffusion Theorem

After merging recursive call occurrences and identifying associative operators, we are ready to apply the diffusion theorem.

For bm, it follows from the diffusion theorem that

$$\begin{array}{lcl} bm\ x\ c & = & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!/(g_2 * x)) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\wedge/\ (g_1 * ac)) \wedge (isEmpty\ c') \end{array}$$

Step 4: Optimizing Operators

So far we have derived a parallel program that is described in terms of our parallel primitives. According to our cost model for parallel primitives, we should continue to find efficient implementation for the operations like g_1, g_2, k, $\oplus$ and $\otimes$ that are used in each parallel primitive to obtain a more efficient parallel program.

For bm, we need to show that $\otimes$ as well as the stack operations can be implemented in $O(1)$ parallel time if we want an $O(\log N)$ parallel program for bracket matching. Notice that with the property of $Pop\ (Push\ a\ s) = s$, our stack should, as discussed in [HT99], keep the form of

$$Push\ a_1\ (\ldots(Push\ a_n\ (\underbrace{Pop\ (\ldots(Pop}_{m}\ Empty)))))),$$

that can be naturally represented by

$$([a_1, \cdots, a_n], n, m)$$

Here, m denotes the number of Pop occurrences, and the second component n is added to incrementally compute the length of the first component. With this new representation, we can refine all operations on stack to those using $O(1)$ parallel time as follows.

$$\begin{array}{lcl} Empty & = & ([\,], 0, 0) \\ isEmpty\ ([\,], 0, 0) & = & True \\ isEmpty\ \_ & = & False \\ Push\ c\ (cs, n, m) & = & ([c] +\!\!+ cs, n+1, m) \\ Pop\ (c : cs, n+1, m) & = & (cs, n, m) \\ Pop\ ([\,], 0, m) & = & ([\,], 0, m+1) \end{array}$$

And

$$\begin{array}{l} (cs_1, n_1, m_1) \otimes (cs_2, n_2, m_2) \\ \quad = \textbf{if}\ m_1 \geq n_2\ \textbf{then}\ (cs_1, n_1, m_1 - n_2 + m_2) \\ \qquad \textbf{else}\ (cs_1 +\!\!+ drop\ m_1\ cs_2, n_1 + n_2 - m_1, m_2) \end{array}$$

Function $drop\ n\ x$ is to drop the first n elements from list x. According to the fact that $+\!\!+$ and $drop$ can be implemented using constant parallel time (e.g., under the PRAM parallel model [Ski94]), our final operators of $\otimes$, g_1, and g_2 can be implemented using $O(1)$ parallel time, and we thus got an $O(\log N)$ parallel program for bracket matching.

It has been shown that the bracket matching problem can be solved in $O(\log N)$ parallel time [GR88] where N denotes the length of the input string, but the algorithm involved is rather complicated and its correctness is difficult to prove. To resolve this problem, Cole [Col95] proposed an *informal* development of an *suboptimal* $O(\log^2 n)$ parallel algorithm. In contrast, we propose a formal development of a novel parallel one to solve this problem. In [HT99], we proposed an homomorphic algorithm for the same problem but left as an open work for the derivation of an explicit Nesl parallel program in terms of parallel primitives.

4 Polytypic Diffusion

In this section, we highlight how to generalize the idea of diffusion of recursive definitions on lists to those on other data structures like trees. Rather than giving a formal study of this generalization, we shall concentrate ourselves on trees, and explain our idea in a concrete manner, It should not be difficult at all to generalize from trees to other data structures.

4.1 Tree Parallel Primitives

We consider binary trees defined by

$$Tree\ \alpha = Leaf\ \alpha \mid Node\ \alpha\ (Tree\ \alpha)\ (Tree\ \alpha).$$

Based on the constructive algorithmics [MFP91, Fok92a], we can define a set of tree parallel primitives by a natural generalization of those primitives on lists.

Map

Map is to apply two functions on a tree; one to all leaves and the other to all internal nodes.

$$\begin{array}{lcl} map\ f_1\ f_2\ (Leaf\ a) & = & Leaf\ (f_1\ a) \\ map\ f_1\ f_2\ (Node\ a\ l\ r) & = & Node\ (f_2\ a) \\ & & \quad (map\ f_1\ f_2\ l) \\ & & \quad (map\ f_1\ f_2\ r) \end{array}$$

The parallelism in map should be obvious. For example, using enough processors we can easily implement it in $O(max(T(f_1),T(f_2)))$ parallel time, where $T(f_1)$ and $T(f_2)$ denote the time for computing f_1 and f_2 respectively.

Scan

Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96]. We have two kinds of scan: scanning a tree upwards or downwards. They will be called *upward scan*, denoted by $scan_u$, and *downward scan*, denoted by $scan_d$, respectively.

Upward scan computes sum of all elements with a binary operator $\oplus$, while keeping all running sums during upwards computation. Like list scans requiring a binary operator that are associative, our tree scans relies on a binary operator that are both associative and commutative, which is sufficient (not necessary) to guarantee their efficient parallel implementation.

Given an associative and commutative operator $\oplus : A \to A \to A$, $scan_u$ is defined by

$$\begin{array}{lcl} scan_u\ (\oplus)\ (Leaf\ a) & = & Leaf\ a \\ scan_u\ (\oplus)\ (Node\ a\ l\ r) & = & \textbf{let}\ l' = scan_u\ (\oplus)\ l \\ & & \quad\ r' = scan_u\ (\oplus)\ r \\ & & \textbf{in}\ Node \\ & & \quad (a \oplus root\ l' \oplus root\ r') \\ & & \quad l'\ r' \end{array}$$

where

$$\begin{array}{lcl} root\ (Leaf\ a) & = & a \\ root\ (Node\ a\ l\ r) & = & a. \end{array}$$

Downward scan $scan_d$ is to propagate information from the root to the leaves with some computation at the internal nodes by an associative $\oplus$.

$$\begin{array}{l} scan_d\ (\oplus)\ g_1\ g_2\ (Leaf\ a)\ c \\ \quad = Leaf\ c \\ scan_d\ (\oplus)\ g_1\ g_2\ (Node\ a\ l\ r)\ c \\ \quad = Node\ c\ (scan_d\ (\oplus)\ g_1\ g_2\ l\ (c \oplus g_1\ a))) \\ \qquad\qquad (scan_d\ (\oplus)\ g_1\ g_2\ r\ (c \oplus g_2\ a)) \end{array}$$

Efficient implementation of the scans is not so obvious. Fortunately, so many studies have been devoted to show that the tree contraction technique [LF80, TV84, MR85, GMT87, ADKP87, Ble89] can be applied to implement our scans efficiently, and some more concrete studies can be found [GCS94, Gib96, Ski96]. We do not recapitulate them, rather we summarize the result. For the upward scan, the parallel time is $O(T(\oplus) \times \log N)$ with $N/\log N$ processors, where N denotes the number of tree nodes (no matter how unbalanced the tree is). For the downward scan, the parallel time is $O(T(\oplus) \times \log N + max(T(g_1), T(g_2)))$.

Reduce

Generalizing reduce from that on lists is straightforward. Given an associative and commutative operator $\oplus : A \to A \to A$, *reduce* is defined by

$$\begin{array}{l} reduce\ (\oplus)\ (Leaf\ a) = a \\ reduce\ (\oplus)\ (Node\ a\ l\ r) \\ \quad = a\ \oplus\ reduce\,(\oplus)\,l\ \oplus\ reduce\,(\oplus)\,r \end{array}$$

The reduce can be implemented in parallel by using the tree contraction technique similarly to the upward scan.

Zip

Zip merges two trees of the same form into one by pairwisely gluing elements.

$$\begin{array}{l} zip\ (Leaf\ a_1)\ (Leaf\ a_2) = Leaf\ (a_1, a_2) \\ zip\ (Node\ a_1\ l_1\ r_1)\ (Node\ a_2\ l_2\ r_2) \\ \quad = Node\ (a_1, a_2)\ (zip\ l_1\ l_2)\ (zip\ r_1\ r_2) \end{array}$$

This definition can be extended from two data to any number of data. The parallelism in zip is also obvious.

4.2 Tree Diffusion Theorem

We now generalize the diffusion theorem from list functions to tree functions.

Theorem 2 (Tree Diffusion) Let $h : Tree\ \alpha \to A \to O$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a)\ c & = & k_1(a, c) \\ h\ (Node\ a\ l\ r)\ c & = & k_2(a, c)\ \oplus \\ & & h\ l\ (c \otimes g_1\ a)\ \oplus \\ & & h\ r\ (c \otimes g_2\ a) \end{array}$$

where $\oplus : O \to O \to O$ is an associative and commutative operator, $\otimes : A \to A \to A$ is an associative operator, and k_1, k_2, g_1 and g_2 are given functions. Then, h can be equivalently defined by

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \quad\ ac = zip\ x\ cs \\ & & \textbf{in}\ reduce\ (\oplus)\ (map\ k_1\ k_2\ ac). \end{array}$$

Proof Sketch: This can be proved by induction on the structure of x, quite similar to what we did for the the proof of Theorem 1. □

It is worth noting that the tree diffusion theorem is quite similar to the list diffusion theorem but in a more compact form due to our generalized definition of map and scan.

The tree diffusion theorem can be degenerated to the following corollary where h does not use any accumulating parameter.

Corollary 3 Let $h : Tree\ \alpha \to O$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a) & = & k_1\ a \\ h\ (Node\ a\ l\ r) & = & k_2\ a\ \oplus\ h\,l\ \oplus\ h\,r \end{array}$$

where $\oplus : O \to O \to O$ is an associative and commutative operator, k_1 and k_2 are given functions. Then,

$$h\ x\ =\ reduce\ (\oplus)\ (map\ k_1\ k_2\ x)$$ □

This corollary is similar to the homomorphism lemma in Section 2.2. Another corollary, focusing on treating manipulation of the accumulating parameter, is obtained by eliminating the last reduce step in the new definition of h in the tree diffusion theorem.

Corollary 4 Let $h : Tree\ \alpha \to A \to Tree\ \beta$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a)\ c & = & Leaf\ (k_1(a, c)) \\ h\ (Node\ a\ l\ r)\ c & = & Node\ (k_2(a, c)) \\ & & \quad (h\ l\ (c \otimes g_1\ a)) \\ & & \quad (h\ r\ (c \otimes g_2\ a)) \end{array}$$

where $\otimes : A \to A \to A$ is associative, and k_1, k_2, g_1 and g_2 are given functions. Then, h can be equivalently defined by

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \textbf{in}\ map\ k_1\ k_2\ (zip\ x\ cs). \end{array}$$

□

To see how tree diffusion theorem works, consider the following naive solution to the problem to number each node and leaf of a tree in an infix traversing order:

$$\begin{array}{lcl} nt\ (Leaf\ a)\ c & = & Leaf\ c \\ nt\ (Node\ a\ l\ r)\ c & = & Node\ (c + size\ l) \\ & & \quad (nt\ l\ c) \\ & & \quad (nt\ r\ (c + size\ l + 1)). \end{array}$$

Here $size$, computing the number of nodes of a tree, is defined by

$$\begin{array}{lcl} size\ (Leaf\ a) & = & 1 \\ size\ (Node\ a\ l\ r) & = & 1 + size\ l + size\ r \end{array}$$

or simply by

$$size\ t = reduce\ (+)\ (map\ (\lambda x.1)\ (\lambda x.1)\ t).$$

We number a tree by using a counter c (starting from an initial value). It is actually not easy to derive an $O(\log N)$ (N denotes the number of tree nodes) parallel program, because of two seemingly sequential factors in the above naive specification, the counter c and a probably very unbalanced tree, which may sequentialize the visit of each node.

We cannot directly apply the tree diffusion theorem to parallelize nt, because nt uses an auxiliary function $size$. Fortunely, this can be easily handled by sort of memoisation; for the given tree t, we derive the following $scansize$ from $size$, which can memoise the size of the tree rooted at each tree node

$$scansize\ t = scan_u\ (+)\ (map\ (\lambda x.1)\ (\lambda x.1)\ t).$$

Suppose that we have a parallel operation $getchildren$ that can replace each node in parallel with a pair of the root values of its two children, then we can associate the children's sizes to each node of the original tree by

$$tup\ t = zip\ (getchildren\ (scansize\ t))\ t.$$

We then define

$$nt\ t = nt'\ (tup\ t)$$

and a simple calculation can yield the following definition for nt'.

$$\begin{array}{lcl} nt'\ (Leaf\ ((s_l, s_r), a)\ c & = & Leaf\ c \\ nt'\ (Node\ ((s_l, s_r), a)\ l\ r)\ c & = & Node\ (c + s_l) \\ & & \quad (nt\ l\ c) \\ & & \quad (nt\ r\ (c + (s_l + 1))). \end{array}$$

Now we can apply Corollary 4 to obtain the following explicit parallel code for nt'.

$$\begin{array}{lcl} nt'\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \textbf{in}\ map\ k_1\ k_2\ (zip\ x\ cs) \end{array}$$

where

$$\begin{array}{lcl} k_1\ (((s_l.s_r), a), c) & = & c \\ k_2\ (((s_l.s_r), a), c) & = & c + s_l \\ g_1\ ((s_l.s_r), a) & = & 0 \\ g_2\ ((s_l.s_r), a) & = & s_l + 1. \end{array}$$

5 Related Work and Discussions

Besides the related work as in the introduction, our work is closely related to three kinds of active research work, namely parallel programming in BMF, parallel programming with scans, and polytypic programming.

Parallel Programming in BMF has been attracting many researchers. The initial BMF [Bir87] was designed as a calculus for deriving (sequential) efficient programs on lists. Skillicorn [Ski90] showed that BMF could also provide an architecture-independent parallel model for parallel programming because a small fixed set of higher order functions in BMF such as map, reduce can be mapped efficiently to a wide range of parallel architectures. Along with the extension of BMF from the theory of lists to the uniform theory of most data types, Skillicorn [Ski93b, Ski94, Ski96] coincided these data types as *categorical data types*, and established an architecture-independent cost model for generic catamorphisms. This influence our definitions of parallel primitives over data structures like trees.

Despite the architecture-independent cost model for the extended BMF, we are lacking of powerful parallelization theorem and laws for calculating efficient parallel programs, which more or less prevents it from being widely used. To remedy this situation, Quite a lot of recent studies have been devoted to the development of powerful parallelization methods with BMF [Ski93a, Col95, Gor96b, Gor96a, GDH96, HIT97, HTC98]. As explained in Section 2, the main idea is based on derivation of list homomorphism from a naive specification. This is based on the fact that a list homomorphism can be efficiently implemented by a composition of two parallel primitives, namely reduce and map. Our uniform recursions for structuring the parallel primitives as in the diffusion theorem is more general and easier to be used in programming than list homomorphisms. And our diffusion theorem can be considered as an extension of the homomorphism lemma. Our explicit use of accumulating parameters in recursive definitions and our use of scan for memoization in a parallel way are quite new.

Parallel programming with scans (either on lists or trees) is not new. For example, scan on lists is argued to be an important parallel skeleton [Ble89], and is used as one of the two important parallel constructs in Nesl [Ble92]. However, if we look at those programs in Nesl, they only contain some very simple use of scan (with simple operations like +). It lacks systematic way to develop parallel programs with scans. It might be difficult, even for an Nesl expert, to write an efficient program to solve our running example of bracket matching, because scans with complicated operations needs to be carefully designed.

Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96], but to ensure the existence of efficient parallel implementation the complicated "cooperation condition" must be checked. In contrast, we give a more natural definition using an explicit accumulating parameter, and simplify the condition to guarantee the existence of efficient parallel implementation.

Polytypic programming [JJ96, JJ97] are widely used in the Squigol community [Mal89, Fok92b, MFP91], but its importance in parallel programming has not been well recognized. Starting with [BdM96], more and more algorithmic problems have been considered in a polytypic setting [dM95, Jeu95, Mee96, JJ96]. In this paper, we made an at-

tempt to apply polytypic idea to the development of parallel algorithms.

This work is a continuation of our effort to apply the so-called program calculation technique [THT98] to the development of efficient parallel programs [HIT97, HTC98]. As a matter of fact, our diffusion theorem is much related to our previous parallelization theorem [HTC98]; the parallelization theorem only derives a homomorphic program whereas diffusion theorem gives an explicit parallel program using parallel primitives. Nevertheless, the algorithm to turn a general program to the form that the parallelization theorem can be applied [HTC98] has been borrowed here as shown in Section 3.2.

We are working on a precise definition of the class of recursive definitions that can be parallelized into parallel primitives, and on formalization of the diffusion algorithm in a more mechanical way. Although we have not verify our idea in a practical system yet, we believe that it is promising to be used practically. We are going to embed our idea in, for example, the Nesl-like system in the future.

Another interesting future work is to deal with diffusion of polymorphic recursive definitions into a set of communication primitives, which should be very important to manipulate data communications and distributions.

Acknowledgement

This paper owes much to the thoughtful and helpful discussions with Wei Ngan Chin during his visit of Tokyo University on efficient implementation of list homomorphisms using scan. Thanks are also to Manuel M. T. Chakravarty who suggested us to try deriving parallel primitives from sequential programs rather than from list homomorphisms, and to Akihiko Takano and other Tokyo CACA members for their useful comments.

References

[ADKP87] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proceedings of the Twenty-Fifth Allerton Conference on Communication, Control and Computing*, pages 624–633, September 1987.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Ble89] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[BW88] R. Bird and P. Waddler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CDG96] W. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 579–586, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[Col95] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[CTT97] W. Chin, S. Tan, and Y. Teo. Deriving efficient parallel programs for complex recurrences. In *ACM SIGSAM/SIGNUM International Conference on Parallel Symbolic Computation*, pages 101–110, Hawaii, July 1997. ACM Press.

[dM95] O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1995.

[Fok92a] M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, The Netherlands, September 1992.

[Fok92b] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[GCS94] J. Gibbons, W. Cai, and D. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, (23):1–18, August 1994.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[Gib92] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction* (LNCS 669), pages 122–138. Springer-Verlag, 1992.

[Gib96] J. Gibbons. Computing downwards accumulations on trees quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

[GMT87] H. Gazit, G.L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156. Plenum Press, 1987.

[Gor96a] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Gor96b] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conference on Programming Languages: Implementation, Logics and Programs, LNCS 1140*, pages 274–288. Springer-Verlag, 1996.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[HIT97] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HL93] P. Hammarlund and B. Lisper. Data parallel programming, a survey and a proposal for a new model. Technical Report 93/8-SE, Department of Teleinformatics, Royal Institute of Technology, September 1993.

[HS86] W.D. Hills and Jr. G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HT99] Z. Hu and M. Takeichi. Calculating an optimal homomorphic algorithm for bracket matching. *Parallel Processing Letters*, 9(1), 1999.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, La Jolla, California, June 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *2nd International Summer School on Advanced Functional Programming Techniques, LNCS*. Springer Verlag, July 1996.

[JJ97] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 470–482. ACM Press, January 1997.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[KC98] G. Keller and M. T. Chakravarty. Flatten trees. In *EuroPar'98, LNCS*. Springer-Verlag, September 1998.

[LF80] R.E. Ladner and M.J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[Mal89] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989.

[Mee96] L. Meertens. Calculate polytypically. In *Proc. Conference on PLILP, LNCS 1140*, pages 1–16. Springer Verlag, 1996.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[MR85] G.L. Miller and J. Reif. Parallel tree contraction and its application. In *26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[NO94] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (preliminary report). In *International Workshop on Theory and Practice on Parallel Programming*. LNCS 907, 1994.

[Pra92] T.W. Pratt. Kernel-control parallel versus data parallel: A technical comparison. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, appeared as SIGPLAN Notices, Vol 28, No. 1, January 1993*, pages 5–8, September 1992.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[Ski93a] D.B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[Ski93b] D.B. Skillicorn. Categorical data types. In *Second Workshop on Abstract Models for Parallel Computation*, Oxford University Press, 1993.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[Ski96] D.B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributied Computing*, 39(0160):115–125, 1996.

[THT98] A. Takano, Z. Hu, and M. Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 1998. to appear.

[TV84] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *25th IEEE Symposium on Foundations of Computer Science*, pages 12–22, 1984.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

Shifting Expression Procedures into Reverse*

Mark Tullsen Paul Hudak

Department of Computer Science
Yale University
New Haven CT 06520-8285
mark.tullsen@yale.edu, paul.hudak@yale.edu

Abstract

The best known approach to program transformation is the unfold/fold methodology of Burstall and Darlington: a simple, intuitive, and expressive approach which serves as the basis of many automatic program transformation algorithms (such as partial evaluation and deforestation). Unfortunately unfold/fold does not preserve total correctness and requires maintaining a transformation history of the program. Scherlis invented a similar approach, expression procedures, which solved these two problems: expression procedures preserve total correctness and require no transformation history.

Motivated by our desire to make expression procedures more expressive by eliminating their one-directional nature (they are designed to specialize but not to generalize functions), we have developed an equational specification of expression procedures, in which the essence of expression procedures is expressed in a single transformation rule. Our approach has the following advantages over expression procedures: (1) all program derivations are reversible; (2) transformations can be done which expression procedures cannot do; (3) fewer and simpler rules are used; and (4) the proof of correctness is simpler.

1 Introduction

1.1 Motivation

Program transformation has been an elusive goal of the programming language research community. We talk about it, write about it, preach about it, but in practice don't use it very much. In the functional programming community the situation is especially unfortunate, where we have the (presumably) simplest framework within which to do program transformation, and at the same time have the greatest need for it: functional programs typically run slower than imperative programs with similar functionality. So what is the problem? Why hasn't program transformation been used more in practice? We see several reasons:

1. Program transformation is too tedious. Many steps are often needed to perform even relatively simple transformations.
2. Few good software tools exist for carrying out program transformation. Most of what is done in practice is done on paper, without formal verification.
3. There are foundational problems with some approaches to program transformation. For example, the well-known unfold/fold approach is not *safe*: terminating programs can be transformed into diverging ones.

In an attempt to bridge the gap between the theory and practice of program transformation, we are developing *PATH* (Programmer Assistant for Transforming Haskell), a program transformation system for the language Haskell [HJW92]. PATH, a programmer *assistant*, does not attempt to be fully automatic although it attempts to automate and mechanize as much as possible. For instance, PATH uses a GUI to navigate the program and apply transformations to it. PATH gives the user as much control as is safe, including letting the user write his own meta-programs (such as a deforestation algorithm). Because we give such control to the user, total correctness is considered essential.

1.2 Approaches to Program Transformation

There are a number of approaches which we could take to transforming a functional language such as Haskell. In their survey paper [PS83], Partsch and Steinbrueggen classify various approaches to program transformation into two basic approaches: (1) the schematic, or catalog, approach which is based on using a large catalog of rules, each performing a significant transformation and (2) the generative set approach, which is based on a small set of simple rules which in combination are very expressive. The Bird-Meertens Formalism (or Squiggol) [BdM97, Mee86, MFP91] is an example of the former. Unfold/fold [BD77] and expression procedures [Sch80] are examples of the latter.

Each approach has its advantages: The schematic approach can be more concise and allow for the development of powerful rules which do major transformations in a single step; the rules are symmetric, generally given in the form of transformation templates such as "$P_1 = P_2$ if C". However, a catalog may limit the possible transformations, especially in the presence of arbitrary recursive programs. A generative set approach, such as unfold/fold, is considered to be

*This research was supported in part by NSF under Grant Number CCR-9633390.

more general and works well with arbitrary recursive programs. This approach can be easier to use as there is no need to search a catalog. A disadvantage is that derivations are more verbose: many of the same steps (unfold, simplify, fold) are taken again and again.

A third approach would be to use a theorem prover into which the formal semantics (usually denotational) of the language was embedded, e.g., [Pau87]. Although this is the most general approach, it is also the most complicated: one must understand domain theory, inductive proof techniques, the logic of the theorem prover, etc.

1.3 Our Approach

Section 2 of this paper follows the evolution of the approach taken with PATH: We started off using the generative set approach because it is more general than the schematic approach and it is much simpler than theorem proving. Unwilling to use a method such as unfold/fold which does not preserve total correctness (section 2.1), we used Scherlis's expression procedures, a totally correct method (section 2.2). But while attempting to increase the power of expression procedures by eliminating their one-directional nature (section 2.3) (they were designed to specialize but not to generalize functions), we realized that the generality of expression procedures can be achieved by using a schematic rule, namely Fix-Point Fusion (section 2.4). As a result we can use the schematic approach with a very small base catalog to get a transformation system which is more powerful than expression procedures.

We then give the semantics and transformation rules of our system (section 3) and give examples of some program derivations which would be problematic with other approaches (section 4). We then show how we can very elegantly add Scherlis's "qualified expression procedures" (section 5).

2 From Unfold/Fold to Reversible Expression Procedures

In this section we compare various approaches to program transformation and introduce our reversible expression procedures.

2.1 Unfold/Fold

The best known approach to program transformation is the *unfold/fold* methodology of Burstall and Darlington [BD77]. Their approach is based on six rules: (1) *unfold*: the unfolding of function calls by replacing the call with the body of the function where actual parameters are substituted for formal parameters; (2) *laws*: the use of laws about the primitives of the language; (3) *instantiation*: adding an "instance" of a function definition in which a parameter is replaced by a constant or pattern on both sides of the definition; (4) *fold*: the replacement of an expression by a function call when the function's body can be instantiated to the given expression with suitable actual parameters—this can be done with any previous definition of the function; (5) *definition*: the addition of a new function definition; and (6) *abstraction*: the introduction of a `where` clause. This methodology is extremely effective at a broad range of program transformations.

Unfortunately, the unfold/fold methodology does not preserve total correctness; both the instantiation and fold rules are unsafe. Instantiation is easily modified to be safe, but fold is problematic. For example, consider the program

```
f x = x+1
```

Since the expression `x+1` is an instance of the right-hand-side of a function definition, we can replace it with a call to the function, yielding

```
f x = f x
```

which results in a non-terminating definition for `f`. Although this example is overly simple, similar situations can arise in more subtle contexts, and thus non-termination can inadvertently be introduced.

Several approaches have been proposed to solve this problem of partial correctness. One is to suitably constrain the use of fold, for example as proposed by Kott [Kot85]. Another is to provide a separate proof of termination. Yet another is the "tick algebra" of Sands, which guarantees incremental improvement in performance to all transformations [San95b].

Besides the problem with correctness, unfold/fold has a significant inconvenience in practice: a history must be kept of all versions of the program as it is being transformed (or the user must specify which versions to keep or throw away). This history is essential because previous definitions of functions are used to give folding its power.

2.2 Expression Procedures

Motivated by the problems with unfold/fold, Scherlis proposed *expression procedures* [Sch80, Sch81]. (More recently Sands [San95a] extended this work to a higher-order non-strict language.) Scherlis's key innovation was a new procedure definition mechanism in which the left hand side of an expression procedure definition can be an arbitrary expression: thus the name "expression procedure". In addition to laws about primitive functions, three rules are used to transform programs: *abstraction*, which introduces new procedures; *composition*, which introduces new expression procedures; and *application* (or unfold), which replaces a procedure call or expression procedure call with its definition.

An example of a program derivation using expression procedures (adapted from [San95a]) follows. Suppose we have these two function definitions

```
filter p []     = []
filter p (x:xs) = if p x then x : filter p xs
                         else     filter p xs

iterate f x = x : iterate f (f x)
```

and we wish to specialize the expression

```
filter p (iterate f x)
```

Our goal is to create a new version of `iterate` which is specialized to the context "`filter p _`". The first step is to introduce an expression procedure for this context, by filling in the hole ("`_`") with each side of the definition of `iterate` using the *composition* rule:

```
filter p (iterate f x) =ep=
  filter p (x : iterate f (f x))
```

We now have an expression procedure: the left hand side is not just a function symbol applied to variables and patterns, it is an arbitrary expression. (We use `=` for a regular function definition and `=ep=` for an expression procedure definition.) Now we transform the definition of this expression procedure to obtain

```
filter p (iterate f x) =ep=
  if p x then x : filter p (iterate f (f x))
         else     filter p (iterate f (f x))
```

Next, we notice that the context appears recursively, so by introducing a new function definition using the *abstraction* rule we can transform it to

```
filter p (iterate f x) =ep= filit p f x
filit p f x =
  if p x then x : filter p (iterate f (f x))
         else     filter p (iterate f (f x))
```

Finally, we use the *application* rule to "apply" the expression procedure, obtaining

```
filter p (iterate f x) =ep= filit p f x
filit p f x =
  if p x then x : filit p f (f x)
         else     filit p f (f x)
```

So, the original expression is equal to `filit p f x`, a specialized version of `iterate`.

On the one hand, expression procedures are strictly less powerful than unfold/fold (they can be simulated by unfold/fold); however, in practice, the great majority of unfold/fold transformations can be done just as well by expression procedures. We are not aware of any *useful*[1] and *total correctness preserving* unfold/fold transformations which cannot be done by expression procedures either directly or indirectly (by finding a "common ancestor" from which to derive the two programs we wish to show equivalent).

On the other hand, expression procedures have two key advantages over unfold/fold: (1) each of the transformation rules preserves total correctness, and (2) no history needs to be maintained, as all needed information is embedded in expression procedures; and when compared to various methods of ensuring total correctness in unfold/fold, expression procedures are both easier to use and more expressive:

- Unfold/fold followed by a proof of termination: Expression procedures are simpler as they need no separate proof of termination. They are more expressive as they can transform programs which may not terminate on all inputs (i.e., for which no proof of termination exists) [Sch80].
- Unfold/fold augmented with the methods of Kott or Sands: Kott adds a number of constraints to the form of program derivations and to the laws and functions usable (so much so as to make his method unusable in practice [Fir90, San96]), and Sands requires extra machinery with his "tick" calculus[2]. Both methods are essentially using the whole transformation history to ensure total correctness. Expression procedures, in contrast, are correct at each step irrespective of any history.

Besides the technical improvements, in practice expression procedures have a simpler and more intuitive method of program derivation: with unfold/fold, the ability to add a new "eureka" definition to a program is essential; but with expression procedures, the analogous operation is selecting a recursive function and some context in which to specialize it. Thus, entering eureka definitions by hand is replaced by selecting contexts in the program. This fits well with our vision of using a GUI to apply transformations to the program.

2.3 The Reversibility Problem

Although expression procedures are an improvement over unfold/fold, they have one significant shortcoming: it is easy to specialize a function, but it is not always even possible to generalize a function! This problem, shared with unfold/fold, comes about because the transformation rules are not reversible. For instance, given this definition of `reverse'`

```
reverse' ([],ys) = ys
reverse' (x:xs,ys) = reverse'(xs,x:ys)
```

it is easy to go from

```
reverse(xs) = reverse'(xs,[])
```

to

```
reverse([])   = []
reverse(x:xs) = reverse'(xs, x:[])
```

using unfold/fold (or expression procedures), but, as noted by Burstall and Darlington, it is not possible to derive the first program from the second. Let $P_1 \Rightarrow^{ep} P_2$ signify that we can derive the program P_2 from P_1 using some sequence of expression procedure rules. Note that $\Rightarrow^{ep}$ is not symmetric, nor is $\Rightarrow^{uf}$, the comparable derives relation for unfold/fold. Even when both $P_1 \Rightarrow^{ep} P_2$ and $P_2 \Rightarrow^{ep} P_1$, the derivation associated with $P_1 \Rightarrow^{ep} P_2$ may give no insight into how to find a derivation for $P_2 \Rightarrow^{ep} P_1$.

Is reversibility that important? We believe so, for two reasons: First, adding reversibility makes the system more expressive: as in the `reverse` example, we often want to make programs shorter or more modular; and even if we want a more efficient program, we sometimes need to make it less efficient before making it more efficient (such transformations are impossible with a method—such as expression procedures—in which every transformation step preserves or increases some measure of efficiency). Secondly, reversibility is important because the system becomes simpler if each rule is reversible: the user can learn one law and use it in two directions.

Note that to get reversibility, we could simply add a rule such as this: "if $P_2 \Rightarrow P_1$ then we can transform P_1 to P_2." Such a rule, called *redefinition*, was added to unfold/fold

[1] An example of a non-useful transformation is as follows [BD77, Zhu94]: `f x = 0` can be transformed to

```
f x = if x == 0 then 0 else f (x-1)
```

in unfold/fold, but the reverse transformation cannot be done. Expression procedures cannot transform in either direction.

[2] Sands's tick calculus couldn't prove the correctness of expression procedures: this seems to have been the motivation for his paper on expression procedures [San95a].

by Burstall and Darlington to get around the problem with the `reverse` derivation above. The disadvantage of this approach is that if we have P_1 and want to transform it, we need to know what P_2 is before we start—we can't derive it directly or incrementally from P_1; also, the addition of this ad hoc rule makes the system more complex.

Instead of adding a rule, we would rather modify the rules to make them all reversible; i.e., we want to find a characterization of expression procedures in terms of one or more transformation rules.

2.4 Reversible Expression Procedures

We want to make expression procedures reversible. The compose and apply rules are inherently one-directional, but what if we were to merge into one step all the steps involved in a prototypical expression procedure transformation? There are just four key steps (as seen in the example in section 2.2):

1. the *introduction* of the expression procedure (composition),
2. the *transformation* of the body of the expression procedure,
3. the use of *abstraction* to capture the resulting recursion, and
4. *application* of the expression procedure.

These four steps can be merged as follows. We begin with a strict program context `C[]` and a function definition `f = F[f]`. *Introduction* of the expression procedure (composition) gives

```
C[f] =ep= C[F[f]]
```

which is then *transformed* into the recursive expression procedure

```
C[f] =ep= G[C[f]]
```

for some context `G[]`. After we do *abstraction* we arrive at

```
C[f] =ep= g
g = G[C[f]]
```

Finally, *application* of the expression procedure yields

```
C[f] =ep= g
g = G[g]
```

The above steps can be merged into one rule (expressing the values of `f` and `g` as the fix-points μ`f.F[f]` and μ`g.G[g]`):

$$\texttt{C}[\mu\texttt{f.F[f]}] \Rightarrow^{ep} \mu\texttt{g.G[g]} \quad \text{if} \quad \texttt{C[F[f]]} \Rightarrow^{ep} \texttt{G[C[f]]}, \quad \texttt{C[]} \text{ strict}$$

This one rule replaces the three expression procedure rules—composition, abstraction, and application. We don't have reversibility yet, but if we could replace $\Rightarrow^{ep}$ with $=$ in the above rule we would have this reversible rule,

$$\texttt{C}[\mu\texttt{f.F[f]}] = \mu\texttt{g.G[g]} \quad \text{if} \quad \texttt{C[F[f]]} = \texttt{G[C[f]]}, \quad \texttt{C[]} \text{ strict}$$

a theorem of Stoy [Sto77]. So, we join the company of many who have rediscovered or used this theorem [AK82, GS90, MFP91]. Our contribution is showing its connection to expression procedures. Interestingly, it is a free theorem [Wad89] of the fix-point operator μ. We take its name, Fix-Point Fusion, from Meijer et al. [MFP91] where the power of the theorem is exploited considerably: most of their transformations are instances of this one general theorem.

Fix-Point Fusion can be used in both directions:

$\texttt{C}[\mu\texttt{f.F[f]}] \Rightarrow \mu\texttt{g.G[g]}$ specialization (fusion)
$\mu\texttt{g.G[g]} \Rightarrow \texttt{C}[\mu\texttt{f.F[f]}]$ generalization (fission)

To do fusion, `C[]` and `F[]` are known, and `G[]` is desired; so the premise is proven by finding a derivation `C[F[f]]` $\Rightarrow$ `G[C[f]]`. To do fission, `G[]` is known, the user provides `C[]`, and `F[]` is desired; so the premise is proven by finding a derivation `G[C[f]]` $\Rightarrow$ `C[F[f]]`. (Had an extra "redefinition" rule been added to expression procedures, the user would also need to know the answer, `F[]`, before proceeding.)

So, we have accomplished our goal: we can do expression procedure transformations with one reversible rule, Fix-Point Fusion. The advantages to using Fix-Point Fusion as a replacement for expression procedures are as follows:

- We now have a symmetric derives relation and need no ad hoc rules to get reversibility. (And henceforth we use $=$ rather than $\Rightarrow$.) Thus the system is simpler than expression procedures would be with an extra rule for reversibility: there is *one* rule which the user uses to both specialize and generalize, rather than an extra rule added to a set of "one directional" rules.
- We do not need to extend our base language with expression procedures. Although we would not need to actually implement expression procedures to use them (they are removed in the final program) we would need to add expression procedures to the operational semantics of the language, which complicates reasoning about the transformation system. With our approach there is just one theorem to prove.
- Derivations are now structured in a goal-directed fashion. Program derivations are structured as 1) *goal*: a function and its context are specified; and 2) *subgoal*: the derivation is developed which satisfies the sub-goal (thereby synthesizing the new definition). Besides clearly indicating the goal of each transformation, this allows all the sub-goals of an unreachable goal to be removed easily when the goal is removed. With unfold/fold and expression procedures the derivations *can* be, although seldom are, much more unstructured.

By showing how to do expression procedure transformations with a single transformation rule we have integrated two different approaches to doing program transformation: the schematic approach and the generative set approach (as described in the introduction). Chin and Darlington [CD89], seeing the need to integrate these two approaches, added schematic rules to unfold/fold along with a method to generate new schematic rules using unfold/fold. In contrast, our method of integration is to use the schematic approach in which we include a law which gives us the expressiveness of one generative set approach.

With Fix-Point Fusion we believe we have a solid basis for a totally correct, simple, and expressive transformation

system. We will next describe the language and transformation rules of our system in detail.

3 The Language and Transformation Rules

3.1 The Language

Our language is a typed, non-strict functional language with products, sums, integers, and an explicit fix-point operator (i.e., un-sugared Haskell)[3]. The syntax of programs is as follows:

```
e = λv.e | v | e e
  | (e,e) | fst e | snd e
  | L e | R e | case e of L v -> e; R v -> e
  | n | p
  | μv.e

n ∈ integer constants
v ∈ variables
p ∈ strict primitive functions over integers
```

We consider only well-typed programs. The operational semantics is given by applying the reduction rules (in Figure 1) from left to right with a leftmost outermost reduction strategy.

The following syntactic sugar is used here:

```
              [] ≡ L 0
         e1 : e2 ≡ R (e1,e2)
    [e1,...,en] ≡ e1:(... en:[])
let v = e1 in e2 ≡ (λv.e2) e1
        λ(x,y).e ≡ λv.e{fst v/x}{snd v/y}
        μ(x,y).e ≡ μv.e{fst v/x}{snd v/y}
if e1 then e2 else e3  ≡
  case e1 of L v -> e2; R v -> e3
```

We use $e\{e_1/v\}$ to represent the term e where free occurrences of v are replaced by e_1; The set of free variables in expression `e` is denoted by `fv(e)`. A *context* (ranged over by `C,D,E,F,G`) is an expression with zero or more holes, `[]`. `C[e]` is the expression with `C[]`'s holes replaced by `e`. A context `C[]` *traps* a variable `v` if at any hole in `C[]` the variable `v` is in scope. A context `C[]` is strict if for all closed expressions, `e`, and substitutions, σ, `e` diverges implies that σ`(C[e])` diverges. $e_1 \to e_2$ signifies that e_1 reduces to e_2 in one step and $e_1 \to^+ e_2$ signifies that e_1 reduces to e_2 in one or more steps. More formally, $\to$ is the least relation satisfying the reduction rules (Figure 1) and that is closed under $e_1 \to e_2 \Rightarrow C[e_1] \to C[e_2]$. We also define `*` as follows (note that `f*g` is strict if `f` and `g` are strict):

```
(f*g)(x,y) = (f x, g y)
```

3.2 The Transformation Rules

The transformation rules include the reduction rules as given in Figure 1 plus the additional rules given in Figure 2. (The notion of equivalence in these rules, $=$, is observational equivalence at base types, here just integers.)

[3] Previously our examples used recursion equations, but now we use an explicit fix-point operator in order to make our laws clearer.

Instantiation. The Instantiation rule lets us move strict contexts into case expressions (and in the reverse direction, out). It corresponds to the instantiation rule in unfold/fold (which only goes in one direction). The context `C[]` must be strict: as Sands [San95a] has noted, this corresponds to the conditions described by Runciman et al. [RFJ90] for safe instantiation in a non-strict language.

In actual use the strictness condition needed by the Instantiation and Fix-Point Fusion rules is often detected syntactically: the contexts defined by `S`, an extension of reduction contexts, are strict (where `e` is as defined above):

```
S = [] | S e | p S S | p e S | p S e
  | fst S | snd S
  | case S of L x-> e; R x-> e
```

It is interesting to note that Instantiation happens to be another free theorem, derivable from the type of case.

Eta. We have eta rules for functions, sums, *and* products. The language differs from un-sugared Haskell by having an unlifted product, not a lifted product; however, a lifted product can be had by simply wrapping a constructor around the unlifted product.

Fix-Point Fusion. This rule gives us the ability to transform recursive definitions. As noted previously, the Fix-Point Fusion rule lets us do transformations possible with expression procedures. It requires that the context `C[]` be strict. This strictness condition is necessary, otherwise non-termination could be introduced.

Fix-Point Expansion. This rule enables us to expand, or inline, the definition of a recursive definition inside itself.

4 Examples of Program Derivations

In this section we give some examples of program derivations. Although each of these examples is by necessity short, each demonstrates a transformation which is problematic using other approaches.

4.1 Infinite Lists

Here we derive a program which is an infinite list. Expression procedures are capable of doing this, but unfold/fold with a proof of termination would not. This is because the standard technique of using a well-founded ordering to prove termination of *functions* [Fir90, MW79] is inapplicable for recursively defined *data structures*. We assume that `map` and `succ` are predefined.

```
map succ (μones.1:ones)
_____________{Fusion} (= μtwos.?) _____________
∀ones.
  map succ (1:ones)
=                                          {reduce}
  succ 1 : map succ ones
_____________________________________________________
=
  μtwos. (succ 1) : twos
=                                          {reduce}
  μtwos. 2 : twos
```

$$
\begin{array}{rcll}
\texttt{p}\ n_1 \ldots n_n & \rightarrow & \texttt{n} & \text{where } \texttt{n} = [\![\texttt{p}]\!]\ n_1 \ldots n_n \\
\texttt{fst (e1,e2)} & \rightarrow & \texttt{e1} & \\
\texttt{snd (e1,e2)} & \rightarrow & \texttt{e2} & \\
\texttt{case L e1 of L v -> e2; R v -> e3} & \rightarrow & \texttt{e2\{e1/v\}} & \\
\texttt{case R e1 of L v -> e2; R v -> e3} & \rightarrow & \texttt{e3\{e1/v\}} & \\
(\lambda\texttt{v.e1})\ \texttt{e2} & \rightarrow & \texttt{e1\{e2/v\}} & \\
\mu\texttt{v.e1} & \rightarrow & \texttt{e1}\{\mu\texttt{v.e1/v}\} &
\end{array}
$$

Figure 1: Reduction Rules

Instantiation:

$$
\texttt{C[case e of L x -> e1; R x -> e2]} \quad = \quad \texttt{case e of L x -> C[e1]; R x -> C[e2]}
$$
if `C[]` strict, `fv(e)` not trapped by `C[]`, and `x` $\notin$ `fv(C[])`

Eta:

$$
\begin{array}{rcll}
\texttt{e} & = & \lambda\texttt{v.e v} & \text{if e} :: \alpha \rightarrow \beta \text{ and v} \notin \texttt{fv(e)} \\
\texttt{e} & = & \texttt{(fst e, snd e)} & \text{if e} :: (\alpha, \beta) \\
\texttt{e} & = & \texttt{case e of L x -> L x; R x -> R x} & \text{if e} :: \alpha \mid \beta
\end{array}
$$

Fix-Point Fusion:

$$
\frac{\forall \texttt{f.C[F[f]]} = \texttt{G[C[f]]}}{\texttt{C}[\mu\texttt{f.F[f]}] = \mu\texttt{g.G[g]}} \quad \text{if} \quad \texttt{C[]} \text{ strict, } \texttt{f}, \texttt{g} \text{ neither free in nor trapped by } \texttt{F[]} \text{ or } \texttt{G[]}
$$

Fix-Point Expansion:

$$
\texttt{e} \quad = \quad \mu\text{f.F[f]} \qquad \text{if } \texttt{e} \rightarrow^{+} \texttt{F[e]}
$$

Figure 2: Additional Transformation Rules

A word of explanation is needed about the format of our derivation given here: The horizontal line labeled {*Fusion*} marks where we start to derive the premise of Fix-Point Fusion. This sub-derivation is indented and when the premise is shown we know that the program above the sub-derivation is equal to the program below it. The `(= `μ`twos.?)` corresponds to what the user might enter to give a name to the variable bound by the new μ.

4.2 Fission: The Generalization of Recursive Definitions

The reverse of fusion—bringing a context and a function together—is fission, which splits a function into a context and a function. This cannot in general be done with expression procedures and was the original motivation for our work. Suppose, for example, that we wish to generalize `mapsucc` to the standard `map` function. (To make the derivation smaller we use the `map` which is defined only on infinite lists—a derivation for the standard `map` is the same number of steps but a larger program would be displayed at each step.)

```
μmapsucc.λxs.succ (head xs) : mapsucc (tail xs)
_____________{Fission} (= (μmap.?) succ) _____________
∀map.
  λxs.succ (head xs) : map succ (tail xs)
=                                        {abstract succ}
  (λh.λxs.h (head xs) : map h (tail xs)) succ
_____________________________________________________
=
(μmap.λh.λxs.h (head xs) : map h (tail xs))
  succ
```

If we read the derivation from bottom to top, we simply have the specialization of `map succ`. Here, when we do Fix-Point Fusion—or in this case, Fix-Point Fission—we use a similar notation but the

```
(= (μmap.?) succ)
```

(which corresponds to what the user would enter) gives the form of the desired program.

4.3 Tupling

This example does tupling, or loop fusion. It takes a two-pass average program and derives a one pass algorithm. This transformation is normally beyond the scope of fully automated strategies such as deforestation or partial evaluation (which is not surprising for a user-guided transformation system).

```
let sum = μsum.λxs.case xs of
                     []    -> 0;
                     x:xs' -> x + sum xs'
    len = μlen.λxs.case xs of
                     []    -> 0;
                     x:xs' -> 1 + len xs'

  λys.sum ys / len ys
=                                          {abstract}
  λys.let (s,l) = (sum ys, len ys) in s/l
=                                          {abstract}
  λys.let (s,l) = (λxs.(sum xs, len xs)) ys
      in s/l
```

Now we want to specialize

```
λxs.(sum xs, len xs)
```

but neither `sum` nor `len` is in a strict context, so expression procedures cannot specialize this program nor can Fix-Point Fusion be directly applied here. One way to get around this is to add a construct to the language by which we can express a strict product; but even for cases in which the product is not strict, we can continue with the help of the derived rules *FPF2* and *ABIDES* (derived rules whose definitions and derivations are in the Appendix).

```
λxs.(sum xs, len xs)
_____________{FPF2} (= μsumlen.?) _____________
∀sum,len.
  λxs.(case xs of []->0; x:xs'->x+sum xs',
       case xs of []->0; x:xs'->1+len xs')
=                                          {ABIDES}
  λxs.case xs of
        []   ->(0,0);
        x:xs'->(x+ sum xs',1+ len xs')
=                                       {abstraction}
  λxs.case xs of
        []   ->(0,0);
        x:xs'->let (s,l) = (sum xs',len xs')
               in  (x+s,1+l)
=                                       {abstraction}
  λxs.case xs of
        []   ->(0,0);
        x:xs'->
          let (s,l) = (λxs.(sum xs,len xs)) xs'
          in  (x+s,1+l)
_____________________________________________________
=
  μsumlen.λxs.case xs of
                []    -> (0,0);
                x:xs' -> let (s,l) = sumlen xs'
                         in  (x+s,1+l)
```

The rule *FPF2* is a variation on Fix-Point Fusion which is applicable when we have two recursive definitions. Interestingly, it is also a free theorem of μ: we get Fix Point Fusion (*FPF*) when we use a binary relation, we get *FPF2* when we use a ternary relation.

4.4 Mutually Recursive Expression Procedures

It is not obvious that Fix-Point Fusion along with the other laws can do all transformations that are possible with expression procedures: Can we do all transformations done by combining expression procedure rules in ways other than the prototypical order: composition, laws, abstraction, then application? In the future we would like to prove this; but for now, here is an expression procedure derivation that seems difficult to do with Fix-Point Fusion because the uses of composition, abstraction, and application are completely intertwined. Given this

```
f = F[f,g]
g = G[f,g]
```

and assuming

```
∀f,g. C[F[f,g]] => A[C[f],D[g]]
∀f,g. D[G[f,g]] => B[C[f],D[g]],
```

with expression procedures we can do this:

```
C[f] =ep= C[F[f,g]]
D[g] =ep= D[G[f,g]]
⇒                                          {assumption}
C[f] =ep= A[C[f],D[g]]
D[g] =ep= B[C[f],D[g]]
⇒                                          {abstract twice}
C[f]  =ep= f'
f' = A[C[f],D[g]]
D[g] =ep= g'
g' = B[C[f],D[g]]
⇒                        {apply C[f] twice; apply D[g] twice}
C[f]  =ep= f'
f' = A[f',g']
D[g] =ep= g'
g' = B[f',g']
```

However, this is just as easy to do with Fix-Point Fusion if we express the mutual recursion explicitly. Given this

```
(f,g) = μ(f,g).(F[f,g],G[f,g])
```

we can specialize both definitions at the same time as follows:

```
(C[f],D[g])
=                                             {def. of *}
(C*D)(f,g)
_____________{Fusion} (= μ(f',g').?) _____________
∀f,g.
  (C*D)(F[f,g],G[f,g])
=                                             {def. of *}
  (C[F[f,g]],D[G[f,g]])
=                                           {assumption}
  (A[C[f],D[g]], B[C[f],D[g]])
=                                             {def. of *}
  (A*B)(C[f],D[g])
=                                             {def. of *}
  (A*B)((C*D)(f,g))
_____________________________________________________
=
  μ(f',g').(A*B)(f',g')
=                                          {syntactic sugar}
  (f',g') where f' = A[f',g'] and g' = B[f',g']
```

5 Qualified Expression Procedures

In his thesis [Sch80] Scherlis noted that expression procedures allow us to specialize recursive functions in a syntactic context, but do not allow us to specialize functions based on non-syntactic information. For instance, expression procedures can specialize `f` in the syntactic context "`f x 0`" but couldn't take advantage of "`x > y`" in the specialization of "`f x y`".

To take advantage of the non-syntactic information available, Scherlis extended his system to support "qualified expression procedures"[4]. A qualified expression procedure looks like this

```
{p} e1 =ep= e2
```

in which `p` is a boolean valued expression similar to a precondition. In the transformation of the definition, we can assume `p` is true; the qualified expression procedure may only be applied where the qualifier is true.

Thanks to the non-strict semantics of our language and our schematic approach to expression procedures, we get the power of qualified expression procedures without having to add any ad hoc constructs to the language. Let's say we have an `assert` function defined as

```
assert p e = if p then e else error
```

[4] With these we get the power of Generalized Partial Computation [FN88, Tak91]. The importance of this extra information for specialization of programs is discussed in Sørenson et al. [SGJ94].

(where `error` is equivalent to $\perp$) for which we add some syntactic sugar:

```
{p} e = assert p e
{p _} e = {p e} e
```

With `assert` and some simple laws about it, we get the power of qualified expression procedures. Some easily proven laws regarding assertions are as follows:

Introducing/Eliminating Assertions

```
if p then a else b = if p then {p} a else b
if p then a else b = if p then a else {not p} b
```

Manipulating Assertions

```
{if p then p2 else p3} if p then a else b
                      =
      if p then {p2} a else {p3} b

        {p} = {p} {q}      if p => q
        {p} = {q}          if p = q
  {p} C[e] = C[{p} e]      if C[] strict
```

Using Assertions

```
                {e1=e2} D[e1] = {e1=e2} D[e2]
    {p} D[if p then a else b] = {p} D[a]
{not p} D[if p then a else b] = {not p} D[b]
```

Assertions can be used either as a run-time construct or as a construct which is introduced and removed during transformation. Nothing needs to be added to the language with our approach, nor do our transformation rules require any change, we just add the assertion laws. Assertions and Fix-Point Fusion allow us to bring pre-conditions into a recursion and to bring post-conditions out of a recursion as seen in following two laws, which follow directly from Fix-Point Fusion:

Where `f = `μ`f.`λ`x.F[f]` and `p` is strict.

Pre-condition:

$$\frac{\lambda x.\{p\ x\}\ F[f] = \lambda x.\{p\ x\}\ F[\lambda x.\{p\ x\}\ f\ x]}{\lambda x.\{p\ x\}\ f\ x = \mu f.\lambda x.\{p\ x\}\ F[f]}$$

Post-condition:

$$\frac{\lambda x.F[\lambda x.\{p\ \_\}\ (f\ x)] = \lambda x.\{p\ \_\}\ F[f]}{f = \lambda x.\{p\ \_\}\ (f\ x)}$$

6 Conclusion

In our attempt to increase the expressiveness of expression procedures we have made these contributions:

- We have shown that the essence of expression procedure transformations is Fix-Point Fusion. Thus, the power of expression procedures can be achieved with a schematic program transformation rule.
- As a result, we have integrated two different approaches to doing program transformation: the schematic approach and the generative set approach.
- We can do transformations which neither expression procedures nor safe restrictions of unfold/fold can do. This is a consequence of the reversibility inherent in the schematic approach.
- We have improved on the work of Scherlis and Sands. By doing expression procedures in one step, we gain simplicity over their approaches: (1) we have a simpler proof of correctness since we need only prove one law and need not show that a set of laws preserves consistency and progressiveness; (2) we can dispense with Sands's restriction on where we can perform abstraction in expression procedures (this restriction is motivated by the proof of correctness); and (3) we need not give a semantics to expression procedure definitions.
- We show how, with our approach, we get the expressiveness of Scherlis's qualified expression procedures by simply adding assertion laws.

Future work on PATH will include designing and implementing the user-interface, implementing various meta-programs, building a useful catalog of rules, and applying the system to more realistically sized programs. Other avenues of research related to the work here would be (1) developing a proof that our transformation rules really are as expressive as unrestricted expression procedures, (2) investigating the relative power of our system compared to unfold/fold, and (3) investigating the limits of our approach compared to, for instance, a theorem prover with fix-point induction.

Acknowledgements. We would like to thank David Sands for many helpful comments on an earlier version of this paper.

References

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, April 1982.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[CD89] Wei Ngan Chin and John Darlington. Schematic rules within unfold/fold approach to program transformation. In *TENCON '89: Fourth IEEE Region 10 International Conference*, 1989.

[Fir90] M. A. Firth. *A Fold/Unfold Transformation System for a Non-Strict Language*. PhD thesis, University of York, 1990.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[GS90] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 633–674. North-Holland, Amsterdam, 1990.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[Kot85] L. Kott. Unfold/fold transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 412–433. CUP, 1985.

[Mee86] L. Meertens. Algorithmics - towards programming as a mathematical activity. In J. W. de Bakker, E. M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. CWI Monographs, volume 1.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Verlag, June 1991. LNCS 523.

[MW79] Zohar Manna and Richard Waldinger. Synthesis: Dreams → programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, July 1979.

[Pau87] L. C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987.

[PS83] H. Partsch and R. Steinbrueggen. Program transformation systems. *ACM Computing Surveys*, 15:199–236, 1983.

[RFJ90] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 133–141, London, UK, 1990. Springer-Verlag. British Computer Society Workshops in Computing Series.

[San95a] D. Sands. Higher-order expression procedures. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 1995.

[San95b] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 221–232. ACM Press, 1995.

[San96] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[Sch80] W.L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, California, August 1980. Stanford Computer Science Report STAN-CS-80-818.

[Sch81] W.L. Scherlis. Program improvement by internal specialization. In *Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, January 1981*, pages 41–49. ACM, 1981.

[SGJ94] M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 485–500. Springer-Verlag, 1994.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. *ACM SIGPLAN Notices*, 26(9):1–11, 1991. Proceedings of Conference on Partial Evaluation and Semantics-Based Program Manipulation, New Haven, CT.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*. Springer Verlag, 1989.

[Zhu94] Hong Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, January 1994.

A Derived Rules

A.1 FPF2

Rule:

$$\frac{\texttt{C[] strict},\quad \texttt{∀d,e. C[(D[d],E[e])] = G[C[(d,e)]]}}{\texttt{C[(}\mu\texttt{d.D[d],}\mu\texttt{e.E[e])] = }\mu\texttt{g.G[g]}}$$

Derivation:

Assuming

```
C[] strict
∀d,e. C[(D[d],E[e])] = G[C[(d,e)]]
```

We get

```
  C[(μd.D[d], μe.E[e])]
=                                                        {SPLIT}
  C[μ(d,e).(D*E)(d,e)]
  ------------------{Fusion} (= μg.?) ------------------------
  ∀d,e.
    C[(D*E)(d,e)]
  =                                                      {def. of *}
    C[(D[d],E[e])]
  =                                                      {assumption}
    G[C[(d,e)]]
  ------------------------------------------------------------
=
  μg.G[g]
```

A.2 SPLIT

Rule:

$$\texttt{}\mu\texttt{(d,e).(D*E)(d,e) = (}\mu\texttt{d.D[d], }\mu\texttt{e.E[e])}$$

Derivation:

```
let x = μ(d,e).(D*E)(d,e)

  fst x
=                                          {def. of x}
  fst (μ(d,e).(D*E)(d,e))
=                                          {syntactic sugar}
  fst (μz.(D*E) z)
  ──────────────{Fusion} (= μd.?) ──────────────
  ∀z.
    fst ((D*E) z)
  =                                        {eta expansion}
    fst ((D*E) (fst z,snd z))
  =                                        {def. of *}
    fst (D[fst z],E[snd z])
  =                                        {reduction}
    D[fst z]
  ──────────────────────────────────────────────
=
  μd.D[d]
```

And similarly we get

```
snd x = μe.E[e]
```

Then we get

```
  x
=                                          {eta expansion}
  (fst x, snd x)
=                                          {above two laws}
  (μd.D[d], μe.E[e])
```

A.3 ABIDES

Rule:

```
          (case e of L x->a1; R y->a2,
           case e of L x->b1; R y->b2)
                          =
      case e of L x->(a1,b1); R y->(a2,b2)
```

Derivation:

```
  (case e of L x->a1; R y->a2,
   case e of L x->b1; R y->b2)
=                    {inverse reductions for fst and snd}
  (case e of L x->fst(a1,b1); R y->fst(a2,b2),
   case e of L x->snd(a1,b1); R y->snd(a2,b2))
=                             {reverse instantiation, twice}
  (fst (case e of L x->(a1,b1); R y->(a2,b2)),
   snd (case e of L x->(a1,b1); R y->(a2,b2)))
=                                          {eta contraction}
  case e of L x->(a1,b1); R y->(a2,b2)
```

Slicing Software for Model Construction

Matthew B. Dwyer* John Hatcliff†
Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA
{dwyer,hatcliff}@cis.ksu.edu

Abstract

Applying finite-state verification techniques (*e.g.*, model checking) to software requires that program source code be translated to a finite-state transition system that safely models program behavior. Automatically checking such a transition system for a correctness property is typically very costly, thus it is necessary to reduce the size of the transition system as much as possible. In fact, it is often the case that much of a program's source code is irrelevant for verifying a given correctness property.

In this paper, we apply program slicing techniques to remove automatically such irrelevant code and thus reduce the size of the corresponding transition system models. We give a simple extension of the classical slicing definition, and prove its safety with respect to model checking of linear temporal logic (LTL) formulae. We propose various refinements to slicing that take advantage of common structural patterns appearing in LTL software specifications. Finally, we discuss how this slicing strategy fits into a general methodology for deriving effective software models using abstraction-based program specialization.

1 Introduction

Modern software systems are highly complex, yet they must be extremely reliable and correct. In recent years, finite-state verification techniques, including model checking techniques, have received much attention as a software validation method. These techniques have been effective in validating crucial properties of concurrent software systems in a variety of domains including: network protocols [23], railway interlocking systems [5], and industrial control systems [3]. Despite this success, the high cost of automatically checking a given correctness property against a software system (which typically has an enormous state space) casts doubt on whether broad application of finite-state verification to software systems will be cost-effective.

Most researchers agree that the best way to attack the state-explosion problem is to construct a finite-state transition system that safely abstracts the software semantics [7, 10, 26]. The transition system should be small enough to make automatic checking tractable, yet it should large enough to capture all information relevant to the property being checked. One of the primary difficulties is determining which parts of the program are relevant to the property being checked. In this paper, we show how slicing can automatically throw away irrelevant portions of the software code, and hence safely reduce the size of the transition systems that approximates the software's behavior.

We envision slicing as one of a collection of tools for translating program source code to models that are suitable for verification. We previously illustrated how techniques from abstract interpretation and partial evaluation can be integrated and applied to help automate construction of abstract transition systems [11, 20, 21]. Applying these techniques on several realistic software systems [12, 13] has revealed an interesting interaction between slicing and abstraction building: people currently perform slicing-like operations manually to determine the portions of code that are relevant for verifying a given property. Thus, preprocessing software using slicing before applying partial-evaluation-based abstraction techniques can: *(i)* provide a safe approximation of the relevant portions of code, *(ii)* enable scaling of current manual techniques to significantly larger and more complex systems, *(iii)* reduce the number of components for which abstractions must be selected and help guide that selection, and *(iv)* reduce the size of the program to be treated by abstraction-based partial evaluation tools.

This work is part of a larger project on engineering high-assurance software systems. We are building a set of tools that implements the transition system construction methodology above for Ada and Java. In this paper, we use a simple flowchart language in order to formally investigate fundamental issues. We have implemented a prototype for the slicing system in the paper, and based on this we are scaling up the techniques. We refer the reader to the project web-site `http://www.cis.ksu.edu/santos/bandera` for the extended version of this paper (which contains more examples, technical extensions, and proofs), for the prototype, and for applications of the abstraction techniques to concurrent Ada systems.

In the next section, we describe the flowchart language that we use throughout the paper. We then present, in Section 3, the definition of slicing for this language. We discuss a specific finite-state verification technique, LTL model checking, and our approach to constructing safely abstracted transition systems from source code in Section 4. Section 5 describes how slicing can be applied as a pre-phase to transition system construction. Section 6 sketches several methods for deriving slicing criteria from temporal logic specifi-

*Supported in part by NSF and DARPA under grants CCR-9633388, CCR-9703094, CCR-9708184, and NASA under award NAG 21209.

†Supported in part by NSF under grant CCR-9701418, and NASA under award NAG 21209.

cations based on the shape of commonly-used formula patterns. Section 7 discusses related work on slicing, and Section 8 summarizes and concludes with a description of future work.

2 The Flowchart Language FCL

2.1 Syntax

We take as our source language the simple flowchart language FCL of Gomard and Jones [18, 25, 19]. Figure 1 presents an FCL program that computes the power function. The input parameters the program are `m` and `n`. These variables can be referenced and assigned to throughout the program. Other variables such as `result` can be introduced at any time. The initial value of a variable is 0. The output of program execution is the state of memory when the **return** construct is executed.

Figure 2 presents the syntax of FCL. FCL programs are essentially lists of *basic blocks*. The initial basic block to be executed is specified immediately after the parameter list. In the power program, the initial block is specified by the line `(init)`. Each basic block consists of a *label* followed a (possibly empty) list of *assignments* (we write $\cdot$ for the empty list, and this is elided when the list is non-empty). Each block concludes with a *jump* that transfers control from that block to another one. Instead of including boolean values, any non-zero value represents *true* and zero represents *false* in the test of conditionals.

In the presentation of slicing, we need to reason about nodes in a *statement-level control-flow graph* (CFG) (*i.e.*, a graph where there is a separate node for each assignment and jump) for given program p. We will assume that each statement has a unique index i within each block. Then, nodes can be uniquely identified by a pair $[l.i]$ where l is block label and i is an index value. In Figure 1, statement indices are given as annotations in brackets $[\cdot]$. For example, the second assignment in the `loop` block has the unique identifier (or node number) [`loop`.2].

The following definition introduces notions related to statement-level control-flow graphs.

Definition 1

- *A* flow graph $G = (N, E, s, e)$ *consists of a set N of statement nodes, a set E of directed control-flow edges, a unique start node s, and unique end node e.*
- *The* inverse G^{-1} *of a flowgraph (N, E, s, e) is the flowgraph (N, E^{-1}, e, s) (i.e., all edges are reversed and start/end states are swapped).*
- *Node n* dominates *node m in G (written $dom(n, m)$) if every path from the start node s to m passes through n. (note that this makes the dominates relation reflexive).*
- *Node m* post-dominates *node n in G (written post-$dom(m, n)$) if every path from node m to the end node e passes through n (equivalently, $dom(n, m)$ in G^{-1}).*
- *Node n is* control-dependent *on m (some intuition follows this definition) if*
  1. *there exists a non-trivial path p from m to n such that every node $m' \in p - \{m, n\}$ is post-dominated by n, and*
  2. *m is not post-dominated by n. [33]*

  We write $cd(n)$ for the set of nodes on which n is control-dependent.

Control dependence plays an important role in the rest of the paper. Note that for a node n to be control-dependent on m, m must have a least two exit edges, and there must be two paths that connect m with e such that one contains n and the other does not. For example, in the power program of Figure 1, [`loop`.1], [`loop`.2], and [`loop`.3] are control-dependent on [`test`.1], but [`end`.1] is not since it post-dominates [`test`.1] (*i.e.*, all paths from [`test`.1] to `halt` go through it).

Extracting the CFG from an FCL program p is straightforward. The only possible hitch is that some programs do not satisfy the "unique end node" property required by the definition (for example, the program may have multiple **return**'s). To work around this problem, we assume that when we extract the CFG from a program p, we insert an additional node labeled `halt` that has no successors and its predecessors are all the **return** nodes from p.

2.2 Semantics

The semantics of an FCL program p is expressed as transitions on program states $([l.n], \sigma)$ where l is the label of a block in p, n is the index of the statement in block l, and σ is a store mapping variables to values. A series of transitions gives an *execution trace* through a program's statement-level control flow graph. For example, Figure 3 gives a trace of the power program computing 5^2. Formally, a trace is finite nonempty sequence of states written $\pi = s_1, s_2, \ldots, s_k$. We write π^i for the suffix starting at s_i, *i.e.*, $\pi^i = s_i, s_{i+1}, \ldots$ [1]. We omit a formal definition of the transition relation for FCL programs since it is intuitively clear (a formalization can be found in [19, 20]).

3 Slicing

3.1 Program slices

A *program slice* consists of the parts of a program p that (potentially) affect the variable values that flow into some program point of interest [31]. A *slicing criterion* $C = (n, V)$ specifies the program point n (a node in p's CFG) and a set of variables V of interest.

For example, slicing the *power* program with respect to the slicing criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$ yields the program in Figure 4. Note that the assignments to variables `m` and `result` and the declaration of `m` as an input parameter have been sliced away since they do not affect the value of `n` at line [`loop`.2]. In addition, block `init` is now trivial and can be removed, *e.g.*, in a post-processing phase.

Slicing a program p yields a program p_s such that the traces of p_s are projections of corresponding traces of p. For example, the following trace of p_s is a projection of the trace

[1] Here, we consider only finite traces (corresponding to terminating executions). The extended version of the paper treats infinite executions, which are best expressed using co-inductive reasoning

```
(m n)
(init)

  init: result := 1;   [1]          loop: result := *(result m);  [1]
        goto test;     [2]                n := -(n 1);            [2]
                                          goto test;              [3]
  test: if <(n 1)      [1]
           then end
           else loop;                end:  return;                 [1]
```

Figure 1: An FCL program to compute m^n

Syntax Domains

p	$\in$	Programs[FCL]	x	$\in$	Variables[FCL]
b	$\in$	Blocks[FCL]	e	$\in$	Expressions[FCL]
l	$\in$	Block-Labels[FCL]	c	$\in$	Constants[FCL]
a	$\in$	Assignments[FCL]	j	$\in$	Jumps[FCL]
al	$\in$	Assignment-Lists[FCL]	o	$\in$	Operations[FCL]

Grammar

p	::=	$(x^*)\ (l)\ b^+$	a	::=	x **:=** e; $\mid$ **skip**;
b	::=	$l:\ al\ j$	e	::=	$c \mid x \mid o(e^*)$
al	::=	$a\ al \mid \cdot$	j	::=	**goto** l; $\mid$ **return**; $\mid$ **if** e **then** l_1 **else** l_2;

Figure 2: Syntax of the Flowchart Language FCL

$$
\begin{array}{rll}
 & ([\texttt{init}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]) & \\
\to & ([\texttt{init}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) & \\
\to & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) & \\
\to & ([\texttt{loop}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) & \\
\to & ([\texttt{loop}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 2, \texttt{result} \mapsto 5]) & \\
\to & ([\texttt{loop}.3], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) & \\
\to & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) \ldots & \\
\ldots \to & ([\texttt{loop}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 5]) & \\
\to & ([\texttt{loop}.2], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 1, \texttt{result} \mapsto 25]) & \\
\to & ([\texttt{loop}.3], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) & \\
\to & ([\texttt{test}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) & \\
\to & ([\texttt{end}.1], [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) & \\
\to & (\texttt{halt}, [\texttt{m} \mapsto 5, \texttt{n} \mapsto 0, \texttt{result} \mapsto 25]) &
\end{array}
$$

Figure 3: Trace of power program with $\texttt{m} = 5$ and $\texttt{n} = 2$

```
(n)
(init)

  init: goto test;     [2]          loop: n := -(n 1);            [2]
                                          goto test;              [3]
  test: if <(n 1)      [1]
           then end
           else loop;                end:  return;                 [1]
```

Figure 4: Slice of *power* with respect to criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$

in Figure 3.

$$\begin{array}{rl} & ([\texttt{init}.2], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{test}.1], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{loop}.2], [\texttt{n} \mapsto 2]) \\ \rightarrow & ([\texttt{loop}.3], [\texttt{n} \mapsto 1]) \\ \rightarrow & \cdots \\ \rightarrow & ([\texttt{end}.1], [\texttt{n} \mapsto 0]) \\ \rightarrow & (\texttt{halt}, [\texttt{n} \mapsto 0]) \end{array}$$

Intuitively, a trace π_2 is a projection of a trace π_1 if the sequence of program states in π_2 can be embedded into the sequence of states in π_1. To formalize this, let $\sigma\,|_V$ denote the restriction of the domain of σ to the variables in V. Then, the definition of projection is as follows.

Definition 2 (projection) *Let p be a program. A projection function $\downarrow[M, \nu]$ for p-traces is determined by*

- *a set of nodes M from p's CFG, and*
- *a function ν that maps each node in M to a set of variables V*

and is defined by induction on the length of traces as follows:

$$\downarrow[M,\nu]((n,\sigma),\, s_2,\, \ldots\,, s_k) = \begin{cases} (n, \sigma\,|_{\nu(n)}), \downarrow[M,\nu](s_2,\, \ldots\,, s_k) & \text{if } n \in M \\ \downarrow[M,\nu](s_2,\, \ldots\,, s_k) & \text{if } n \notin M \end{cases}$$

In the classical definition [31, 32] of slicing criterion, one specifies exactly one point node of interest in the CFG along with a set of variables of interest at that node. This was the case with the example slice of the power program above.

For our applications, we may be interested in *multiple* program points, and so we generalize the notion of slicing criterion as follows.

Definition 3 (slicing criterion) *A slicing criterion C for a program p is a non-empty set of pairs*

$$\{(n_1, V_1),\, \ldots\,,\, (n_k, V_k)\}$$

where each n_i is a node in p's statement flow-graph and V_i is a subset (possibly empty) of the variables in p. The nodes n_i are required to be pairwise distinct.

Note that a criterion C can be viewed as a function from $\{n_1, \ldots, n_k\}$ to $\wp$(Variables[FCL]). In this case, we write *domain*(C) to denote $\{n_1, \ldots, n_k\}$. Thus, a slicing criterion C determines a projection function $\downarrow[\mathit{domain}(C), C]$ which we abbreviate as $\downarrow[C]$. We can now formalize the notion of program slice.

Definition 4 (program slice) *Given program p with an associated CFG, let C be a slicing criterion for p. Then a program p_s (also called the* residual *program) is slice of p with respect to C if for any p execution trace $\pi = s_0, \ldots, s_k$,*

$$\downarrow[C](\pi) \;= \downarrow[C](\pi_s)$$

where π_s is the execution trace of p_s running with initial state s_0.

For example, let $C = \{([\texttt{loop}.2], \{\texttt{n}\})\}$, and let π and π_s be the execution traces of the power program (Figure 1) and the slice of the power program (Figure 4), respectively. Then,

$$\begin{array}{rl} & \downarrow[C](\pi) \\ = & \downarrow[C](\pi_s) \\ = & ([\texttt{loop}.2], [\texttt{n} \mapsto 2]), ([\texttt{loop}.2], [\texttt{n} \mapsto 1]). \end{array}$$

3.2 Computing slices

Given a program p and slicing criterion C, Definition 4 admits many programs p_s as slices of p (in fact, p itself is a (trivial) slice of p). Weiser notes that the problem of finding a statement minimal slice of p is incomputable [32]. Below we give a minor adaptation of Weiser's algorithm for computing conservative slices, *i.e.*, slices that may contain more statements than necessary.[2]

3.2.1 Initial approximation of a slice

Computing a slice involves (among other things) identifying assignments that can affect the values of variables given in the slicing criterion. To do this, one computes information similar to *reaching definitions*. This requires keeping track of the variables referenced and the variables defined at each node in the CFG.

Definition 5 (definitions and references)

- *Let def(n) be the set of variables defined (i.e., assigned to) at node n (always a singleton or empty set).*
- *Let ref(n) be the set of variables referenced at node n.*

Figure 5 shows the *def* and *ref* sets for the power program of Figure 1.

Next, for each node in the CFG we compute a set of *relevant variables*. Relevant variables are those variables whose values must be known so as to compute the values of the variables in the slicing criterion.

Definition 6 (initially relevant variables) *Let $C = \{(n_1, V_1), \ldots, (n_k, V_k)\}$ be a slicing criterion. Then $R^0_C(n)$ is the set of all variables v such that either:*

1. *$n = n_i$ and $v \in V_i$ for some $i \in \{1, \ldots, k\}$, or*
2. *n is an immediate predecessor of a node m such that either:*
   - *(a) $v \in$ ref(n) and def$(n) \cap R^0_C(m) \neq \emptyset$, or*
   - *(b) $v \notin$ def(n) and $v \in R^0_C(m)$.*

Intuitively, a variable v is relevant at node n if (1) we are at the line of the slicing criterion and we are slicing on v, or (2) n immediately precedes a node m such that (a) v is used to define a variable x that is relevant at m (*i.e.*, the value of x depends on v), or (b) v is relevant at m and it is not "killed off" by the definition at line n. Figure 5 presents the initial sets of relevant variables sets for the power program of Figure 1 with slicing criterion $C = ([\texttt{loop}.2], \{\texttt{n}\})$. Intuitively, `n` is relevant along all paths leading into node [`loop`.2]. In the `end` block, `n` is a dead variable and thus it is no longer relevant.

The classical definition of slicing does not require nodes mentioned in the slice criterion to occur in the computed slice. To force these nodes to occur, we define a set of *obligatory nodes* — nodes that must occur in the slice even if they fail to define variables that are eventually deemed relevant.

Definition 7 (obligatory nodes) *The set O_C of obligatory nodes is defined as follows:*

$$O_C \;=\; \{n \in N \mid (n, V) \in C\}$$

[2] The algorithm we give actually is based on Tip's corrected version of Weiser's algorithm [31].

Node	*def*	*ref*	*cd*	R_C^0	R_C^1
[init.1]	{result}	∅	∅	{n}	{n}
[init.2]	∅	∅	∅	{n}	{n}
[test.1]	∅	{n}	{[test.1]}	{n}	{n}
[loop.1]	{result}	{result, m}	{[test.1]}	{n}	{n}
[loop.2]	{n}	{n}	{[test.1]}	{n}	{n}
[loop.3]	∅	∅	{[test.1]}	{n}	{n}
[end.1]	∅	∅	∅	∅	∅

$$O_C = \{[\texttt{loop.2}]\} \qquad \begin{array}{l} S_C^0 = \{[\texttt{loop.2}]\} \\ B_C^0 = \{[\texttt{test.1}]\} \end{array} \qquad \begin{array}{l} S_C^1 = \{[\texttt{loop.2}], [\texttt{test.1}]\} \\ B_C^1 = \{[\texttt{test.1}]\} \end{array}$$

Figure 5: Results of the slicing algorithm for the power program and slicing criteria $C = ([\texttt{loop.2}], \{\texttt{n}\})$

The initial slice set S_C^0 is the set of nodes that define variables that are relevant at a successor.

Definition 8 (initial slice set) *The initial slice set S_C^0 is defined as follows:*

$$\{n \in N \mid \exists m \,.\, (n,m) \in E \wedge R_C^0(m) \cap \mathit{def}(n) \neq \emptyset\}$$

Figure 5 presents the initial slice set S_C^0 for the power program of Figure 1. Node [loop.2] is the only node in S_C^0 since it is the only node that defines a variable that is relevant at a successor.

Note that S_C^0 does not include any conditionals since conditionals make no definitions. How do we tell what conditionals should be added? Intuitively, a conditional at node n should be added if $m \in S_C^0 \cup O_C$ and m is control-dependent on n. This set of conditionals B_C^0 is called the *branch set.*

Definition 9 (branch set) *The initial branch set B_C^0 is defined as follows:*

$$B_C^0 = \bigcup_{n \in (S_C^0 \cup O_C)} \mathit{cd}(n)$$

Figure 5 presents the control-dependence information and the initial branch set B_C^0 for the power program of Figure 1. As explained in Section 2.1, [loop.1], [loop.2], and [loop.3] are control-dependent on [test.1]. Since [loop.2] is in $S_C^0 \cup O_C$, control-dependency dictates that [test.1] be included in the B_C^0.

3.3 Iterative construction

Now we have to keep iterating this process. That is, we add the conditionals that influence nodes already in the slice. Then, we must add to the slice nodes that are needed to compute expressions in the tests of conditionals, and so on until a fixed point is reached.

Definition 10 (iterations)

- *relevant variables*

$$R_C^{i+1}(n) = R_C^i(n) \cup \bigcup_{b \in B_C^i} R^0_{bc(b)}(n)$$

where the branch criterion $bc(b) = \{(b, \mathit{ref}(b))\}$ That is, the relevant variables at node n are those that were relevant in the previous iteration, plus those that are needed to decide the conditionals that control definitions in the previous slice set. Finding such nodes for a branch b is equivalent to slicing the program with the criterion $\{(b, \mathit{ref}(b))\}$.

- *slice set*

$$S_C^{i+1} = \{n \in N \mid (n \in B_C^i) \vee (\exists m \,.\, (n,m) \in E \wedge R_C^{i+1}(m) \cap \mathit{def}(n) \neq \emptyset)\}$$

That is, the slice set contains all the conditionals that controlled nodes in the previous slice set, and all nodes that define relevant variables.

- *branch set*

$$B_C^{i+1} = \{b \in N \mid \exists n \in S_C^{i+1} \cup O_C \,.\, b \in \mathit{cd}(n)\}$$

That is, the conditionals required are those that control nodes in the current slice set or obligatory nodes.

Figure 5 presents the sets R_C^1, S_C^1, and B_C^1 which result from the second iteration of the algorithm. On the next iteration, a fixed point is reached since n is the only variable required to compute the conditional test at [test.1] and it is already relevant at [test.1].

In the iterations, the size of $R_C^i(n)$ for all nodes n and S_C^i is increasing, and since $R_C^i(n)$ is bounded above by the number of variables in the program and S_C^i is bounded above by the number of nodes in the CFG, then the iteration eventually reaches a fixed points $R_C^i(n)$ and S_C^i.

3.4 Constructing a residual program

Given R_C and S_C, the following definition informally summarizes how a residual program is constructed. The intuition is that if an assignment is in S_C, then it must appear in the residual program. If the assignment is not in S_C but in O_C, then the assignment must be to an irrelevant variable. Since the node must appear in the residual program, the assignment is replaced with a **skip**. All **goto** and **return**

jumps must appear in the residual program. However, if an **if** is not in S_C, then no relevant assignment or obligatory node is control dependent upon it. Therefore, it doesn't matter if we take the true branch or the false branch. In this case, we can simply jump to the point where the two branches merge.

Definition 11 (residual program construction) *Given a program $p = (x^*)\ (l)\ b^+$ and slicing criterion C, let R_C, S_C, O_C be the sets constructed by the process above. A residual program p_s is constructed as follows.*

- *For each parameter x in p, x is a parameter in p_s only if $x \in R_C([l.1])$ where l is the label of the initial block of p.*
- *The label of the initial block of p_s is the label of the initial block of p.*
- *For each block b in p, form a residual block b_s as follows.*
  - *For each assignment line a (with identifier $[l.i]$), if $[l.i] \in S_C$ then assignment a appears in the residual program with identifier $[l.i]$, otherwise if $[l.i] \in O_C$ then the assignment becomes a* **skip** *with identifier $[l.i]$ in the residual program, otherwise the node is left out of the residual program.*
  - *For jump j in b, if $j =$* **goto** *l'; or $j =$* **return**; *then j is the jump in b_s, otherwise we must have $j =$* **if** *e* **then** *l_1* **else** *l_2; with some identifier $[l.i]$. Now if $[l.i] \in S_C$ then j is the jump in b_s, otherwise the jump in b_s is* **goto** *l'; with identifier $[l.i]$ where l' is the label of the nearest post-dominating block for both l_1 and l_2.*

Finally, post-processing removes all blocks that are not targets of jumps in p_s (these have become unreachable).

4 Finite-state Verification

As noted in the introduction, a variety of finite-state verification techniques have been used to verify properties of software. To make our presentation more concrete, we consider a single finite-state verification technique: model checking of specifications written in linear temporal logic (LTL). LTL model checking has been used to reason about properties of a wide range of real software systems; we have used it, for example, to validate properties of a programming framework that provides parallel scheduling in a variety of systems (e.g., parallel implementations of finite-element, computational fluid dynamics, and program flow analysis problems) [16, 15].

4.1 Linear temporal logic

Linear temporal logic [27] is a rich formalism for specifying state and action sequencing properties of systems. An LTL specification describes the intended behavior of a system on all possible executions.

The syntax of LTL includes primitive propositions P with the usual propositional connectives, and three temporal operators.

$$\begin{array}{rcl} & & \textit{(propositional connectives)} \\ \psi & ::= & P \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \Rightarrow \psi_2 \mid \\ & & \textit{(temporal operators)} \\ & & \Box\psi \mid \Diamond\psi \mid \psi_1\ \mathcal{U}\ \psi_2 \end{array}$$

When specifying properties of software systems, one typically uses LTL formulas to reason about execution of particular program points (*e.g.*, entering or exiting a procedure) as well as values of particular program variables. To capture the essence of this for FCL, we use the following primitive propositions.

$$P \quad ::= \quad [l.i] \mid [x\ rop\ c]$$

- Intuitively, $[l.i]$ holds when execution reaches node i in the block labeled l.
- Intuitively, $[x\ rop\ c]$ holds when the value of variable x at the current node is related to $[\![c]\!]$ by the relational operator *rop* (*e.g.*, [x=0] where *rop* is $=$).

Formally, the semantics of a primitive proposition is defined with respect to states.

$$[\![[l.i]]\!](n,\sigma) = \begin{cases} \mathit{true} & \text{if } n = [l.i] \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$[\![[x\ rop\ c]]\!](n,\sigma) = \begin{cases} \mathit{true} & \text{if } \sigma(x)\ [\![rop]\!]\ [\![c]\!] \\ \mathit{false} & \text{otherwise} \end{cases}$$

The semantics of a formula is defined with respect to a trace. The temporal operator $\Box$ requires that its argument be true from the current state onward, the $\Diamond$ operator requires that its argument become true at some point in the future, and the $\mathcal{U}$ operator requires that its first argument is true up to the point where the second argument becomes true. Formally [24], let $\pi = s_1, \ldots, s_k$. Then,

$$\begin{array}{rcl} \pi \models [l.i] & \text{iff} & [\![[l.i]]\!]s_1 = \mathit{true} \\ \pi \models [x\ rop\ c] & \text{iff} & [\![[x\ rop\ c]]\!]s_1 = \mathit{true} \\ \pi \models \psi_1 \wedge \psi_2 & \text{iff} & \pi \models \psi_1 \text{ and } \pi \models \psi_2 \\ \pi \models \psi_1 \vee \psi_2 & \text{iff} & \pi \models \psi_1 \text{ or } \pi \models \psi_2 \\ \pi \models \psi_1 \Rightarrow \psi_2 & \text{iff} & \pi \models \psi_1 \text{ implies } \pi \models \psi_2 \\ \pi \models \Box\psi & \text{iff} & \pi^i \models \psi \text{ for all } i \\ \pi \models \Diamond\psi & \text{iff} & \pi^i \models \psi \text{ for some } i \\ \pi \models \psi_1\ \mathcal{U}\ \psi_2 & \text{iff} & \text{there exists an } i \text{ such that} \\ & & \pi^i \models \psi_2, \text{ and for all} \\ & & j = 1, \ldots, i-1, \pi^j \models \psi_1 \end{array}$$

Here are some simple specifications using the logic:

- $\Diamond$[15.1]
  eventually block l5 will be executed
- $\Box([12.1] \Rightarrow \Diamond[13.1])$
  whenever block l2 is executed, block l3 is always subsequently executed
- $\Box([15.1] \Rightarrow \neg x = 0)$
  whenever block l5 is executed x is non-zero
- $\Box$(x<10)
  x is always less than 10

4.2 Software model construction

To apply finite-state verification to a software system, one must construct a finite-state transition system that safely abstracts the software semantics. The transition system should be small enough to make automatic checking tractable, yet it should be large enough to capture all information relevant to the property being checked. Relevant information can by extracted by an appropriate abstract interpretation (AI) [9].

In our approach [12, 20], the user declares for each program variable an abstract domain to be used for interpreting operations on the variable. Using a process that combines abstract interpretation and partial evaluation (which we call *abstraction-based program specialization* (ABPS)), a residual program is created by propagating abstract values and specializing each program point with respect to these abstract values [20, 21]. In the residual program, concrete constants are replaced with abstract constants. The residual program is a safely approximating finite-state program with a fixed number of variables defined over finite abstract domains. This program can then be submitted to a toolset [8, 14] that generates input for existing model checking tools, such as SMV [28] and SPIN [23]. This approach has been applied to verify correctness properties of several software systems written in Ada [12, 13].

In the steps described above, *the user's main task is to pick appropriate AI's*, *i.e.*, AI's that extract relevant information, but throw away irrelevant information. The general idea behind our methodology for chosing AI's is to start simple (use an AI's that throw all information about dataflow away) and incrementally refine the AI's based on information from the specification to be verified and from the program.

1. **Start with the point AI:** Initially all variables are modeled with the *point* AI (*i.e.*, a domain with a single value $\top$ where all operations return $\top$). In effect, this throws away all information about a variable's value.

2. **Identify semantic features in the specification:** The specification formula to be checked includes, in the form of propositions, different semantic features of the program (e.g., valuations of specific program variables). These features must be modeled precisely by an AI to have any hope of checking the property. For example, if the formula includes a proposition [x=0], then instead of using the point AI for x, one must use *e.g.*, an AI with the domain {zero, pos, $\top$} that we refer to as a *zero-pos* AI.

3. **Select controlling variables:** In addition to variables mentioned explicitly in the specification, we must also use refined AI's for variables on which specification variables are control dependent. The predicates in the controlling conditionals suggest semantic features that should be modeled by an AI. For example, if a specification variable x is control-dependent upon a conditional **if**`even`?(y) ... one should use an even/odd AI for y.

4. **Select variables with broadest impact:** When confronted with multiple controlling variables to model, select the one that appears most often in a conditional.

To illustrate the methodology, Figure 6 presents an FCL rendering of an Ada process that controls readers and writers of a common resource [8]. In the Ada system, this server process runs concurrently with other client processes, and requests such as `start-read`, `stop-read` are entry points (rendezvouz points) in the control process. In the FCL code of Figure 6, requests are given in the program parameter `reqs` – a list of values in the subrange [1..4]. Figure 7 presents the block-level control-flow graph for the FCL program.

Assume we are interested in reasoning about the invariance property

$$\Box([\texttt{start-read.1}] \Rightarrow [\texttt{WriterPresent=0}]).$$

The key features that are mentioned explicitly in this specification are values of variable `WriterPresent` and execution of the `start-read` block. The point AI does not provide enough precision to determine the states where `WriterPresent` has value zero. An effective AI for `WriterPresent` must be able to distinguish zero values from positive values; we choose the *zero-pos* AI.

At this point we could generate an abstracted model and check the property or consider additional refinements of the model; we choose the latter for illustrating our example. We now determine the variables upon which the node [`start-read.1`] and nodes with assignments to `WriterPresent` are control dependent. In our example, there are three such variables: `WriterPresent`, `ActiveReaders` and `req`. We are already modeling `WriterPresent` and `req` is being used to model external choice of interactions with the control program via input. We could choose to bind `ActiveReaders` to a more refined AI than point. Given that the conditional expressions involving that variable are `ActiveReaders=0` and `ActiveReaders>0`, we might also choose a *zero-pos* AI. Thus, only `ErrorFlag` is abstracted using the point AI.

At this point, we would generate an abstracted model and check the property. If a true result is obtained then we are sure that the property holds on the program, even though the finite-state system only models two variables with any precision. If a false result is obtained then we must examine the counter-example produced by the model checker. It may reveal a true defect in the program or it may reveal an infeasible path through the model. In the latter case, we identify the variables in the conditionals along the counter-example's path as candidates for binding to more precise AIs.

This methodology is essentially a heuristic search to find the variables in the program that can influence the execution behavior of the program relative to the property's propositions. When a variable is determined to be potentially influential, its abstraction is refined to strengthen the resulting system model. In the absence of such a determination, the variable is modeled with a *point* abstraction which essentially ignores any effect it may have; although in the future it may be determined to have an influence in which case its abstraction will be refined.

5 Reducing Models Using Slicing

As illustrated above, picking appropriate abstractions is nontrivial and could benefit greatly from some form of automated assistance. The key aspects of the methodology for picking abstractions included

1. picking out an initial set of relevant variables V and relevant statements (*i.e.*, CFG nodes N) mentioned in the LTL specification,

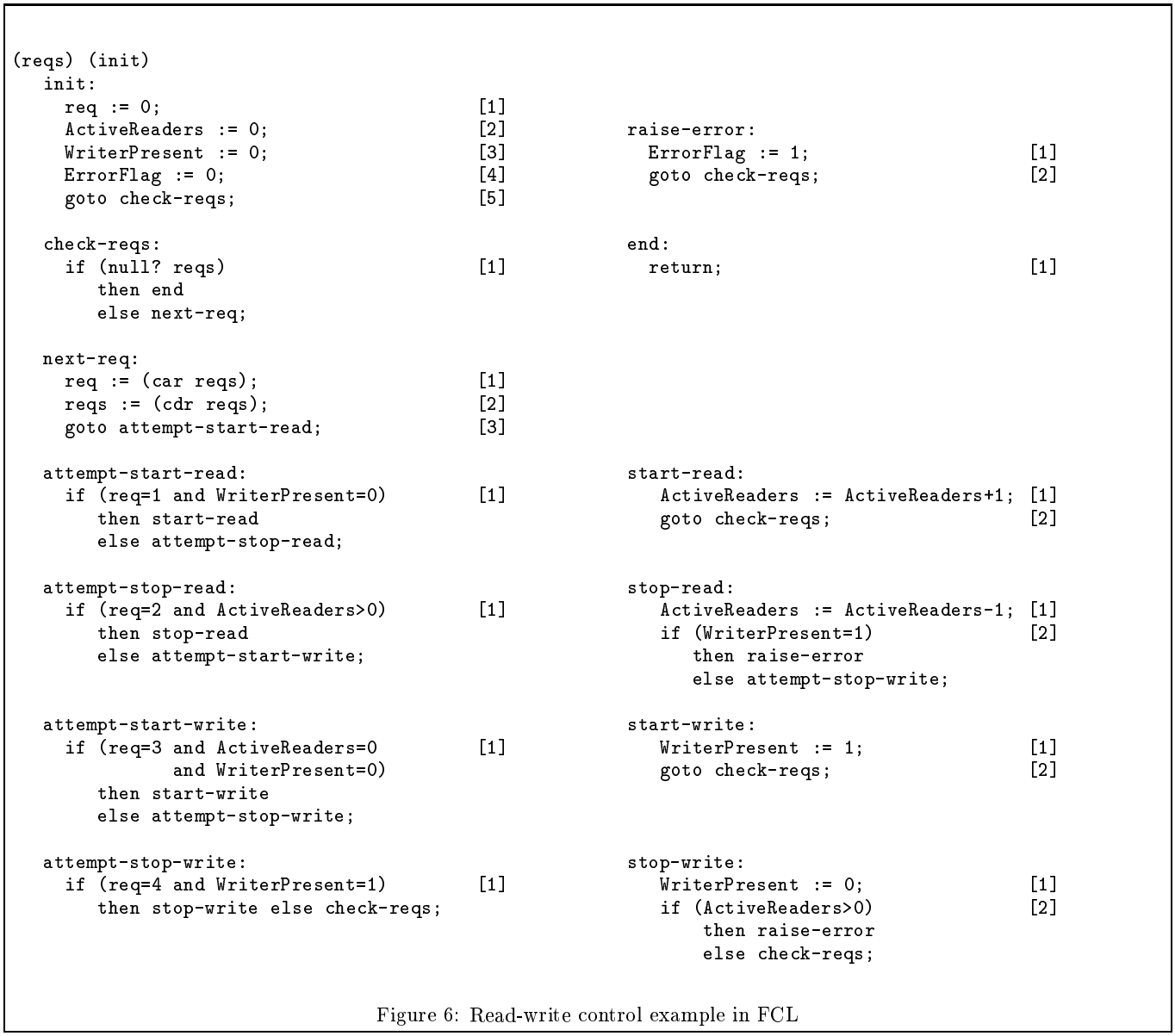

Figure 6: Read-write control example in FCL

2. identifying appropriate AI's for variables in V,
3. using control dependence information, picking out an additional set(s) of variables W that indirectly influence V and N, and
4. identifying appropriate AI's for variables in W.

Intuitively, all variables not in $V \cup W$ are irrelevant and can be abstracted with the point AI.

Clearly, item (1) can be automated by a simple pass over the LTL specification. Moreover, the information in item (3) is *exactly* the information that would be produced by slicing the program p based on a criterion generated from information in (1). Thus, pre-processing a program to be verified using slicing provides automated support for our methodology. Specifically, slicing can *(i)* identify relevant variables (which require AI's other than the point AI), *(ii)* eliminate irrelevant program variables from consideration in the abstraction selection process (they will not be present in the residual program p_s yielded by slicing), and *(iii)* reduce the size of the software and thus the size of the transition system to be analyzed. Other forms of support are needed for items (2) and (4) above.

For this approach, given a program p and a specification ψ, we desire a *criterion extraction function* extract that extracts an appropriate slicing criterion C from ψ. Slicing p with respect to C should yield a smaller residual program p_s that (a) preserves and reflects the satisfaction of ψ, and (b) has as little irrelevant information as possible.

The following requirement expresses condition (a) above.

Requirement 1 (LTL-preserving extract**)** *Given program p and a specification ψ, let $C = \mathsf{extract}(\psi)$, and let p_s the result of slicing p with respect to C. Then for any p execution trace $\pi = s_1, \ldots, s_k$,*

$$\pi \models \psi \;\; \textit{iff} \;\; \pi_s \models \psi$$

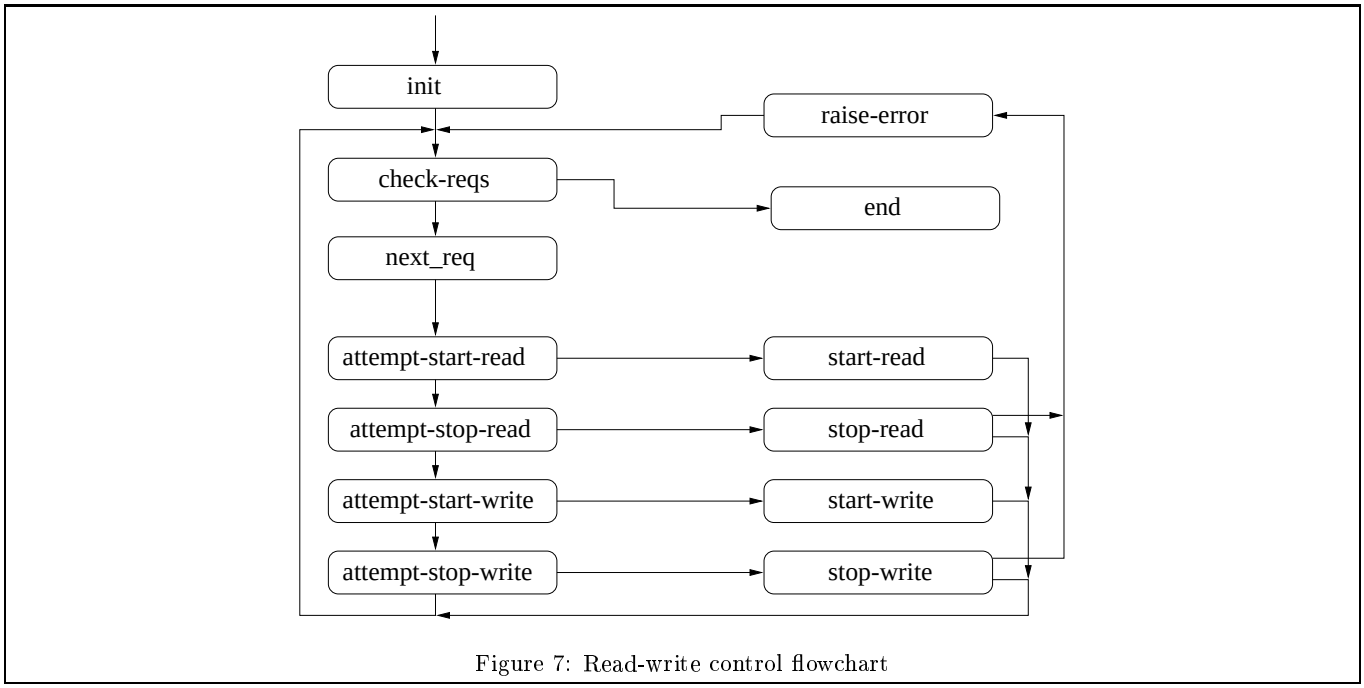

Figure 7: Read-write control flowchart

where π_s is the execution trace of p_s running with initial state s_1.

5.1 Proposition-based slicing criterion

We now consider some technical points that will guide us in defining an appropriate extraction function. As stated above, we want to preserve the satisfaction of the formula ψ yet remove as much irrelevant information from the original trace π as possible. We have already discussed the situation where certain variables' values can be eliminated from the states in a trace π because they do not influence the satifaction of the formula ψ under π. What is important in this is that we have used purely syntactic information (the set of variables mentioned in ψ) to reduce the state space.

Let's explain this reduction in more general terms. Consider a trace

$$\pi = s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n.$$

Assume that the state transition s_i, s_{i+1} does not influence the statisfaction of ψ. Formally, $\pi \models \psi$ iff $\pi_s \models \psi$ where π_s is the compressed trace (the transition s_i, s_{i+1} has been compressed)

$$\pi_s = s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n.$$

Another view of the change from π to π_s is that the action α that causes the change from s_{i-1} to s_i and the action α' that causes the change from s_i to s_{i+1} have been combined into an action α'' that moves from s_{i-1} to s_{i+1}. Intuitively, the formula ψ "doesn't need to know" about the intermediate state s_i. For example, the irrelevant transition might be an assignment to an irrelevant variable, or a transition between nodes $[l.i]$ and $[l.(i+1)]$ not mentioned in ψ.

What is the technical justification for identifying compressible transitions using a purely syntactic examination of only the propositions in a formula ψ? The answer lies in the fact that, for the temporal operators we are treating, state transitions that don't change the satisfication of the primitive propositions of the formula ψ do not influence the satisfaction of ψ itself. This means that we can justify many trace compressions by reasoning about only single transitions and satisfaction of primitive propositions. We will see below that this property does not hold when one includes other temporal operators such as the *next state* operator $\circ$.

We now formalize these notions. The following definition gives a notion of proposition invariance with respect to a particular transition.

Definition 12 (P-stuttering transition) *Let P be a primitive proposition, and let*

$$\pi = s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n.$$

The transition $s_i \rightarrow s_{i+1}$ is said to be P-stuttering *when*

$$[\![P]\!]s_i = [\![P]\!]s_{i+1}.$$

If $\mathcal{P}$ is a set of primitive propositions and for each proposition $P \in \mathcal{P}$ the transition $s_i \rightarrow s_{i+1}$ is P-stuttering, then the transition is said to be $\mathcal{P}$-stuttering.

The following lemma states that the satisfaction of a formula ψ containing primitive propositions $\mathcal{P}$ is invariant with respect to expansion and compression of $\mathcal{P}$-stuttering steps.

Lemma 1 *Let ψ be a formula and let $\mathcal{P}$ be the set of primitive propositions appearing in ψ. For all traces*

$$\pi = s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n,$$

if $s_i \rightarrow s_{i+1}$ is $\mathcal{P}$-stuttering, then

$$\pi \models \psi \quad \textit{iff} \quad \pi_s \models \psi$$

where

$$\pi_s = s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n.$$

This lemma fails when one includes the next state operator $\circ$ [23] with the following semantics

$$s_1, s_2, ... \models \circ\psi \quad \text{iff} \quad s_2, ... \models \psi.$$

For example, consider the trace

$$\pi = ([l.1], \sigma_1), ([l.2], \sigma_2), ([l.3], \sigma_3), ([l.4], \sigma_4).$$

Let P be the proposition $[l.3]$ and note $\pi \models \circ\circ P$. Now $([l.1], \sigma_1) \rightarrow ([l.2], \sigma_2)$ is P-stuttering (P is false in both states), but compressing the transition to obtain

$$\pi_s = ([l.2], \sigma_2), ([l.3], \sigma_3), ([l.4], \sigma_4).$$

does not preserve satification of the formula (*i.e.*, $\pi_s \not\models \circ\circ P$).

Intuitively, the next state operator allows one to count states, and thus any attempt to optimize by compressing transitions in this setting is problematic. For this reason, some systems like SPIN [23] do not guarantee that the semantics of $\circ$ will be preserved during, *e.g.*, partial-order reduction optimizations.

Given a formula ψ where $\mathcal{P}$ is the set of propositions in ψ, we now want to define an extraction function that guarantees that transitions that are not $\mathcal{P}$-stuttering are preserved in residual program traces.

- For variable propositions $P = [x \; rop \; c]$, observe that only definitions of the variable x may cause the variable to change value (*i.e.*, cause a transition to be non-P-stuttering). This suggests that for each proposition $[x \; rop \; c]$ in a given specification ψ, each assignment to x should be included in the residual program. Moreover, x should be considered relevant.
- For a proposition $P = [l.i]$, entering or leaving CFG node $[l.i]$ can cause the proposition to change value (*i.e.*, cause the transition to be non-P-stuttering). One might imagine that we only need the slice to include the statement $[l.i]$ for each such proposition in the formula. However, it is possible that compression might remove all intermediate nodes between two occurences of the node $[l.i]$. This, as well as similar situations, do not preserve that state changes associated with entering and exiting the node. Therefore, in addition to the node $[l.i]$, we must ensure that all immediate successors and all immediate predecessors of $[l.i]$ are included in the slice.

Based on these arguments, we define an extraction function as follows.

Definition 13 (Proposition-based extraction) *Given a program p and specification ψ, let V be the set of all program variables occurring in ψ, and let $\{n_1, \ldots, n_k\}$ be the set of all nodes that contain assignments to variables in V unioned with the set N_P of all nodes appearing in node propositions of ψ together the successors and predecessors of each node in N_P. Then* $\mathsf{extract}(\psi) \stackrel{\text{def}}{=} \{(n_1, V), \ldots, (n_k, V)\}$.

Property 1 *The extraction function* extract *satisfies Requirement 1.*

As an example,

$\mathsf{extract}(\Box([\texttt{start-read.1}] \Rightarrow [\texttt{WriterPresent=0}]))$

yields the following criterion C_1:

$$\left\{ \begin{array}{l} ([\texttt{start-read.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{attemp-start-read.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{start-read.2}], \{\texttt{WriterPresent}\}), \\ ([\texttt{init.3}], \{\texttt{WriterPresent}\}), \\ ([\texttt{start-write.1}], \{\texttt{WriterPresent}\}), \\ ([\texttt{stop-write.1}], \{\texttt{WriterPresent}\}) \end{array} \right\}.$$

Here, the first three lines of the criterion are the [start-read.1] node mentioned in the formula, along with its predecessor and successor. The last three lines are the nodes where WriterPresent is assigned a value.

Figure 8 presents the resulting slice. The slice is identical to the original program except that the variable ErrorFlag and the block raise-error disappear from the program. Thus, slicing automatically detects what our abstracting methodology yielded in the previous section: for the given specification, only ErrorFlag is irrelevant. The previous conditional jumps in stop-read and stop-write to raise-error are replaced with unconditional jumps to check-req. In this case, the slicing algorithm has detected that the nodes in the raise-error block are irrelevant, and the conditional jumps are replaced with unconditional jumps to the node where the true and false paths leading out of the conditionals meet.

As a second example, consider the specification $\psi = \Diamond[\texttt{check-reqs.1}]$ ([check-reqs.1] is eventually executed). In this case $\mathsf{extract}(\psi)$ yields the criterion C_2:

$$\left\{ \begin{array}{l} ([\texttt{check-reqs.1}], \emptyset), \\ ([\texttt{init.5}], \emptyset), \\ ([\texttt{end.1}], \emptyset), \\ ([\texttt{next-req.1}], \emptyset) \end{array} \right\}.$$

Here, the lines of the criterion are the [check-reqs.1] node mentioned in the formula, along with its predecessor and successors. Since there are no variable propositions in the specification, no variables are specified as relevant in the criterion.

Figure 9 presents the resulting slice. It is obvious that the residual program is sufficient for verifying the reachability of [check-req.1] as given in the specification. All variables are eliminated except reqs which appears in the test at [check-reqs.1]. Even though it not strictly necessary for verifying the property, this conditional is retained by the slicing algorithm since it is control-dependent upon itself. In addition, the slicing criterion dictates that the node [next-req.1] should be in the slice. However, since the assignment at this node does not assign to a relevant variable, the assignment can be replaced with **skip**. Finally, the jump to check-reqs at node [next-req.3] in the residual program is the result of chaining through a series of trivial **goto**'s during post-processing.

6 Future Work

The previous criteria have considered individual propositions. Many property specifications, however, describe states using multiple propositions or state relationships between states that are characterized by different propositions. In this section, we give some informal suggestions about how the structure of these complex specifications may be exploited to produce refined slicing criterion.

```
(reqs) (init)
init:
  req := 0;                                [1]
  ActiveReaders := 0;                      [2]
  WriterPresent := 0;                      [3]

  goto check-reqs;                         [5]

check-reqs:                                              end:
  if (null? reqs)                          [1]           return;                                  [1]
     then end
     else next-req;

next-req:
  req := (car reqs);                       [1]
  reqs := (cdr reqs);                      [2]
  goto attempt-start-read;                 [3]

attempt-start-read:                                      start-read:
  if (req=1 and WriterPresent=0)           [1]           ActiveReaders := ActiveReaders+1;        [1]
     then start-read                                     goto check-reqs;                         [2]
     else attempt-stop-read;

attempt-stop-read:                                       stop-read:
  if (req=2 and ActiveReaders>0)           [1]           ActiveReaders := ActiveReaders-1;        [1]
     then stop-read                                      goto check-reqs;                         [2]
     else attempt-start-write;


attempt-start-write:                                     start-write:
  if (req=3 and ActiveReaders=0            [1]           WriterPresent := 1;                      [1]
            and WriterPresent=0)                         goto check-reqs;                         [2]
     then start-write
     else attempt-stop-write;

attempt-stop-write:                                      stop-write:
  if (req=4 and WriterPresent=1)           [1]           WriterPresent := 0;                      [1]
     then stop-write else check-reqs;                    goto check-reqs;                         [2]
```

Figure 8: Slice of read-write control program with respect to C_1

```
(reqs) (init)
init:
  goto check-reqs;                         [5]

check-reqs:                                              end:
  if (null? reqs)                          [1]           return;                                [1]
     then end
     else next-req;

next-req:
  skip;                                    [1]
  reqs := (cdr reqs);                      [2]
  goto check-reqs;                         [3]
```

Figure 9: Slice of read-write control program with respect to C_2

```
(1)   x := 0;            x := zero;          -- not included in slice
(2)   x := x + 1;        x := pos;           x := pos;
(3)   x := -x;           x := neg;           -- not included in slice
(4)   x :=  x * x;       x := pos;           x := pos;
```

Figure 10: Slicing abstracted programs

Consider a simple conjunction of propositions appearing in an eventuality specification

$$\Diamond([\texttt{1.1}] \wedge [\texttt{x=0}]).$$

Rather than slicing on the propositions separately, we can use the semantics of $\wedge$ to refine the slicing criterion. For this property, we are not interested in all assignments to `x` but only those that can influence the value at [1.1]. Thus, our slicing criterion would be: $\texttt{extract}(\psi) \stackrel{\text{def}}{=} \{([\texttt{1.1}], x)\}$. This approach generalizes in any setting where the program point proposition occurs positively with any number of variable propositions as conjuncts.

Thus far, we have considered slicing as a prelude to ABPS. Application of ABPS can, however, reveal semantic information about variable values in statement syntax, thereby making it available for use in slicing.

Figure 10 illustrates a sequence of assignments to `x`, on the left, and the abstracted sequence assignments, in the middle, resulting from binding of the classic signs AI [1] to `x` during ABPS. In such a situation we can determine transitions in the values of propositions related to `x` (e.g., `x>0`) syntactically.

Consider a response property [15] of the form

$$\Box(\psi_1 \Rightarrow \Diamond\psi_2)$$

Our proposition slicing criterion would be based on solely on ψ_1 and ψ_2. As with the conjunctions above, we observe two facts about the structure of this formula that can be exploited.

1. Within the $\Box$ is an implication, thus we need only reason about statements that cause the value of ψ_1 to become true (since false values will guarantee that the entire formula is true).

2. Since the right-hand side of the $\Rightarrow$ is a $\Diamond$, we need only reason about the first statement, in a sequence of statements, that causes ψ_2 to become true.

The right column of Figure 10 illustrates the effect of applying observation 1 to eliminate assignments that do not cause a positive transition in $\psi_1 = x > 0$ from the sliced program. Note that if a proposition involving `x` appears in ψ_2 then the slicing criterion may be expanded to include additional statements.

In addition, a program point where ψ_1 holds which is post-dominated by a point at which ψ_2 holds need not be considered for the purpose of checking response, since the existence of this relationship implies that the response holds for this occurrence of ψ_1.

Observation 2 can be exploited using post-domination information. A program point where ψ_2 holds which is post-dominated by another point where ψ_2 holds does not need to be included in the slice. This is because only one program point at which ψ_2 holds is required on any path for the $\Diamond$ formula to become true. Thus, any post-dominated ψ_2 nodes may be eliminated.

This refined slicing criteria defined above requires the use of auxiliary information, such as post-domination information, that needs to be available prior to slicing. While the cost of gathering this information and processing it to compute slicing criteria may be non-trivial, it will be dominated by the very high cost of performing model checking on the sliced system. In most cases, the cost of reducing the the size of the system presented to the model checker will be more than offset by reduced model check time.

We have discussed two refined criteria based on structural properties of the formula being checked. Similar refinements can be defined for a number of other classes of specifications including precedence and chain properties [15]. These refinements use essentially the same information as described above for response properties; precedence properties require dominator rather than post-dominator information.

We note that the refined response criteria is applicable only when the property to be checked is of a very specific form, even slight variations in the structure of the formula may render the sliced program unsafe. A recent survey of property specification for finite-state verification showed that response and precedence properties of the form described above occur quite frequently in practice [16]; 48% of 555 real-world specifications fell into these two categories. For this reason, we believe that the effort to define a series of special cases for extracting criteria based on formula structure is justified despite its apparent narrowness.

7 Related Work

Program slicing was developed as a technique for simplifying programs for debugging and for identifying parts of programs that can execute in parallel [32]. Since its development the concept of slicing has been applied to a wide variety of problems including: program understanding, debugging, differencing, integration, and testing [31]. In these applications, it is crucial that the slice preserve the exact execution semantics of the original program with respect to the slicing criterion. In our work, we are interested only preserving the ability to successfully model check properties that are correct; this weakening allows for the refinement of slicing criteria based on the property being checked.

Slicing has been generalized to other software artifacts [30] including: attribute grammars, requirements models [22] and formal specifications [4]. Cimitile et. al. [6] use Z specifications to define slicing criteria for identifying reusable code in legacy systems. In their work, they use a combination of symbolic execution and theorem proving to process

the specifications and derive the slicing criteria. In contrast, we identify necessary conditions for sub-formula of commonly occurring patterns of specifications and use those conditions to guide safe refinement of our basic proposition slicing criteria.

Our work touches on the relationship between program specialization and slicing. We use slicing as a prelude to specialization and suggest that abstraction-based specialization may reveal semantic features in the residual program's syntax that could be used by refined slicing criteria. Reps and Turnidge [29] have studied this relationship from a different perspective. They show that while similar the techniques are not equivalent; not all slicing transformations can be achieved with specialization and vice versa.

While slicing can be viewed as a state-space reduction technique it has a number of important theoretical and practical differences from other reduction techniques appearing in the literature. State-space reduction, such as [17], preserve correctness with respect to a specific class of correctness properties. In contrast, our approach to slicing based on criteria extracted from formulae yields compressed traces that contain the state changes relevant to propositions contained in the temporal logic formula. Our approach yields programs that remain both sound and complete with respect to property checking. This is in sharp contrast to the many abstraction techniques developed in the literature (*e.g.*[7]) which sacrifice completeness for tractability. Finally, even though significant progress has been made on developing algorithms and data structures to reduce model checking times, such as OBDDs [2], those techniques should be seen as a complement to slicing. If slicing removes variables from the system that do not influence the behavior to be checked then the model checker will run faster regardless of the particular implementation techniques it employs.

8 Conclusion

We have presented a variation of program slicing for a simple imperative language. We have shown how slicing criteria can be defined that guarantee the preservation of model check semantics for LTL specifications in the sliced program. We have implemented a prototype tool that performs this slicing and experimented with a number of examples. Based on this work we are scaling up the prototype to handle significantly more complex features of programs including: structured data, treatment of procedures, and multi-threaded programs that communicate through shared data. While these extensions are non-trivial they will build of the solid base laid out in the work reported in this paper.

Acknowledgements

Thanks to James Corbett, Michael Huth, and David Schmidt for several very illuminating discussions. Thanks also to Hongjun Zheng for helpful comments on an earlier draft.

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[3] T. Cattel. Process control design using spin. In *Proceedings of the First SPIN Workshop*, October 1995.

[4] J. Chang and D.JH. Richardson. Static and dynamic specification slicing. In *Proceedings of the Fourth Irvine Software Symposium*, April 1994.

[5] A. Cimatti, F. Giunchiglia, G. Mongardi, F. Torielli, and P. Traverso. Model checking safety critical software with spin: an application to a railway interlocking system. In *Proceedings of the Third SPIN Workshop*, April 1997.

[6] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In Gianluigi Caldiera and Keith Bennett, editors, *Proceedings of the International Conference on Software Maintenance*, pages 124–133, Washington, October 1995. IEEE Computer Society Press.

[7] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[10] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[11] M.B. Dwyer, J. Hatcliff, and M. Nanda. Using partial evaluation to enable verification of concurrent software. *ACM Computing Surveys*, 30(3es), Sept, 1998.

[12] M.B. Dwyer and C. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998.

[13] M.B. Dwyer and C. Pasareanu. Model checking generic container implementations. Technical Report 98-10, Kansas State University, Department of Computing and Informatio n Sciences, 1998.

[14] M.B. Dwyer, C. Pasareanu, and J. Corbett. Translating ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Informatio n Sciences, 1998.

[15] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[16] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999. (to appear).

[17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Second Workshop on Computer Aided Verification*, pages 417–428, July 1991.

[18] C. Gomard and N.D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[19] J. Hatcliff. An introduction to partial evaluation using a simple flowchart language. In John Hatcliff, Peter Thiemann, and Torben Mogensen, editors, *Proceedings of the 1998 DIKU International Summer School on Partial Evaluation*, number (to appear) in Tutorials in Computer Science, Copenhagen, Denmark, June 1998.

[20] J. Hatcliff, M.B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *LNCS 1490*. Principles of Declarative Programming 10th International Symposium, PLILP'98, September 1998.

[21] J. Hatcliff, M.B. Dwyer, Shawn Laubach, and Nanda Muhammad. Specializing configurable systems for finite-state verification. Technical Report 98-4, Kansas State University, Department of Computing and Information Sciences, 1998.

[22] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 450–467. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

[23] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[24] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 1999. To appear.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[26] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajiani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[28] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[29] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Berlin: Springer-Verlag, 1996.

[30] A. M. Sloane and J. Holdsworth. Beyond traditional program slicing. In S. J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 180–186, New York, January 1996. ACM Press.

[31] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. Surveys the state-of-the-art in program slicing and gives many references to the literature.

[32] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[33] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.

Effective Optimisation of Multiple Traversals in Lazy Languages

Wei-Ngan CHIN Aik-Hui GOH Siau-Cheng KHOO
School of Computing
National University of Singapore

Abstract

Tupling transformation strategy can be applied to eliminate *redundant calls* in a program and also to eliminate *multiple traversals* of data structures. While the former application can produce super-linear speedup in the transformed program, the effectiveness of the latter has yet to be appreciated. In this paper, we investigate the pragmatic issues behind elimination of multiple data traversal in the context of lazy languages, and propose a framework of tupling tactic called *strictness-guided tupling*. This tactic is capable of exploiting specialised strictness contexts where possible to effect tupling optimisation. Two further enhancements of the tupling tactic are also proposed. One achieves *circular tupling* when multiple traversals from nested recursive calls are eliminated. The other exploits *speculative strictness* to further improve the performance of tupling. Benchmarks are given throughout the paper to illustrate the performance gains due to these tactics.

Keywords: *Tupling, Multiple Traversals, Strictness, Circular Programs, Speculation.*

1 Introduction

Tupling transformation strategy can be applied to eliminate redundant calls in a program and to eliminate multiple traversals of data structures. While the speed up gained from redundant calls elimination is undisputed, the effectiveness of eliminating multiple data-structure traversals has been largely ignored.

Consider the following function, `av` , which traverses a list twice:

```
av xs  = let {u = sum xs; v = len xs} in  u / v
sum xs = case xs of
         []     -> 0
         (y:ys) -> let {u = sum ys} in y + u
len xs = case xs of
         []     -> 0
         (z:zs) -> let {v = len ys} in 1 + v
```

Application of tupling to `av` returns a function that eliminates double traversal of the input list by `sum` and `len` , as follows:

```
av xs = let {(u,v) = avtup xs} in u / v
avtup xs = case xs of
           []     -> (0,0)
           (y:ys) -> let {(u,v)  = avtup ys
                          u' = y + u
                          v' = 1 + v}
                     in (u', v')
```

Even though multiple traversals are eliminated, the tupled program performs significantly *worse* than the original program, particularly under *lazy evaluation*. This is shown in the first two rows of Tab. 1.

	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
Before Tup.	24,207,372	0.02	8.58	0.21	8.81
After Tup.	60,209,100	0.04	15.31	2.14	17.49
New Tup.	16,208,668	0.05	5.63	0.07	5.75

Table 1: Measurement for executing 100 times of `av [1..10000]` under Glasgow GHC 0.29

In this paper, we propose an enhanced tupling tactic that *yields practical effectiveness*. The new tactic uses strictness information of the subject program to guide the transformation. We call it *strictness-guided tupling*. It transforms `av` function to the following `avtup'` program:

```
av xs = let! {(u,v) = avtup' xs} in u / v
avtup' xs = case xs of
            []     -> (0,0)
            (y:ys) -> let! {(u,v)  = avtup' ys
                            u' = y + u
                            v' = 1 + v}}
                      in    (u', v')
```

Notice that `let!` has been used in place of `let`. This forces the local declarations of `let!` to be evaluated strictly. It can help in two ways. Firstly, tuple components, such as `u'` and `v'`, can be evaluated strictly; alleviating the need for their closures to be built. Secondly, the tuple result itself, e.g. `(u',v')`, can be strictly evaluated and be returned via the stack (or registers), instead of being constructed in the heap. (Some strict languages, such as Moscow ML, do not presently have this

capability of returning tupled results via the stack. As a result, multiple traverals optimisation does not work properly for them.) With these optimisations, `avtup'` runs about 35% faster as shown in the last row of Tab. 1. Note that this improvement is due solely to elimination of multiple traversals. To isolate the effect of strictness optimisation, the original code used in the first row of Tab. 1 was also subjected to strictness analysis before its performance was measured.

Based on this strictness-guided tupling tactic, we investigate the use of strictness information to further improve the transformed programs, as well as to systematically generate circular programs. The key contributions of our paper are as follows:

- We provide the first pragmatic evidence on the usefulness of *tupling tactic for eliminating multiple traversals*. This task is particularly hard for lazy languages, as the extra cost incurred by tupled programs easily negate the gain from the elimination of multiple traversals. We show how meaningful gain can be achieved via a strictness-guided tupling algorithm.
- We advocate the use of an advance but yet practical strictness analysis (with support for strictness of recursive types) for our tupling tactic. While many advanced strictness analysis have been proposed in the literature[Bur91, Wad87], there have been few practical justifications for their adoption. This has led a number of researchers to believe that simple strictness is sufficient for most programs[PJP93]. We show how our *tupling tactic guided by an advanced strictness analysis can give new impetus to both techniques* - through more opportunities for optimisation.
- We highlight a novel use of tupling due to [Bir84] that results in tupled circular programs. A systematic way to incorporate *circular tupling* into our framework is proposed, providing an opportunity for Bird's technique to be automated.
- We propose a new analysis framework for *speculative strictness* and show how it could be utilised to enhance tupling.

The rest of the paper is organised as follows: Section 2 describes the language syntax; Section 3 gives an overview of basic tupling transformation. We discuss the use of strictness information during tupling, and present strictness-guided tupling in Section 4. In Section 6, we derive circular programs through tupling. This is followed by a proposal for speculative strictness to be used with conventional strictness to support more aggressive tupling (Section 7). Finally, we raise some important issues for discussion in Section 8, before concluding the paper.

2 Language

Our subject programs are written in a first-order, typed, lazy functional language (Fig. 1), similar to the intermediate languages of *practical functional compilers*. `let` statements are non-recursive. `let!` statements are used to introduce strict evaluation of abstracted expressions. `letrec` statements enable mutual-recursive definitions.

We also represent an expression `e` as $\mathcal{C}\langle \mathtt{e}_1,\ldots,\mathtt{e}_n\rangle$ to convey the idea that 'somewhere inside expression `e` lies the sub-expressions $\mathtt{e}_1, \ldots, \mathtt{e}_n$'. This allows us to look into nested sub-expressions without being too bothered by unnecessary contextual details.

The class of functions which will be subject to tupling transformation are known as *SRP functions* [Chi93]. These are functions with *single recursion parameter*. A recursion parameter has type of a recursive data structure. Argument bound to such recursion parameter are guaranteed to reduce in size when the associated function calls are unfolded. Examples of SRP functions include `len`, `sum` , `av` , as well as `rev` and `sumseg` , which will be defined in due course. For convenience, we place the recursion parameter at the first position of an SRP function, and write it as `xs` whenever possible.

3 Basic Tupling Algorithm, an Overview

The basic tupling algorithm proposed in [Chi93] can be used to eliminate both multiple traversals and redundant calls for strict languages. The algorithm is constructed based on the fold/unfold rules of [BD77]. Prior to transformation, the system determines a class of SRP functions, *SRPSet*, whose calls are to be gathered. In Fig. 2, we highlight the main components of the algorithm by listing five syntax-directed rules, B1, ..., B5.

Rules B1 and B2 merely skip over outer `let` and `case` expressions to search inside their sub-expressions for set of calls which could be tupled. The main rules of tupling are:

- B3 - To float out inner `let` abstraction, so that calls located in the `let` body with locally-defined variables can be collected for tupling. We forbid floating of `let` abstraction which contains calls to functions in *SRPSet*, so that the transformer does not miss the opportunity for collecting those calls. Implicitly, an expression can only be floated within the the binding scope of all of its free variables, and variables are renamed whenever necessary to avoid name clash.

  We allow liberal application of floating. Arbitrary floating of `let` abstraction may lead to unnecessary closures being built. Although we do not do it for the code presented in this paper, we can apply the `let` "float-in" technique of [PJPS96] to post-process the code after tupling.
- B4 - To gather multiple calls with common recursion arguments together to form a new tuple function, followed by unfolding of the calls. Gathering of calls provides an opportunity for redundant calls to be shared, and also facilitates rule B5.

$$
\begin{array}{lcl}
e & ::= & k \mid v \mid f\ (v_1, \ldots, v_n) \mid op\ (v_1, \ldots, v_n) \mid c\ (v_1, \ldots, v_n) \mid \texttt{case}\ e_0\ \texttt{of}\ \{c_i\ v_{i1} \ldots v_{in} \rightarrow e_i\}_i \\
 & & \mid\ \texttt{let}\ \{(v_{1i}, \ldots, v_{ni}) = e_i\}_{i \in M}\ \texttt{in}\ e \mid\ \texttt{let!}\ \{(v_{1i}, \ldots, v_{ni}) = e_i\}_{i \in M}\ \texttt{in}\ e \\
 & & \mid\ \texttt{letrec}\{(v_{1i}, \ldots, v_{ni}) = e_i\}_{i \in M}\ \texttt{in}\ e \\
p & \in & \textit{Program} \\
 & & p\ ::= \{\ f_i\ (v_{i1}, \ldots, v_{ik})\ =\ e_i\}_i
\end{array}
$$

Context Notation :

$$
\begin{array}{lcl}
\mathcal{C}\langle\rangle & = & \#m \mid k \mid v \mid \texttt{f}\ (\mathcal{C}\langle\rangle_1, \ldots, \mathcal{C}\langle\rangle_n) \mid \texttt{op}\ (\mathcal{C}\langle\rangle_1, \ldots, \mathcal{C}\langle\rangle_n) \\
 & \mid & \texttt{c}\ (\mathcal{C}\langle\rangle_1, \ldots, \mathcal{C}\langle\rangle_n) \mid\ \texttt{let}\ \{v_i = \mathcal{C}\langle\rangle_i\}_{i \in M}\ \texttt{in}\ \mathcal{C}\langle\rangle \\
 & \mid & \texttt{case}\ \mathcal{C}\langle\rangle_0\ \texttt{of}\ \{c_i\ v_{i1} \ldots v_{in} \rightarrow \mathcal{C}\langle\rangle_i\}_i \\
 & & \text{where \# is a special variable known as a hole, labelled with a number } m.
\end{array}
$$

Figure 1: First-order, typed, lazy functional language

B1 (Skip outer `let`)

$\mathcal{T}_B\ ds\ [\texttt{let}\ \{(v_1, \ldots, v_n) = (t_1, \ldots, t_n)\}\ \texttt{in}\ t] \Rightarrow \texttt{let}\ \{(v_1, \ldots, v_n) = (t'_1, \ldots, t'_n)\}\ \texttt{in}\ \mathcal{T}_B\ ds\ [t]$
where $t'_i = \mathcal{T}_B\ ds\ [t_i] \quad \forall i \in 1..n$

B2 (Skip outer `case`)

$\mathcal{T}_B\ ds\ [\texttt{case}\ t\ \texttt{of}\ \{\ p_i\ \texttt{->}\ t_i\ \}_{i \in N}\ ] \Rightarrow \texttt{case}\ \mathcal{T}_B\ ds\ [\ t\ ]\ \texttt{of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}_B\ ds\ [\ t_i\ ]\ \}_{i \in N}$

B3 (Float out inner `let`)

$\mathcal{T}_B\ ds\ [\ \mathcal{C}\langle\texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ t_1\rangle\ ] \Rightarrow \texttt{let}\ \{\ v = t\ \}\ \texttt{in}\ \mathcal{T}_B\ ds\ [\ \mathcal{C}\langle t_1\rangle\ ]$
where t *contains no calls to functions in SRPSet*

B4 (Gather calls)

$\mathcal{T}_B\ ds\ [\ \mathcal{C}\langle f_1(xs,\vec{t_1}), \ldots, f_n(xs,\vec{t_n})\rangle\ ] \Rightarrow$
if $n > 1$ *then* $\texttt{let}\ \{(u_1,\ldots,u_n) = \texttt{ftup}(\texttt{xs},\vec{v})\}\ \texttt{in}\ \mathcal{T}_B\ ds'\ [\ \mathcal{C}\langle\ u_1, \ldots, u_n\ \rangle\ ]$
where $ds' = ds \cup \{\ (\texttt{ftup}\ (\texttt{xs},\vec{v}), (f_1(xs,\vec{t_1}), \ldots, f_n(xs,\vec{t_n}))) \mid \vec{v} = freevar[\vec{t_1},\ldots,\vec{t_n}]\ \}$
$\texttt{ftup}\ (\texttt{xs},\vec{v}) = \mathcal{T}'_B\ ds'\ [(tf_1[\vec{t_1}/\vec{v_1}],\ldots,tf_n[\vec{t_n}/\vec{v_n}])\ ]$
$\forall i \in 1 \cdots n\ .\ tf_i$ *is the RHS of function* f_i.

B5 (Tuple `case`)

$\mathcal{T}'_B\ ds\ [(\texttt{case xs of}\ \{p_i\ \texttt{->}\ t1_i\}_{i \in N}, \ldots, \texttt{case xs of}\ \{p_i\ \texttt{->}\ tn_i\}_{i \in N})] \Rightarrow$
$\texttt{case xs of}\ \{\ p_i\ \texttt{->}\ \mathcal{T}_B\ ds\ [\ (t1_i, \ldots, tn_i)\ ]\ \}_{i \in N}$

Figure 2: Basic Tupling Algorithm

Calls gathered are recorded in the set *ds*, which acts as a pool of memoised points for terminating tuple transformation.

- B5 - To combine multiple case constructs (testing on the same recursion argument) into a single case construct. This eliminates multiple traversals over the common recursion argument.

Termination of the tupling transformation is facilitated by the well-known folding operation. Thus, instead of defining a new tuple function at B4, previously-defined tuple function is used whenever each tuple of calls gathered is identical (modulo variable renaming and ordering of the calls) to an earlier tuple.

Let us consider the effect of basic tupling algorithm on a typical function with multiple traversals shown in Fig. 3a. Inside the definition of `f`, there are two functions, `g` and `h` that traverse some common recursive structure. Generally speaking, their recursive calls occur in nested function applications and so we have to abstract out these calls by let-expressions. The basic tupling algorithm will transform the above functions to the form in Fig. 3b, where closures are made explicit through the `let` construct.

Closures built in the programs are identified in Fig. 3. Those that exist before *and* after tupling are linked by a solid line. Notice that more closures are built after tupling.

With this view of cost allocation, the extra cost incurred from tupling are: building one extra closure for the `n`-tuple and `n` extra closures for the *tuple components*. We shall refer to the first kind of closure as *tuple-closure*, and the second kind as *component-closure*.

4 Strictness-Guided Tupling

Naive elimination of multiple traversals may cause performance to degrade for lazy programs. This is due to the extra closures created by our transformation in an attempt to adhere to the lazy semantics of the subject language. Fortunately, our programs are often inherently stricter. Exploiting such strictness properties may help avoid some of these extra closures. For example, consider the `avtup(xs)` call in the RHS of `av` in Sec. 1. Both components of `avtup(xs)` are strictly needed, and so are every recursive call to `avtup`. The former requirement can help eliminate closures for the tuple-components, while the latter suggests that the tu-

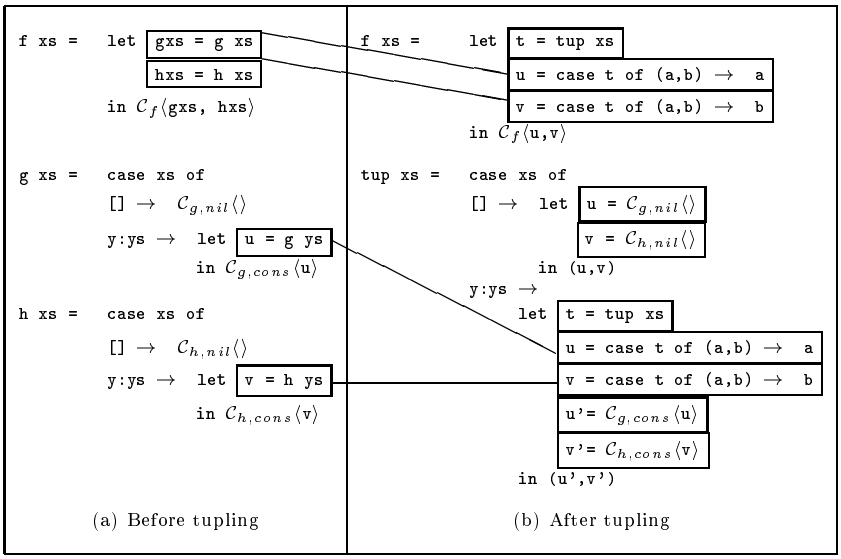

Figure 3: Closure Allocation (A box indicates a closure)

ple itself can be strictly evaluated with its result returned via the stack (or registers).

4.1 A Suitable Strictness Analysis

To implement the suggested enhancement, we require an appropriate strictness analyser. Here, we propose to use a strictness domain based on the 4-point strictness analysis first introduced by Wadler [Wad87] for recursive list-type objects. The four strictness values required are:

- `?` *Don't Know* - Expression may not be evaluated;
- `!` *Simple strictness* - Expression is evaluated to its head normal form;
- `$` *Tail-strictness* - Recursively evaluates all recursive components of the expression;
- $\star$ *Total-strictness* - All recursive components are evaluated to tail-strict form, while the non-recursive component are evaluated to head normal form.

Note the partial ordering of strictness values based on information containment: $\star \sqsubseteq$ `$` $\sqsubseteq$ `!` $\sqsubseteq$ `?`.

Given an expression, e, and a strictness value s, we write $e ::_{\mathcal{S}} s \vdash \theta$ to mean e be used in the strictness context expressed by s to infer strictness environment θ which captures the strictness of all free/bound variables in e.

The 4-point strictness domain was originally introduced for the list-type but it is straightforward to extend it to arbitrary recursive types - with `$` to denote that recursive spines of the expression will be evaluated, and $\star$ implies evaluating the non-recursive components to head normal form, in addition to evaluating the recursive spines. Based on this domain, functions `sum` and `len` of Sec. 1 could be annotated with the following strictness rules:

```
sum ::S  ⋆ → !
len ::S  $ → !
```

These strictness rules state that, when the result of evaluation is required in head normal form, `len` is tail-strict (`$`) on its list-input, while `sum` is totally-strict ($\star$) on its input. Furthermore, for functions which may be invoked at different strictness contexts, we associate a set of strictness rules to it, such as the function `rev` defined below:

```
rev ::S  {$ × ? → !, $ × $→ $, ⋆ × ⋆→ ⋆}
rev(xs,ws) = case xs of {[] -> ws ;
                         x:xs' -> rev(xs',x:ws)}
```

For example, the first rule of `rev` is applicable under simple strictness context, while the second and third rules are applicable under tail-strict and totally-strict contexts, respectively.[1]

For accuracy, our chosen strictness domain also incorporates *disjunctive* strictness information, similar to that proposed by Jensen in [Jen97]. Consider:

```
f(xs,y,z) = case xs of {[] -> y ; x:xs' -> z}
```

To capture the strictness of this function `f` more accurately, we require:

[1] By default, no argument will get evaluated when the function result is not needed. That is, $f ::_{\mathcal{S}}$ `?`$\rightarrow$ `?` for all function f.

`f` $::_{\mathcal{S}}$ `(!,!,?)` $\vee$ `(!,?,!)` $\rightarrow$ `!`

This indicates that the first argument will be evaluated to head normal form, and so will either the second or the third argument, but not both.

Though rarely used in practice, such strictness analyses are not new. Many papers have been written about similarly advanced strictness analyses, often more sophisticated than the version presented here. Rules for analysing their strictness properties are also quite standard, and can be found in [Wad87] and [Jen97].

4.2 Strictness-Guided Tupling

We now look at how the proposed strictness annotations can help guide our tupling algorithm. Fig. 4 gives the main rules for our enhanced tupling algorithm.

We first show the conditions under which creation of extra closures may be avoided during tupling. Next, we look at the propagation of strictness information by our strictness-guided tupling algorithm.

Consider a tupled-function call below with three components which may have been obtained from a call-gathering step (S5).

`let (u1,u2,u3) =` (`ftup(xs,t)` $::_{\mathcal{S}}$ s) `in` $\mathcal{C}\langle$`u1,u2,u3`$\rangle$

Without strictness information, we can only assume that all three tuple variables `u1, u2, u3` may not be needed during execution. Hence, the call `ftup(xs,t)` must be created as a tuple-closure. However, if strictness information s is available we may be able to decide if the `ftup(xs,t)` could be evaluated strictly. In fact, we can evaluate this tuple-call strictly if at least one of `u1,u2,u3` is known to be evaluated to head normal form.

In order words, if $s \sqsubseteq$ (`!,?,?`) $\vee$ (`?,!,?`) $\vee$ (`?,?,!`), we can convert the above lazy `let` to a strict `let!`. For convenience, we write $\mathcal{M}_n$ as an abbreviation of the strictness context $\vee_{i=1}^{n}\{(s_1,\ldots,s_n) \mid s_i = \texttt{!} \wedge s_j = \texttt{?}, \forall j \neq i\}$. The latter implies that at least one component of its associated tuple will be needed.

Our rule to *avoid constructing tuple-closure* can now be expressed as follows, which is an intuitive interpretation of rule S5 in Fig. 4.

if $s \sqsubseteq \mathcal{M}_n \wedge$ `ftup(xs,t)` $::_{\mathcal{S}}$ s then
`let {(u1,..,un) = ftup(xs,t)} in` $\mathcal{C}\langle$`u1,u2,u3`$\rangle$ $\Rightarrow$
`let! {(u1,..,un) = ftup(xs,t)} in` $\mathcal{C}\langle$`u1,u2,u3`$\rangle$.

Furthermore, when a tuple component is expected to be strictly evaluated, we can avoid building closure for that component. Consider the tuple (e_1 $::_{\mathcal{S}}$ s_1, ..., e_n $::_{\mathcal{S}}$ s_n). If it is determined that $s_i \sqsubseteq$ `!`, then we can evaluate e_i strictly, provided it is neither a variable nor a constant. (Variables/constants do not result in new closures and hence need not be strictly evaluated). A general rule to *avoid building component-closure* is as follows, which corresponds to rule S6 in Fig. 4:

if $s_i \sqsubseteq$ `!` $\wedge$ e_i $::_{\mathcal{S}}$ s_i then
(e_1, ..., e_i, ..., e_n) $\Rightarrow$
`let!` $\{v_i = e_i\}$ `in` (e_1, ... ,v_i, ..., e_n).

These two strictness optimisations are keys to effective tupling for lazy languages. To enable these two optimisations, our tupling algorithm must aggressively propagate strictness information during tupling. Where possible, it should also use the best strictness context available for a given tuple of abstracted calls; and aggressively propagate this strictness context into subexpressions, where feasible. This is expressed in the other rules of the algorithm, where attempts are made to propagate the strictness context to the subexpressions during tupling. Rule S1 is similar to rule B1 of the basic tupling algorithm. Rules S2 reduces `case` expression when possible. Rule S3 allows each branch of the lifted `case` to be exposed to the propagated strictness context. Lastly, rules S4 and S7 are similar to rule B3 and B5 in the basic algorithm.

5 An Example

In this section, we provide a detailed example of tupling application. Consider the function `sumseg` :

```
sumseg(xs,n) =
  case xs of
  []     -> 0
  (y:ys) -> case (n==0) of
              True  -> 0
              False -> let { n' = n - 1 }
                       in  y + sumseg(ys,n')
```

If we apply tupling to `sumseg(xs,n)+sumseg(xs,m)` , we will gather these two calls under strictness context (!$\times$!). Transformation guided by this strictness context will be as follows (several arguments of the transformation are omitted for clarity):

$\mathcal{T}$ (!$\times$!) [`sumseg(xs,n)+sumseg(xs,m)`]
(*S5*)$\Rightarrow$ `let! (u,v) = ftup(xs,n,m)` $::_{\mathcal{S}}$ (!$\times$!) `in u+v`
Define `ftup(xs,n,m)` $::_{\mathcal{S}}$ (!$\times$!) =
$\mathcal{T}'$ (!$\times$!) [(`case xs of {[] -> 0; (y:ys) ->` $\mathcal{C}\langle n\rangle$},
`case xs of {[] -> 0; (y:ys) ->` $\mathcal{C}\langle m\rangle$})]
where $\mathcal{C}\langle a\rangle$ = `case (`a`== 0) of`
`True -> 0 ;`
`False -> let { n' =` a`-1}`
`in y + sumseg(ys,n')`
(*S7*)$\Rightarrow$ `case xs of`
`[] -> (0,0)`
`y:ys ->` $\mathcal{T}$ (!$\times$!)[($\mathcal{C}\langle n\rangle$, $\mathcal{C}\langle m\rangle$)]

Though the case-test has been lifted from the components, there are still two inner case constructs, denoted by $\mathcal{C}\langle n\rangle$ and $\mathcal{C}\langle m\rangle$, which actually make the recursive calls to `sumseg` occur in lazy context (wrt the RHS of `sumseg`). However, as the strictness information (!$\times$!) is propagated to the subexpressions during transformation, both $\mathcal{C}\langle n\rangle$ and $\mathcal{C}\langle m\rangle$ lie in strict context, and this enable us to transform ($\mathcal{C}\langle n\rangle$,$\mathcal{C}\langle m\rangle$) further. We apply S3 twice to lift two `case` expressions, and then apply S4 several times on each of the branches of the `case` expressions:

S1 (Skip outer `let`)

$\mathcal{T}\ s_0\ ds\ [$`let` $\{(v_1,\ldots,v_n) = (t_1,\ldots,t_n)\}$ `in` $t] \Rightarrow$ `let` $\{(v_1,\ldots,v_n) = (t'_1,\ldots,t'_n)\}$ `in` $\mathcal{T}\ s\ ds\ [t]$
where $t ::_{\mathcal{S}} s_0 \vdash \theta$
$t'_i = \mathcal{T}\ (\theta[v_i])\ ds\ [t_i] \quad \forall i \in M$

S2 (Reduce `case`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle$ `case` $c_j(\vec{t})$ `of` $\{\ c_i(\vec{v})$ `->` $t_i\ \}_{i\in M}\ \rangle\ ] \Rightarrow \mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle\ t_j[\vec{t}/\vec{v}]\ \rangle\ ]$

S3 (Lift `case`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle$`case` t `of` $\{\ p_i$ `->` $t_i\ \}_{i\in M}\rangle] \Rightarrow$
if $(\theta[u]) \sqsubseteq$ `!` *then* `case` $(\mathcal{T}\ (\theta[u])\ ds\ [t])$ `of` $\{\ p_i$ `->` $\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t'_i[p_i/t]\rangle\ ]\ \}_{i\in M}$
where $\mathcal{C}\langle$`case` u `of` $\{\ p_i$ `->` $t_i\ \}_{i\in M}\rangle ::_{\mathcal{S}} s_0 \vdash \theta$ *for new variable* u

S4 (Float out inner `let`)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle$`let` $\{\ v = t\ \}$ `in` $t_1\rangle\ ] \Rightarrow$ `let` $\{\ v = t\ \}$ `in` $\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t_1\rangle\ ]$
where t *contains no calls to functions in SRPSet*

S5 (Gather calls)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n})\rangle] \Rightarrow$
if $s \sqsubseteq \mathcal{M}_n \wedge n > 1$ *then* `let!` $\{(u_1,\ldots,u_n) = ftup(xs,\vec{v})\}$ `in` $\mathcal{T}\ s_0\ ds'\ [\mathcal{C}\langle u_1,\ldots,u_n\rangle\ ]$
where $\mathcal{C}\langle u_1,\ldots,u_n\rangle ::_{\mathcal{S}} s_0 \vdash \theta$ *for new variables* $u_1,\ldots,u_n$
$s = \theta[(u_1,\ldots,u_n)]$
$ds' = ds \cup \{\ (ftup(xs,\vec{v}),\ s,(f_1(xs,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n}))) \mid \vec{v} = freevar[\vec{t_1},\ldots,\ \vec{t_n}]\ \}$
$ftup(xs,\vec{v}) = \mathcal{T}'\ (s \sqcap \mathcal{M}_n)\ ds'\ [\ (tf_1[\vec{t_1}/\vec{v_1}],\ldots,tf_n[\vec{t_n}/\vec{v_n}])\ ]$
$\forall i \in 1\cdots n\ .\ tf_i$ *is the RHS of function* f_i.

S6 (Make tuple components strict)

$\mathcal{T}\ s_0\ ds\ [\ (t_1 ::_{\mathcal{S}} s_1,\ldots,t_n ::_{\mathcal{S}} s_n) ::_{\mathcal{S}} s_0] \Rightarrow$ `let!` $\{\ v_i = t_i\ \}_{i\in N}$ `in` $\mathcal{T}\ s_0\ ds\ (t'_1,\ldots,t'_n)$
where $t'_i =$ *if* $i \in N$ *then* v_i *else* t_i
$N \equiv \{j \mid j \in 1\cdots n, s_j \sqsubseteq !, not(isVariable?(t_j)\}, not(isConstant?(t_j)\}.$

S7 (Tuple `case`)

$\mathcal{T}'\ s_0\ ds\ [($`case xs of` $\{p_i$ `->` $t1_i\}_{i\in M},\ \ldots,$ `case xs of` $\{p_i$ `->` $tn_i\}_{i\in M})\ ] \Rightarrow$
`case xs of` $\{p_i$ `->` $\mathcal{T}\ s_0\ ds\ [\ (t1_i,\ldots,tn_i)]\ \}_{i\in M}$

Figure 4: Strictness-Guided Tupling Algorithm

```
T(!×!) [ (C⟨n⟩,C⟨m⟩) ]     (S3,S4)⇒
   case (n==0) of
   True -> case (m==0) of
           True -> T(!×!) [ (0,0) ]
           False -> let {m' = m - 1}
                    in  T(!×!)[(0,y+sumseg(ys,m'))]
   False -> case (m==0) of
            True -> let {n' = n - 1}
                    in  T(!×!)[(y+sumseg(ys,n'),0)]
            False -> let {n' = n - 1
                          m' = m - 1}
                     in T(!×!)[(y+sumseg(ys,n'),
                                y+sumseg(ys,m'))].
```

Consider the four branches in the code. The first branch does not have any recursive call, while the second and third branches have only one recursive call each. Hence, call-gathering need not be invoked. The last branch contains two recursive calls which now lie in strict context for our tupling transformer to gather. As this specialised context is identical to the previous tuple definition of `ftup`, the algorithm performs a fold operation to end the recursive transformation. Finally, as both the tuple-call and their components lie in their respective strict contexts, we can apply rules S5 and S6 to yield the following:

```
T(!×!)[(y+sumseg(ys,n'),y+sumseg(ys,m'))]
(S5,S6)⇒ let! {(u,v) = ftup(ys,n',m')::S (!×!)
                u' = y + u
                v' = y + v }
         in    (u', v')
```

The final transformed code is as follows:

```
ftup(xs,n,m)::S (!×!) =
case xs of
[] -> (0,0)
(y:ys') ->
 case (n==0) of
 True -> case (m==0) of
         True -> (0,0)
         False -> let m' = m - 1 in
                  let! v = y+sumseg(ys,m')
                  in   (0, v)
 False -> case (m==0) of
          True -> let n' = n - 1 in
                  let! u = y+sumseg(ys,n')
                  in   (u, 0)
          False ->
            let {n' = n - 1
                 m' = m - 1}
            in   let! {(u,v)=ftup(ys,n',m')::S (!×!)
                       u' = y + u
                       v' = y + v}
                 in    (u', v')
```

Tab. 2 shows the run-time improvement of the transformed program.

At this point, we would like to justify our decision to allow strictness annotations to guide tupling.

One may wonder if it might be simpler to just apply strictness optimisation after basic tupling? The following is the result of transforming the same expression with basic tupling tactic:

```
ftup1(xs,n,m) =
   case xs of
   [] -> (0,0)
   (y:ys) ->
```

1000 times of `sumseg` with `xs = [0..10000]`					
	Heap (bytes)	Time(s)			
		INIT	MUT	GC	Total
`sumseg(xs,900)+sumseg(xs,900)`					
No Tupling	72,071,280	0.04	54.66	0.13	54.83
Basic Tupling	140,608,348	0.01	74.09	1.47	75.57
New Tupling	43,283,450	0.01	41.80	0.04	41.85
`sumseg(xs,250)+sumseg(xs,750)`					
No Tupling	40,068,280	0.03	30.29	0.03	30.35
Basic Tupling	117,204,988	0.02	50.23	0.65	50.90
New Tupling	32,088,520	0.01	27.04	0.05	27.10

Table 2: Execution times of `sumseg`

```
let {n' = n - 1
     m' = m - 1
     (u,v) = ftup1(ys,n',m') } in
case (n==0) of
True -> case (m==0) of
        True -> (0,0)
        False -> (0, y + v)
False -> case (m==0) of
         True -> (y + u, 0)
         False -> (y + u, y + v)
```

The above code is less efficient than the code produced by strictness-guided tupling: it will invoke calls to `ftup1` throughout, resulting in the creation of unnecessary tuple-closures as well as component-closures. Because of the lack of strictness information, the basic tupling algorithm was not able to delve into various branches of the `case` expressions, and can only gather similar calls at a global level. At that point, strictness analysis alone is unable to recover our earlier level of optimisation. Several other steps are also needed, including the inverse of tupling. A more sophisticated strictness analysis over the tupled program would have to be applied, followed by float-in of tupled-calls into `case` branches to exploit better strictness. This may not be sufficient, as some of the tupled calls (e.g. in second and third branches of `sumseg`) might have to be "untupled" to achieve better performance. Our proposal to use strictness-guided tupling is therefore simpler, as it introduces tupled-functions only when tuple-closures can be eliminated. This provides some guarantee on the performance of each such tupled-functions.

One may also wonder if it is better to apply strictness optimisation prior to basic tupling? This approach may be helpful to the extent that it could help compile away the `$`- and $\star$-strictness annotations into our code. However, strictness propagation for each tuple of calls gathered is still required during tupling. Without it, some opportunities for eliminating closures for tuple-results and their components would be lost. For example, when gathering two calls c_1 and c_2 under strictness contexts s_1, s_2, we should propagate the strictness contexts s_1, s_2 during tupling. Failure to do so may result in less closures being eliminated.

6 Circular Tupling

A particularly elegant use of tupling was proposed by Richard Bird [Bir84] where he showed how circular tupled programs could be used to eliminate multiple traversals of nested function calls. A classic example is the palindrome function:

```
pal(xs)    = eq(xs,rev(xs,[]))
eq(xs,ys)  = case xs of { [] -> (ys==[]);
   x:xs' -> case ys of { [] -> False;
            y:ys' -> (x==y) and eq(xs',ys')} }
rev(xs,ws) = case xs of { [] -> ws;
                          x:xs' -> rev(xs',x:ws) }
```

Here, we have two recursive calls, namely `eq(xs,_)` and `rev(xs,[])`. However, `rev` call is nested within `eq` call. These two calls separately traverse the same data structure `xs`. Bird showed how such nested expressions can be manually transformed to tupled circular functions. In this section, we shall examine how circular tupling could be systematically handled by our enhanced tupling algorithm. A key step is to introduce a recursive `letrec` construct with a circular variable to unnest the inner call. For the palindrome example, this step results in:

```
pal(xs) = letrec (u0,u1)=re_tup(xs,[],u0 ::F?) ::S(!×!)
          in u1
```

Define:

```
re_tup(xs,ws,u0)
        = (rev(xs,ws),eq(xs,u0 ::F?)) ::S(!×!)
```

The two recursive calls were found in strict contexts - motivating our use of strictness annotation `(!×!)` for the gathered calls. Also, as the variable `u0` is circular, it must not be evaluated strictly; otherwise our program will chase after a result that has not been created yet. Hence, during the introduction of `letrec`, circular variables must be placed in lazy context. To achieve this, we introduce a new annotation $e ::_{\mathcal{F}} s$ whose purpose is to *force* subterm e to a stated strictness s.

To incorporate this step into our tupling method, we require a special rule, *S9*, shown in Fig. 5. Note that a nested inner call $f_0(xs,\vec{t_0})$ is abstracted via a circular variable. Applying our enhanced tupling algorithm to the above example yields:

```
re_tup(xs,ws,u0) ::S(!×!)
  = case xs of { [] -> (ws,u0 ::F?==[]);
    x:xs' -> let!  (u,v)=re_tup(xs',x:ws,tl(u0)) ::S(!×?)
    in (u, case u0 ::F? of { [] -> False;
                             y:ys' -> (x==y) and v} }
re_tup(xs,ws,u0) ::S(!×?)
  = case xs of { [] -> (ws,u0 ::F?==[]);
    x:xs' -> let!  (u,v)=re_tup(xs',x:ws,tl(u0)) ::S(!×?)
    in (u, case u0 ::F? of { [] -> False;
                             y:ys' -> (x==y) and v} }
tl(xs) = case xs of {x:xs' -> xs'}
hd(xs) = case xs of {x:xs' -> x}
```

S8 (Lift `case` - Speculative)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle$`case` t `of { } `p_i `->` t_i `}`$_{i\in M}\rangle]$ $\Rightarrow$

if $s \sqsubseteq$ `!` *then* `case` $(\mathcal{T}\ s\ ds\ [t])$ `of { } `p_i `->` $\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle t'_i[p_i/t]\rangle\ ]$ `}`$_{i\in M}$

where $\theta_0 \vdash t ::_{\mathcal{O}} s$ *(NB:* θ_0 *is the strictness environment of its current context.)*

S9 (Circular Tupling)

$\mathcal{T}\ s_0\ ds\ [\ \mathcal{C}\langle f_1(xs,f_0(xs,\vec{t_0}),\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n})\rangle]$ $\Rightarrow$

`letrec` $\{(u_0,\ldots,u_n) = ftup(xs,u_0 ::_{\mathcal{F}} ?,\vec{v})\}$ `in` $\mathcal{T}\ s_0\ ds'\ [\mathcal{C}\langle u_1,\ldots,u_n\rangle\ ]$

where (`let` $u_1 = f_1(xs,u_0,\vec{t_1})$ `in` $\mathcal{C}\langle u_1,\ldots,u_n\rangle) ::_{\mathcal{S}} s_0 \vdash \theta$ *for new variables* $u_0,\ldots,u_n$

$s = \theta[(u_0,\ldots,u_n)]$

$ds' = ds \cup \{\ (ftup(xs,u_0,\vec{v}),\ s,(f_0(xs,\vec{t_0}),\ f_1(xs,u_0,\vec{t_1}),\ \ldots,\ f_n(xs,\vec{t_n}))) \mid \vec{v} = freevar[\vec{t_0},\ldots,\vec{t_n}]\ \}$

$ftup(xs,\vec{v}) = \mathcal{T}'\ (s \sqcap \mathcal{M}_n)\ ds'\ [\ (tf_0[\vec{t_0}/\vec{v_0}],tf_1[u_0/v_0,\vec{t_1}/\vec{v_1}],\ldots,tf_n[\vec{t_n}/\vec{v_n}])\ ]$

$\forall i \in 0\cdots n.$ tf_i *is the RHS of function* f_i.

S10 (Make tuple components strict - Speculative)

$\mathcal{T}\ s_0\ ds\ [\ (t_1\ ::_{\mathcal{O}}\ s_1,\ \ldots,\ t_n\ ::_{\mathcal{O}}\ s_n)]$ $\Rightarrow$ `let!` $\{\ v_i = t_i\ \}_{i\in N}$ `in` $\mathcal{T}\ s_0\ ds\ [\ (t'_1,\ldots,t'_n)\ ]$

where t'_i $= if\ i \in N\ then\ v_i\ else\ t_i$

$N \equiv \{j \mid j \in 1\cdots n, s_j \sqsubseteq !, not(isVariable?(t_j)), not(isConstant?(t_j))\}$.

Figure 5: Extra Rules for Enhanced Tupling Algorithm

Two `re_tup` definitions were introduced under strictness contexts $(!\times!)$ and $(!\times?)$, respectively. However, as both definitions are syntactically identical we could combine them into a single definition to reduce code duplication. Also, as our tupling algorithm is strictness-guided, it is able to detect that tuple-closures were unnecessary for the recursive `re_tup` calls. In addition, the first component of `re_tup` calls can be strictly evaluated, but not the second component. The lazy annotation on `u0` forces closures to be built for the second component, despite the fact that the original `eq` call was lying in a strict context. This is in contrast to [Bir84] which requires a manual intervention (before tupling) to redefine `eq` to make its second parameter 'lazy'.

Another interesting example is the program to replace all tips of a given tree by its minimum value.

```
data Tree(a) = Leaf(a)  | Node(Tree(a),Tree(a))
mintip(t) = repl(t,mint(t))
repl(t,m) = case t of { Leaf(a) -> Leaf(m);
            Node(l,r) -> Node(repl(l,m),repl(r,m)) }
mint(t)   = case t of { Leaf(a) -> a;
            Node(l,r) -> min2(mint(l),mint(r)) }
min2(x,y) = case (x < y) of { True -> x; False -> y }
```

As `mintip` returns a tree structure, it may be evaluated under different strictness contexts. If we use a head-strict `!` context for `mintip` and apply basic tupling algorithm, we obtain:

```
mintip(t) ::S != letrec (u,v)=rm_tup(t,v) ::S (!×?) in u
rm_tup(t,m) ::S (!×?)
   = case t of { Leaf(a) -> (Leaf(m),a);
     Node(l,r) -> let (u,v)=rm_tup(l,m) ::S (?×?) in
                  let (y,z)=rm_tup(r,m) ::S (?×?) in
                  (Node(u,y),min2(v,z)) } }
rm_tup(t,m) ::S (?×?)
   = case t of { Leaf(a) -> (Leaf(m),a);
     Node(l,r) -> let (u,v)=rm_tup(l,m) ::S (?×?) in
                  let (y,z)=rm_tup(r,m) ::S (?×?) in
                  (Node(u,y),min2(v,z)) }
```

Such functions are less efficient than their untupled equivalent. Fortunately, our enhanced tupling algorithm avoids such functions, since Step *S5* only introduces each tupled function when its corresponding tuple-closure can be eliminated.

If `mintip` is used under a tail-strict `$` context, we can obtain a more efficient tupled program:

```
mintip(t) ::S $= letrec (u,v)=rm_tup(t,v) ::S ($×?) in u
rm_tup(t,m) ::S ($×?)
   = case t of { Leaf(a) -> let!  r=Leaf(m) in (r,a);
     Node(l,r) -> let!  (u,v)=rm_tup(l,m) ::S ($×?) in
                  let!  (y,z)=rm_tup(r,m) ::S ($×?) in
                  let!  r=Node(u,y) in (r,min2(v,z)) }
```

No tuple-closures will be built on the heap. However, the second component, involving `min2` calls, will still be built as thunks.

An even better result can be obtained if `mintip` is transformed under a totally-strict $\star$ context. Under this scenario, our tupling algorithm obtains:

```
mintip(t) ::S ⋆ = letrec (u,v)=rm_tup(t,v) ::S (⋆×!) in u
rm_tup(t,m) ::S (⋆×!)
   = case t of { Leaf(a) -> let!  r=Leaf(m) in (r,a);
     Node(l,r) -> let!  (u,v)=rm_tup(l,m) ::S (⋆×!) in
                  let!  (y,z)=rm_tup(r,m) ::S (⋆×!) in
                  let!  r=Node(u,y) in
                  let!  n=min2(v,z) in (r,n) }
```

Performance for all three tupled programs, under their respective strictness contexts, are shown in Table 3. Naive-tupling under !-context ends up being worse than no tupling. However, both $-strict and $\star$-strict tuplings are better by about 16% and 18%, respectively, when compared to the corresponding untupled programs with the same level of strictness. This gain is due solely to the elimination of multiple traversals. The tupled function of *mintip* under $\star$-strictness is itself about 18% better than the corresponding tupled function under $-strictness. This gain is due to the elimination of closures for the second component of `rm_tup` calls.

100 times of `mintip` on a tree of depth 12					
	Heap	Time(s)			
	(bytes)	INIT	MUT	GC	Total
No Tupling (!)	29,679,824	0.02	19.49	0.57	20.08
Tupling with (!)	52,620,228	0.01	27.81	4.14	31.96
No Tupling ($)	28,039,728	0.03	17.28	0.85	18.16
Tupling with ($)	23,126,952	0.02	14.51	0.80	15.33
No Tupling ($\star$)	24,762,928	0.03	14.51	0.57	15.11
Tupling with ($\star$)	18,211,728	0.02	11.95	0.34	12.31

Table 3: Execution times of `mintip`

7 Speculative Strictness

The optimisation achieved by our tupling algorithm is to a large extent determined by the level of strictness detected. Better strictness can often be found if more specialised strictness contexts are considered. However, a drawback is that more specialised contexts often meant more code duplication. For example, if we decide to keep all three strictness contexts for `mintip`, we will need to keep three variants of its tupled functions.

An alternative strategy is to make use of *speculative strictness*. Conventional strictness analysis is used to detect expressions which are definitely needed, and could therefore be evaluated safely in advance. Speculative strictness, on the other hand, is used to detect expressions which may not be needed but are nevertheless safe to evaluate because their (advance) evaluations do not contribute to new sources of non-termination.

Definition 1: *Speculative Strictness*
Given an expression e, under non-strict context `?`, and strictness environment θ, we could safely evaluate e under speculative strictness s if its evaluation does not result in non-termination, i.e. $\theta \vdash eval_s\, e \neq \bot$. Note that $eval_s\, e$ denotes the evaluation of e to the strictness extent s.

To support speculative strictness, we propose a new set of analysis rules in Fig. 6 which could determine the extent an expression e can be safely over-evaluated, under strictness environment θ. This forward analysis is written as $\theta \vdash e ::_{\mathcal{O}} s$, where s is the speculative strictness of e under θ. Note that data construction is written in the figure as $c(\mathtt{e}_1, \ldots, \mathtt{e}_m \mid \mathtt{e}_{m+1}, \ldots, \mathtt{e}_n)$, where $\mathtt{e}_1, \ldots, \mathtt{e}_m$ are the non-recursive components of the constructor, and $\mathtt{e}_{m+1}, \ldots, \mathtt{e}_n$ are the recursive components. For example, the list constructor can be written as `cons(x|xs)` .

Some simple examples of speculative strictness are given below:

$$\begin{array}{rcl} \{\mathtt{x} ::_{\mathcal{S}} \mathtt{!}, \mathtt{p} ::_{\mathcal{S}} \mathtt{!}\} & \vdash & \mathtt{(x>p)} ::_{\mathcal{O}} \mathtt{!} \\ \{\mathtt{x} ::_{\mathcal{S}} \mathtt{!}\} & \vdash & \mathtt{(x<p)} ::_{\mathcal{O}} \mathtt{?} \\ \{\mathtt{x} ::_{\mathcal{S}} \mathtt{!}, \mathtt{p} ::_{\mathcal{S}} \mathtt{!}\} & \vdash & \mathtt{(x/p)} ::_{\mathcal{O}} \mathtt{!} \\ \{\mathtt{x} ::_{\mathcal{S}} \mathtt{!}, \mathtt{p} ::_{\mathcal{S}} \mathtt{!}\} & \vdash & \mathtt{(x+p)} ::_{\mathcal{O}} \mathtt{!} \end{array}$$

The strictness environment $\{\mathtt{x} ::_{\mathcal{S}} \mathtt{!}, \mathtt{p} ::_{\mathcal{S}} \mathtt{!}\}$ indicates that both `x` and `p` are assumed to have been evaluated. Under this assumption, the expression `(x>p)` can be speculatively evaluated with strictness `!`, even though it may not be required. This is possible since such an evaluation will not cause non-termination, under the assumption that `x` and `p` are already evaluated. In the second example, `(x<p)` cannot be speculatively evaluated since its argument `p` (which may be $\bot$) has not been evaluated yet.

Some operators such as `/` and `+` may cause run-time errors, e.g. divide-by-zero or overflow exceptions, even when their inputs have already been evaluated. Such exceptions may alter their programs' semantics, making them unsuitable for speculation. Fortunately, modern architectures provide a solution to this problem [MCH+92] by suppressing these exceptions initially, and re-asserting them later when it is determined that they occur in the original program. Our formulation of speculative strictness aggressively assumes this capability.

To illustrate the usefulness of speculative strictness, consider the quicksort function.

```
qsort(xs)    = case xs of { [] -> [];
   x:xs' -> qsort(lower(xs',x))++[x]
              ++qsort(higher(xs',x)) }
lower(xs,p)  = case xs of { [] -> [];
   x:xs' -> case (x < p) of {True -> x:lower(xs',p)
                             False -> lower(xs',p)} }
higher(xs,p) = case xs of { [] -> [];
   x:xs' -> case (x ≥ p) of {True -> x:higher(xs',p)
                             False -> higher(xs',p)} }
```

The `qsort` function has strictness signature
`qsort` $::_{\mathcal{S}}$ `{$ -> !, $ -> $, * -> *}`

If we apply our tupling algorithm under strictness context `!` (but without speculative strictness), we could gather its two calls `(lower(xs,x),higher(xs,x))` under strictness (`$`×`?`), followed by transformation to:

```
qsort(xs) ::S !
   = case xs of { [] -> [];
     x:xs' -> let!  (u,v)=lh_tup(xs',x) ::S($×?)
              in qsort(u)++[x]++qsort(v) }
lh_tup(xs,p) ::S($×?)
   = case xs of { [] -> ([],[]);
   x:xs' -> let!  (u,v)=lh_tup(xs',p) ::S($×?) in
    case (x < p) of {
    True -> let!  a=x:u in
            (a,case (x ≥ p) of { True -> x:v;
                                 False -> v});
    False -> (u,case (x ≥ p) of { True -> x:v;
                                  False -> v})}
```

This tupled program is already quite efficient[2], having avoided the need for tuple-closures, and also closures for the first components. However, closures for the second components, involving `higher` calls, are still created. To avoid these closures, there is no need to resort to a

[2]It is possible to improve the tupled version of `qsort` further by exploiting the fact that `(x≥p)=not(x<p)`. This would then allow two common tests to be combined, with two dead branches of inner `case` eliminated. Though we do not show it here, it is a side-benefit make possible by tupling tactic.

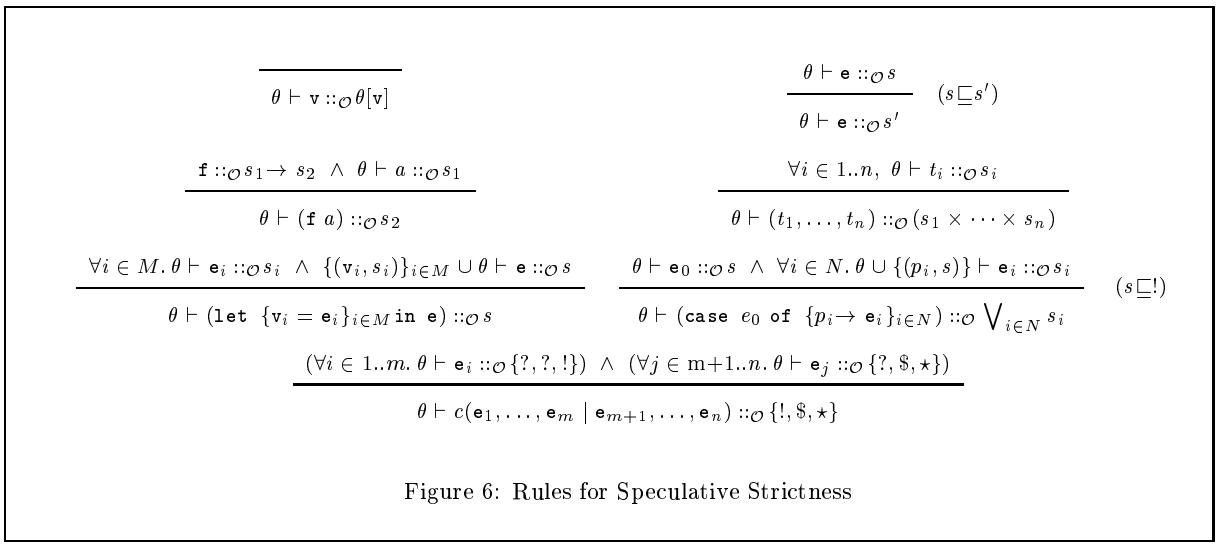

Figure 6: Rules for Speculative Strictness

`$`-strict context for `qsort`. Instead, we could use speculative strictness on the second component, under its strictness environment θ={`x` $::_{\mathcal{S}}$ `!`,`xs` $::_{\mathcal{S}}$`$`,`p` $::_{\mathcal{S}}$ `!`}. The two inner `case` constructs has a common test, `(x` $\geq$ `p)` $::_{\mathcal{O}}$ `!`, that can be speculatively strict, and so are the cons-cell of `(x:v)` $::_{\mathcal{O}}$ `!`. Applying rules *S8* (to lift `case` speculatively) and *S10* (to make tuple-component speculatively strict), we obtain the following optimised code where closures for the second components have now been eliminated.

```
lh_tup(xs,p) ::S ($×?)
   = case xs of { [] -> ([],[]);
      x:xs' -> let!  (u,v)=lh_tup(xs',p) ::S ($×?) in
      case (x < p) of {
       True -> let!  {a=x:u; b=x:v} in
               case (x ≥ p) of { True -> (a,b);
                                 False -> (a,v)});
       False -> let!  b=x:v in
                case (x ≥ p) of {True -> (u,b);
                                 False -> (u,v)}}
```

Speculative strictness can help reduce the amount of thunks created. However, there is still a trade-off involved, namely that if a speculatively strict expression e is not used, its over-evaluation becomes an overhead.

To highlight the usefulness of speculation, we measured the average time taken for `take(sort(xs),r)` with `r` ranging from `1` to |`xs`|. The performance of three versions of `qsort` are shown in Table 4. The tupled version of `qsort` under `!`-strict context has about the same speed as the untupled version. The gain from multiple traversals has been eroded by the need to build tuples for the second components of tuple-call. With the aid of speculative strictness, our tupled program now runs about 16% faster, despite the fact that some of the speculative evaluations may be redundant. This gain comes from the elimination of closures via safe over-evaluation.

1000 times of `qsort` on 1000 integers The n-th time taking the first n integers					
	Heap	Time(s)			
	(bytes)	INIT	MUT	GC	Total
No Tupling	146,336,536	0.02	146.44	0.44	146.90
Tupling with (`!`)	186,535,224	0.03	148.89	0.63	149.55
+ Speculation	134,184,556	0.00	125.68	0.23	125.91

Table 4: Execution times of `qsort`

8 Discussion

It is relatively easy to show that our tupling transformation terminates. First, we observe that the application of $\mathcal{T}$ either advances towards the subexpressions, or reduces the complexity of the expression at hand. Second, it is likely that there be an increase in the number of possible tuple functions created during transformation, compared to the conventional (basic) tupling tactic. Such increment is due to the specialisation of calls with respect to strictness information, in addition to the usual symbolic arguments. Since there are only finitely many distinct strictness information associated with a function, the increment in the number of tuple functions created will be bounded. Thus, the termination proof of our new tupling tactic can be reduced to that of the conventional tupling [Chi93].

Due to space constraint, we do not provide any correctness proof to our tupling tactics. Nevertheless, we note that such correctness proof depends on the correctness of each transformation rules as well as the soundness of the strictness analysis. The former can be mirrored from the relevant work in program transformation, such as the work by Runciman for ensuring adherence to lazy semantics [RFJ89] and that by Sands to ensure total correctness of transformed programs [San95]. Our strictness analysis can be considered as a simplified variant of disjunctive program analysis [Jen97], and its soundness can be proven in a similar spirit.

We have so far described the mechanism for tupling functions with single recursion arguments. More work needs to be done for devising effective tupling algorithm for functions with multiple recursion arguments in lazy programs. Here, it is necessary to consider synchronisation of changes of multiple arguments between two calls, in addition to the strictness information of the individual arguments. Handling of functions with multiple recursion arguments in strict languages has been described in [CKL98].

9 Conclusion

Most past work on tupling/tabulation techniques, such as those in [Coh83, Pet84, Chi93, HITT97], have focused mainly on the mechanics for realising big gains from the elimination of redundant calls. While impressive, there are still some doubts over how often redundant calls occur in practice. Functions which perform multiple traversals are likely to be more common. However, to the best of our knowledge, there have been no systematic investigation into the use of tupling for the elimination of multiple traversals. This work is useful as it could meaningfully complement the pool of existing optimising techniques for functional programs.

Applying this optimisation to lazy functional languages poses a particularly difficult challenge since naive elimination of multiple traversals could cause performance to degrade, rather than improve. We have demonstrated that due to laziness, such tupling results in the building of unnecessary closures. This is the bane of tupling and manifests itself in the penalty of both heap usage and execution time. To rectify this condition, we have proposed a range of solutions to minimise the building of these closures by forcing their evaluations when it is safe to do so, guided by strictness analysis. By using specialised strictness contexts where possible, we could guide our transformer to uncover more opportunites for effective tupling. Our tupling transformer could also exploit a more aggressive form of strictness, known as speculative strictness. As demonstrated, significant savings are possible, despite the potential for redundant over-evaluation.

It is important to note that the effect of strictness-guided tupling cannot be achieved by naive tupling, followed by (advanced) strictness analysis. Without the guidance of strictness contexts, blind introduction of tuple-functions could cause inefficient tupling to occur, from which strictness analysis alone could not recover.

There is little doubt that tupled functions are very useful. Apart from the elimination of redundant calls and multiple traversals, tupled function are often *linear* with respect to the common arguments (i.e. each now occurs only once in the RHS of the equation). This linearity property has a number of advantages, including:

- It can help avoid *space leaks* that are due to unsynchronised multiple traversals of large data structures, via a compilation technique described in [Spa93].
- It can facilitate deforestation (and other transformations) that impose a *linearity* restriction [Wad88], used for efficiency and/or termination reasons.
- It can improve opportunity for *uniqueness typing* [BS93], which is good for storage overwriting and other optimisations.

Because of these nice performance attributes, functional programmers often go out of their way to write such tupled functions, despite them being more awkward, error-prone and harder to write and read. It is hoped that this burden on programmers could be relieved in the near future. We have a prototype implementation of our strictness-guided tupling algorithm for a restricted first-order language. At the current moment, it requires strictness signatures of user-defined functions to be given. For future work, we plan to extend our proposal to the full higher-order language, and provide for a type-based strictness inference analyser.

References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, January 1977.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[BS93] E. Barendsen and J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 45–51, Bombay, India, December 1993.

[Bur91] G.L. Burn. The evaluation transformer model of reduction and its correctness. In *TAPSOFT (LNCS)*, Brighton, 1991.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, June 1993. ACM Press.

[CKL98] W.N. Chin, S.C. Khoo, and T.W. Lee. Synchronisation analyses to stop tupling. In *European Symposium on Programming (LNCS 1381)*, pages 75–89, April 1998.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Trans. on Programming Languages and Systems*, 5(3):265–299, July 1983.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates

multiple traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, Netherlands, June 1997. ACM Press.

[Jen97] Thomas. Jensen. Disjunctive program analysis for algebraic data types. *ACM Trans. on Programming Languages and Systems*, 19(5):751–803, September 1997.

[MCH+92] S.A. Mahlke, W. Chen, W. Hwu, B. Ramakrishna Rau, and S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *5th ASPLOS*, pages 238–247, Boston, Massachusetts, October 1992. ACM Press.

[Pet84] Alberto Pettorossi. A powerful strategy for deriving programs by transformation. In *3rd ACM LISP and Functional Programming Conference*, pages 273–281. ACM Press, 1984.

[PJP93] Simon Peyton-Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Functional Programming Workshop*, pages 201–220, Springer-Verlag Workshop Series, 1993.

[PJPS96] S. Peyton-Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvannia, May 1996. ACM Press.

[RFJ89] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language : An approach to instantiation. In *Glasgow Functional Programming Workshop*, August 1989.

[San95] David Sands. Total correctness by local improvement in program transformation. In *22nd ACM Principles of Programming Languages Conference*, pages 221–232. ACM Press, January 1995.

[Spa93] Jan Sparud. How to avoid space-leak without a garbage collector. In *ACM Conference on Functional Programming and Computer Architecture*, pages 117–122, Copenhagen, Denmark, June 1993. ACM Press.

[Wad87] Phil Wadler. Strictness analysis in non-flat domains (by abstract interpretation over finite domains). In A. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, UK, 1987.

[Wad88] Phil Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Nancy, France, (LNCS, vol 300, pp. 344–358), March 1988.

Declarative aspect-oriented programming*

Ralf Lämmel

Department of Computer Science
University of Rostock
D-18051 Rostock
Germany

Abstract

Aspect-oriented programming addresses the problem that the implementation of some properties such as error handling and optimization tends to cross-cut the basic functionality. To overcome that problem special languages are used to specify such properties—the so-called aspects—in isolation. The software application is obtained by weaving the aspect code and the implementation of properties corresponding to basic functionality—the so-called components. This paper investigates the suitability of functional meta-programs to specify aspects and to perform weaving. The proposal focuses on the declarative paradigm (logic programming, attribute grammars, natural semantics, constructive algebraic specification etc.) as far as components are concerned, whereas aspects are represented by program transformations. Weaving is regarded as a program composition returning a combination of the components satisfying all the aspects. The computational behaviour of the components is preserved during weaving. The proposal improves reusability of declarative programs. The approach is generic in the sense that it is applicable to several representatives of the declarative paradigm. Several roles of aspect code are defined and analysed.

1 Introduction

1.1 Aspect-oriented programming

Kiczales et al. recently proposed aspect-oriented programming (AOP) [KLM+97, AOP97, AOP98] as an extension of the traditional approach to programming coping well with functional decomposition. Procedures, functions, methods, modules, APIs, classes etc. are used in the traditional approach to implement all kind of properties. However, certain properties such as error handling and optimization tend to cross-cut the functionality resulting in tangled code which is then unclear and hard to modify and adapt. AOP attempts to close this well-known gap between requirements / design and implementation.

AOP is based on the following assumptions. Properties which must be implemented are subdivided into *components* corresponding to basic functionality and *aspects* corresponding to non-functional properties. A property is a *component* if it can be cleanly encapsulated in a procedure, a method etc. Otherwise it is an *aspect*. Components tend to be units of the system's functional decomposition, whereas aspects cross-cut the system's functionality and they usually affect the performance or semantics of the components in systemic ways. Thus, components are usually developed in an imperative / object-oriented language, whereas special language support is needed for the development of aspect code. The application is obtained by *weaving* the components and the aspects. Note that without AOP the properties corresponding to aspects have to be scattered throughout the code representing the basic functionality.

Example 1 Suppose persistency must be added to an object-oriented program. The implementation of the property "persistency" cross-cuts the classes implementing the basic functionality because the classes need to be adapted in a *systemic* way to "externalize" their state and to support "population" when objects are re-created.

Let us consider a very specific aspect in the context of persistency, that is to say *efficient* persistency where a persistency manager should keep track of the objects which have changed their state. Thereby, the update of the persistent storage can be done more efficiently.

Given a class *foo*, for example, the instances of which should be persistent, a method call like the one in the box below must be inserted for every change of the internal state.

```
class foo extends ··· {
// attributes
  type1 attr1;
  ···

// methods
  public result_type1 operation1 (···) {
    ···
    attr1 = ···; // changing the state
    this.keep_persistency_manager_informed();
    ···
  }
  ···
}
```

Thus, an aspect language is needed to specify this kind of systematic adaptation. Somehow it must be possible to specify the join point (i.e. the point where the aspect is woven into the component) "the statement after an assignment to an attribute". Weaving means to take some class definitions and the aspect code and to emit class definitions with the efficient persistency implemented. ♢

The original proposal of AOP [KLM+97] leaves open the question what actual languages for developing aspect code and what actual forms of weaving are appropriate. Since

*This work was supported, in part, by *Deutsche Forschungsgemeinschaft*, in the project *KOKS*.

then, several researchers have proposed potential aspect languages and forms of weaving, from readily available but dedicated weavers to sophisticated program transformations still to be implemented. One example in [KLM+97] deals with loop fusion (that is a kind of optimization), where the aspect code can be regarded as a set of rewrite rules acting on data flow diagrams derived from the component code. Weaving means here to build the data flow diagrams, to apply the rewrite rules and to emit C-code from the "optimized" data flow diagrams.

Thus, the fundamental question in AOP is what are the aspects, how to represent them and how to weave components and aspects. This paper attempts to answer these questions in a certain way.

1.2 Another instance of AOP

Our paper provides a general but still effective proposal for aspect code and weaving based on functional meta-programs. Components are declarative programs, whereas aspects are implemented by program transformations. Weaving is considered as a program composition combining components and aspects by essentially applying the transformations modelling aspects to components. Note that the original proposal of AOP and most work in the field focus on procedural (object-oriented) languages as far as components are concerned, whereas our concrete instance of AOP relies on declarative languages.

Typical examples of aspects in declarative programming are concerned with

- optimization, e.g. based on fold/unfold-strategies,
- failure handling, error recovery, exception handling,
- propagation, accumulation, synthesis of data,
- stylistic properties, e.g. CPS based on conversion, and
- refinements of the computational behaviour.

Our proposal for AOP improves *reusability* of declarative programs because aspects can be described in isolation. Thus, declarative programs can be programmed in a more modular fashion because they may abstract from the aspects. Our approach illustrates an amalgamation of program synthesis, program transformation, program analysis and program composition based on a generic framework for meta-programming in the declarative paradigm. The approach is generic in the sense that is applicable to several representatives of the declarative paradigm, e.g. natural semantics, attribute grammars, (constructive) algebraic specifications and logic programs. The framework for meta-programming is described in Section 2.

Some meta-programming operators modelling roles of aspect code are studied in Section 3. One particular form of weaving is presented in detail in Section 4. A general challenge in AOP is to ensure correctness of weaving, e.g. in the sense that the meaning of the components is preserved by the weaver. Otherwise, AOP would be unnerving because the person writing the aspects could foul the weaver and introduce unintended behaviour into the woven program. Most other works in the field of AOP focus on identification and characterization of aspects, aspect languages and possible forms of weaving. In contrast, this paper attempts to provide some theoretical grounds in the context of correctness. Reasoning is based on certain preservation properties of program transformations. The suggested form of weaving and the properties of the suggested roles of aspect code suffice to guarantee the preservation of the computational behaviour of the components by the weaver.

1.3 The running example

The discussion is rooted by small interpreter examples specified in the style of natural semantics[1]. Figure 1 shows an interpreter definition for a simple imperative language core. The natural semantics consists of two relations $do : \mathsf{C} \times \mathsf{ST} \to \mathsf{ST}$ describing how the execution of statements affects the store and $eval : \mathsf{E} \times \mathsf{ST} \to \mathsf{VAL}$ modelling expression evaluation free of side-effects. In the sense of AOP Figure 1 is a program implementing various properties related to the interpretation of basic language constructs. Note that it is not the intention of this paper to propose a certain style of developing interpreters for (simple) languages. The domain was rather chosen because the properties involved in simple interpreters are well-understood.

$$do(\mathsf{skip}, \mathsf{ST}) \to (\mathsf{ST}) \qquad [\mathsf{skip}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}') \;\wedge\; do(\mathsf{C}_2, \mathsf{ST}') \to (\mathsf{ST}'')}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}'')} \qquad [\mathsf{concat}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}) \to (\mathsf{VAL}) \;\wedge\; update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}')}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}) \to (\mathsf{ST}')} \qquad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}) \to (\mathsf{VAL})} \qquad [\mathsf{var}]$$

$$\ldots$$

Figure 1: An interpreter of a simple language

Now let us assume that we want to derive an interpreter coping with I/O as well. Figure 2 contains the corresponding interpreter rules. It is assumed that the input and the output are modelled by sequences (lists) of values. An expression of the form read is assumed to be evaluated to the head of the sequence of the current input; refer to the rule [read]. A statement of the form write(E) is assumed to output the value of the expression E; refer to the rule [write]. The rule [main] should be regarded as a kind of axiom for interpreting programs consuming an input and producing an output. The problem with the initial program in Figure 1 and the additional component in Figure 2 is that they are not compatible with each other. Figure 1 does not meet the properties "input propagation" and "output accumulation" which are obviously needed for the implementation of I/O in Figure 2. What is needed is an adaptation of Figure 1; refer to Figure 3. Meta-programs can be used to perform such adaptations, where the specifications in the style of natural semantics are regarded as target programs. Informally, the required adaptation can be performed in the following steps. Input and output positions of sort IN are added and the resulting parameters of sort IN are connected to code an accumulator. Output positions of sort OUT are added. Multiple parameters of sort OUT in a rule are combined in premises with the symbol *append*. For some rules the empty output is returned.

[1] The following conventions for natural semantics rules are assumed in this paper: $\to$ is used to separate inputs and outputs in propositions. Sort identifiers are used to derive variables of the sort by adding possibly quotes and indices, e.g. C_1 is a variable of sort C.

The key idea here is that "input propagation" and "output accumulation" are regarded as aspects modelled by program transformations. Weaving would be an elaboration of such a scenario, where several modules are adapted according to some aspects. All these issues are made clear in Section 4.

$$\frac{head(\mathsf{IN}) \to (\mathsf{VAL}) \;\wedge\; tail(\mathsf{IN}) \to (\mathsf{IN}')}{eval(\mathsf{read}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}')} \;[\mathsf{read}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{VAL}, \mathsf{IN}') \;\wedge\; list(\mathsf{VAL}) \to (\mathsf{OUT})}{do(\mathsf{write}(\mathsf{E}), \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}, \mathsf{IN}', \mathsf{OUT})} \;[\mathsf{write}]$$

$$\frac{init \to (\mathsf{ST}) \;\wedge\; do(\mathsf{C}, \mathsf{ST}, \mathsf{IN}) \to (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT})}{prog(\mathsf{C}, \mathsf{IN}) \to (\mathsf{OUT})} \;[\mathsf{main}]$$

Figure 2: I/O constructs

$$\frac{nil \to \mathsf{OUT}}{do(\mathsf{skip}, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}, \boxed{\mathsf{IN}}, \boxed{\mathsf{OUT}})} \;[\mathsf{skip}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}', \boxed{\mathsf{IN}'}, \boxed{\mathsf{OUT}_1}) \;\wedge\; do(\mathsf{C}_2, \mathsf{ST}', \boxed{\mathsf{IN}'}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}''}, \boxed{\mathsf{OUT}_2}) \;\wedge\; \boxed{append(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT})}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}''}, \boxed{\mathsf{OUT}})} \;[\mathsf{concat}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}'}) \;\wedge\; update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \;\wedge\; \boxed{nil \to \mathsf{OUT}}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{ST}', \boxed{\mathsf{IN}'}, \boxed{\mathsf{OUT}})} \;[\mathsf{assign}]$$

$$\ldots$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \boxed{\mathsf{IN}}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}})} \;[\mathsf{var}]$$

$$\ldots$$

Figure 3: An adaptation to cope with input/output

The remaining paper is structured as follows. First, a formal, generic framework for functional meta-programs is developed. Afterwards, certain operators facilitating program manipulation are outlined. These operators will be useful to implement aspects by means of meta-programs. We carry on by defining our instance of AOP. Finally, we comment on results, related work and future work.

2 The meta-programming framework

Our approach to the development of aspect code and to weaving is based on a meta-programming framework which is developed in the following steps. First, a representation for declarative target programs such as natural semantics, attribute grammars, logic programs and (constructive) algebraic specifications is declared. Second, the corresponding structural definitions are restricted to obtain the domains of proper target program fragments. These domains are embedded into a typed λ-calculus, where some further specification constructs are added as well. Finally, properties of meta-programs, e.g. preservation properties, are defined.

2.1 The representation of target programs

The representation of target programs (such as the rules in Figure 1 etc.) and fragments of them is given by certain domains which are intended to capture the common constructs in (first-order) declarative languages such as rules, propositions and parameters. There is, for example, a domain Rule which can be regarded as an abstraction from inference rules in natural semantics or syntactical rules together with the attributes + semantic rules in attribute grammars.

We assume the following domains:

- Rules *compatible* collections of rules
- Rule tagged rules
- Conclusion conclusions
- Premise premises
- Element parameterized names (propositions)
- Parameter annotated parameters such as variables
- Tag countable set of tags
- Name countable set of names
- Variable countable set of variable identifiers
- Sort countable set of sort identifiers

Example 2 The rules shown in Figure 1 form an element of Rules. There are rules (elements of Rule) tagged by [skip], [concat], [assign] and [var]. The conclusion of the rule [concat] is $do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}'')$, whereas the premises of the rule are $do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}')$ and $do(\mathsf{C}_2, \mathsf{ST}') \to (\mathsf{ST}'')$. The conclusion and the premises are parameterized names, i.e. elements of Element. Parameters are variables, e.g. ST with several occurrences in [concat], or proper terms, e.g. $\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2)$. ♢

Refer to Figure 4 for the corresponding structural definition[2]. Note that barred names are used to point out that the structural definition needs to be restricted further to define the corresponding domain of *proper* fragments. Proper collections of rules, for example, are modelled by the domain $\overline{\mathsf{Rules}}$ obtained as a restriction of the structural definition $\overline{\mathsf{Rules}}$, where the restriction is concerned, for example, with compatibility of the types of all the single rules.

$$\begin{aligned}
\overline{\mathsf{Rules}} &= \overline{\mathsf{Rule}}^* \\
\overline{\mathsf{Rule}} &= \mathsf{Tag} \times \overline{\mathsf{Conclusion}} \times \overline{\mathsf{Premise}}^* \\
\overline{\mathsf{Conclusion}} &= \overline{\mathsf{Element}} \\
\overline{\mathsf{Premise}} &= \overline{\mathsf{Element}} + \cdots \\
\overline{\mathsf{Element}} &= \mathsf{Name} \times \overline{\mathsf{Parameter}}^* \times \overline{\mathsf{Parameter}}^* \\
\overline{\mathsf{Parameter}} &= (\mathsf{Variable} + \cdots) \times \mathsf{Sort}
\end{aligned}$$

Refinement assumed in this paper

$$\begin{aligned}
\overline{\mathsf{Rules}} &= \mathcal{P}(\overline{\mathsf{Rule}}) \\
\overline{\mathsf{Parameter}} &= (\mathsf{Variable} + \overline{\mathsf{Term}}) \times \mathsf{Sort} \\
\overline{\mathsf{Term}} &= \mathsf{Functor} \times \overline{\mathsf{Parameter}}^*
\end{aligned}$$

Figure 4: Structural definition of representations

The structural definition has possibly to be refined and extended for particular instances. The domain Parameter for example needs to be extended for compound parameters in contrast to variables in the sense of terms in natural semantics. Another possible extension concerns the domain Premise which must be extended to cope with semantic rules

[2] $\mathcal{P}$ denotes the power set constructor. We are mainly concerned with finite subsets in this paper.

in attribute grammars. In this paper a refinement of the domains is assumed which copes with terms based on the following additional domains:

Term compound parameters (terms)
Functor countable set of functors for terms

Furthermore, we abstract from the order of rules in Rules. Thus, collections of rules are rather subsets of Rule than sequences. Refer to Figure 4 for the corresponding domain equations.

2.2 Properties of target programs

The structural definition from Figure 4 is restricted to obtain domains of proper fragments. Thereby, it can be guaranteed that meta-programs deal with rather proper fragments than arbitrary representations. Technically, inference rules are given to obtain the domains $\mathsf{Rules} \subset \overline{\mathsf{Rules}}$, ..., $\mathsf{Parameter} \subset \overline{\mathsf{Parameter}}$; refer to Figure 5[3]. Proper fragments are expected to satisfy

- well-formedness ($\mathcal{WF}$) in the sense of basic requirements, e.g. that the tags of a collection of rules are pairwise distinct or—more generally—that proper compound fragments are built only from proper fragments and
- well-typedness ($\mathcal{WT}$) in the sense of a type system with sorts and modes as used for example in logic programming [Boy96], i.e. it must be possible to associate profiles with all the symbols used in a target program fragment.

For *complete* target programs further properties can be relevant, e.g. certain completeness properties such as reducedness in the sense of context-free grammars (CFG) or non-circularity in the sense of attribute grammars. A particular property dealing with some minimal requirement concerning the data flow will be considered below.

$$\frac{\begin{array}{l}\overline{r}_i \in \mathsf{Rule} \text{ for } i = 1, \ldots, n \\ \wedge\ \pi_{\mathsf{Tag}}(\overline{r}_i) \neq \pi_{\mathsf{Tag}}(\overline{r}_j) \text{ for } i, j = 1, \ldots, n, i \neq j \\ \wedge\ \exists\, \Sigma : (\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i, \ldots, n)\end{array}}{\{\overline{r}_1, \ldots, \overline{r}_n\} \in \mathsf{Rules}} \quad [\mathsf{Rules}]$$

$$\frac{\begin{array}{l}\overline{r} \in \overline{\mathsf{Rule}} \\ \wedge\ \pi_2(\overline{r}) \in \mathsf{Element} \\ \wedge\ \pi_i(\pi_3(\overline{r})) \in \mathsf{Element} \text{ for } i = 1, \ldots, \#\pi_3(\overline{r}) \\ \wedge\ \exists\, \Sigma : \mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r})\end{array}}{\overline{r} \in \mathsf{Rule}} \quad [\mathsf{Rule}]$$

$$\frac{\begin{array}{l}\overline{e} \in \overline{\mathsf{Element}} \text{ is of the form } \langle n, in, out\rangle \\ \wedge\ \pi_i(in) \in \mathsf{Parameter} \text{ for } i = 1, \ldots, \#in \\ \wedge\ \pi_i(out) \in \mathsf{Parameter} \text{ for } i = 1, \ldots, \#out \\ \wedge\ \exists\, , , \Sigma : \mathcal{WT}_{\mathsf{Element}}(, , \Sigma, \overline{e})\end{array}}{\overline{e} \in \mathsf{Element}} \quad [\mathsf{Element}]$$

$$\frac{\begin{array}{l}\overline{p} \in \overline{\mathsf{Parameter}} \\ \wedge\ \exists\, , , \Sigma : \mathcal{TYPE}_{\mathsf{Parameter}}(, , \Sigma, \overline{p}) = \pi_{\mathsf{Sort}}(\overline{p})\end{array}}{\overline{p} \in \mathsf{Parameter}} \quad [\mathsf{Parameter}]$$

Figure 5: Properties of target program fragments

[3] π_i denotes the i-th projection for tuples and sequences. For a product $D = D_1 \times \cdots \times D_n$ the notation π_{D_i} is also used for π_i if the D is obvious from the context and the i is uniquely defined by the D_i. $\#s$ denotes the length of the sequence s.

$\mathcal{WF}$ is encoded directly in the inference rules in Figure 5, whereas most details of $\mathcal{WT}$ are defined by auxiliary relations shown in Figure 6.[4] Consider for example the inference rule [Rules] defining proper collections of rules. Its premises state the following properties:

- The single rules must be proper rules themselves ($\mathcal{WF}$).
- The tags must be pairwise distinct ($\mathcal{WF}$).
- The types of the rules must be compatible ($\mathcal{WT}$).

$$\frac{\begin{array}{l}\mathcal{WT}_{\mathsf{Rules}}(\Sigma, \overline{rs}) \\ \wedge\ \Sigma \text{ is minimal, i.e. } \forall\, \Sigma' \neq \Sigma : \\ \quad \mathcal{WT}_{\mathsf{Rules}}(\Sigma', \overline{rs}) \Rightarrow |\Sigma| \leq \left|\Sigma'\right|\end{array}}{\mathcal{TYPE}_{\mathsf{Rules}}(\overline{rs}) \to \Sigma} \quad [\mathcal{WT}.1]$$

$$\frac{\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i = 1, \ldots, n}{\mathcal{WT}_{\mathsf{Rules}}(\Sigma, \{\overline{r}_1, \ldots, \overline{r}_n\})} \quad [\mathcal{WT}.2]$$

$$\frac{\exists\, , : (\mathcal{WT}_{\mathsf{Element}}(, , \Sigma, \overline{e}_i) \text{ for } i = 0, \ldots, n)}{\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \langle t, \overline{e}_0, \langle \overline{e}_1, \ldots, \overline{e}_n\rangle\rangle)} \quad [\mathcal{WT}.3]$$

$$\frac{\begin{array}{l}s : \sigma_1 \times \cdots \times \sigma_m \to \sigma_{m+1} \times \cdots \times \sigma_k \in \Sigma \\ \wedge\ \mathbf{in}_{\mathsf{Name}}(n) = s \\ \wedge\ \mathcal{TYPE}_{\mathsf{Parameter}}(, , \Sigma, \overline{p}_i) \to \sigma_i \\ \quad \text{for } i = 1, \ldots, k\end{array}}{\mathcal{WT}_{\mathsf{Element}}(, , \Sigma, \langle n, \langle \overline{p}_1, \ldots, \overline{p}_m\rangle, \langle \overline{p}_{m+1}, \ldots, \overline{p}_k\rangle\rangle)} \quad [\mathcal{WT}.4]$$

$$\frac{\begin{array}{l}\exists\, v : \mathbf{in}_{\mathsf{Variable}}(v) = \pi_1(\overline{p}) \\ \wedge\ \pi_{\mathsf{Sort}}(\overline{p}) = \sigma \\ \wedge\ (v : \sigma) \in ,\end{array}}{\mathcal{TYPE}_{\mathsf{Parameter}}(, , \Sigma, \overline{p}) \to \sigma} \quad [\mathcal{WT}.5]$$

Figure 6: The type system based on sorts and modes

Let us consider $\mathcal{WT}$ slightly more in detail. Symbols such as names used in elements (that is to say propositions) or functors used in terms get associated profiles based on sorts and modes, i.e. there are some input and some output positions each of a certain sort. We assume the following domains:

$$\begin{array}{lcll}\mathsf{Sigma} & \subseteq & \overline{\mathsf{Sigma}} & = \mathcal{P}(\overline{\mathsf{Profile}}) \\ \mathsf{Profile} & \subseteq & \overline{\mathsf{Profile}} & = \mathsf{Symbol} \times \mathsf{Sort}^\star \times \mathsf{Sort}^\star \\ \mathsf{Symbol} & = & & \mathsf{Name} + \mathsf{Functor} + \cdots\end{array}$$

In certain instances, proper profiles have to be restricted. The profile of a functor, for example, has a simple target in contrast to a proper Cartesian product. Signatures Σ are (finite) subsets of Profile. Again restrictions might be appropriate in certain instances. In the paper it is assumed for example that overloading is prohibited, i.e. a signature $\Sigma \in \mathsf{Sigma}$ must satisfy that $\forall p, p' \in \Sigma$:

$$\pi_{\mathsf{Symbol}}(p) = \pi_{\mathsf{Symbol}}(p') \Rightarrow p = p'.$$

The initial type system (without terms) is presented in Figure 6. $\mathcal{TYPE}_{\mathsf{Rules}}(rs)$, for example, denotes the type (i.e. the signature) of some rules rs. The type system can be refined to cope with specific constructs and properties in particular instances, e.g. terms.

[4] As far as coalesced sums $D = D_1 + \cdots + D_n$ are concerned, $\mathbf{in}_{D_i}(d)$ denotes the injection of $d \in D_i$ into D. Note that the D should be obvious from the context.

Example 3 $\mathcal{TYPE}_{\mathsf{Rules}}$(Figure 1) corresponds to the following set of profiles:

$$\begin{array}{lcl} do & : & \mathsf{C} \times \mathsf{ST} \to \mathsf{ST} \\ eval & : & \mathsf{E} \times \mathsf{ST} \to \mathsf{VAL} \\ update & : & \mathsf{ST} \times \mathsf{ID} \times \mathsf{VAL} \to \mathsf{ST} \\ apply & : & \mathsf{ST} \times \mathsf{ID} \to \mathsf{VAL} \\ \mathsf{skip} & : & \to \mathsf{C} \\ \mathsf{concat} & : & \mathsf{C} \times \mathsf{C} \to \mathsf{C} \\ \mathsf{assign} & : & \mathsf{ID} \times \mathsf{E} \to \mathsf{C} \\ \mathsf{var} & : & \mathsf{ID} \to \mathsf{E} \\ \ldots & & \end{array}$$

♢

Note that all target program fragments shown in the paper are proper fragments what implies that they are well-typed. Constructor operations according to the notation for axioms, inference rules and propositions used in Figure 1 etc. are assumed. These constructors must be regarded as partial in the sense that applications of them are defined iff the resulting fragment is a proper fragment.

As it was mentioned above *complete* target programs must probably satisfy further specific properties. The following definition provides some terms regarding a minimum requirement for the completeness of the data flow.

Definition 1 Given a rule $r \in \mathsf{Rule}$, the output (resp. input) positions of the conclusion and the input (resp. output) positions of the premises in r are called *applied* (resp. *defining*) *positions in* r. *Applied* resp. *defining variable occurrences* are variables on applied resp. defining positions. The set of applied resp. defining variable occurrences in r is denoted by $\mathcal{AO}(r)$ resp. $\mathcal{DO}(r)$.

The *data flow* in a collection of rules $rs \in \mathsf{Rules}$ is called *complete* if $\mathcal{DFC}(rs)$ holds.

$$\frac{\mathcal{AO}(\overline{r}_i) \subseteq \mathcal{DO}(\overline{r}_i) \text{ for } i = 1, \ldots, n}{\mathcal{DFC}(\{\overline{r}_1, \ldots, \overline{r}_n\})} \quad [\mathcal{DFC}]$$

♢

Thus, completeness of the data flow in a collection of rules means that for each rule r the applied variable occurrences ($\mathcal{AO}(r)$) are contained in the defining variable occurrences ($\mathcal{DO}(r)$). The idea behind the terms applied and defining positions is that the variables with occurrences on applied positions are expected to be "computed" in terms of variables with occurrences on the defining positions. These terms are used in much the same way in, for example, extended attribute grammars [WM77]. Thereby, we may speak of *undefined* and *unused variables*, where a variable v is undefined in the rule r if $v \in \mathcal{AO}(r) \setminus \mathcal{DO}(r)$; dually for unused variables.

Example 4 All the variables in Figure 1 have a defining occurrence. Thus, $\mathcal{DFC}$(Figure 1) holds and there are no undefined variables. Actually, there are no unused variables either because all variables with a defining occurrence have an applied occurrence. ♢

$\mathcal{DFC}$ should not be required for intermediate results of program transformations but only for final results, that is to say complete programs (modules). To transform Figure 1 into Figure 3, for example, it is very suitable to insert first the additional parameter positions resulting in an intermediate result which does not satisfy $\mathcal{DFC}$. The additional premises are inserted and the data flow is established afterwards. Refer to Figure 8 for the intermediate result developed in the next section.

Instantiating and refining the framework probably other or more refined properties than just $\mathcal{DFC}$ will be relevant for complete programs, e.g. non-circularity in attribute grammars, call-correctness in logic programs [Boy96], unknowns in natural semantics, reducedness or interface conformance properties in the sense of module systems.

Another notion is needed to reason about target programs. The notion of a skeleton is similar in intent to the notion of the underlying CFG of an attribute grammar. Roughly, the skeleton of some rules is obtained by considering only the shapes of the rules, where the shape of a rule is a triple consisting of its tag, the name of the conclusion and the sequence of names of those premises which are meant to contribute to the skeleton. The "contributing" symbols are at least the defined symbols, i.e. symbols with an occurrence in a conclusion. Thus, in a sense a skeleton is degenerated collection of rules, without parameterization. Skeletons are a useful tool in meta-programs to abstract from the structure of target programs (components). Moreover, the notion facilitates pairing of rules. These applications will be clarified later on.

Definition 2 Let be $rs = \{r_1, \ldots, r_n\} \in \mathsf{Rules}$ and $ss \in \mathcal{P}(\mathsf{Name})$. The *defined symbols in* rs are denoted by $\mathcal{DS}(rs)$.

$$\mathcal{DS}(rs) = \bigcup_{i=1}^{n} \pi_{\mathsf{Name}}(\pi_{\mathsf{Conclusion}}(r_i))$$

The *skeleton of* rs *w.r.t.* ss is the set $\{sh_1, \ldots, sh_n\} \in \mathcal{P}(\mathsf{Tag} \times \mathsf{Name} \times \mathsf{Name}^\star)$ such that

$$\begin{array}{rcl} \pi_{\mathsf{Tag}}(sh_i) & = & \pi_{\mathsf{Tag}}(r_i) \\ \pi_{\mathsf{Name}}(sh_i) & = & \pi_{\mathsf{Name}}(\pi_{\mathsf{Conclusion}}(r_i)) \\ \pi_{\mathsf{Name}^\star}(sh_i) & = & \langle s_1, \ldots, s_m \rangle \end{array}$$

with $s_1, \ldots, s_m \in \mathcal{DS}(rs) \cup ss$ and there are natural numbers $q_1, \ldots, q_m$ such that $1 \leq q_1 < \cdots < q_m \leq premises$, $s_j = name_{q_j}$ for $j = 1, \ldots, m$ and $\forall k \in \{1, \ldots, premises\} \setminus \{q_1, \ldots, q_m\}$: $name_k \notin \mathcal{DS}(rs) \cup ss$, where

$$\begin{array}{rcl} premises & = & \#\pi_{\mathsf{Premise}^\star}(r_i) \\ name_x & = & \pi_{\mathsf{Name}}(\pi_x(\pi_{\mathsf{Premise}^\star}(r_i))) \end{array}$$

for $i = 1, \ldots, n$.

The skeleton of rs w.r.t. ss is denoted by

$$\mathcal{SKELETON}(rs, ss).$$

♢

The role of the names ss in the above definition is to specify more skeleton symbols, i.e. symbols contributing to the skeleton, than just the defined symbols. That is necessary if incomplete target programs, e.g. modules in the sense of components, are taken into consideration. Note that if $\mathcal{SKELETON}(rs_1, ss) = \mathcal{SKELETON}(rs_2, ss)$, then $\mathcal{SKELETON}(rs_1, ss') = \mathcal{SKELETON}(rs_2, ss')$ for $ss' \subseteq ss$. Figure 7 provides the corresponding structural definition and the restriction of it to characterize proper skeletons. For proper collections of shapes it must hold that the tags are pairwise distinct, similarly to proper collections of rules.

Example 5 Using a CFG notation $\mathcal{SKELETON}$(Figure 1, $\emptyset$) can be represented as follows:

$$\begin{array}{lll} [\mathsf{skip}] & do & : \; . \\ [\mathsf{concat}] & do & : \; do, do. \\ [\mathsf{assign}] & do & : \; eval. \\ \ldots & & \\ [\mathsf{var}] & eval & : \; . \\ \ldots & & \end{array}$$

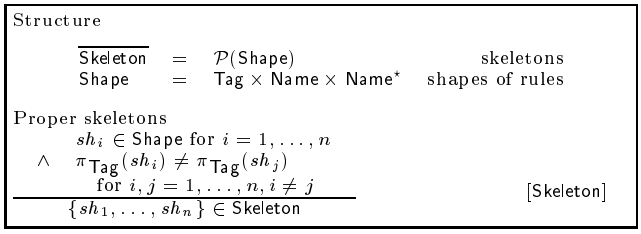

Figure 7: Skeletons

Note that Figure 3 has the same skeleton. In both cases all the auxiliary computations accessing the store, producing and combining outputs etc. do not contribute to the skeleton. ♢

Finally, a few remarks on equality on Rules are in place. Structural equality modulo renaming of variables is denoted by $(\_ = \_)$. Furthermore, equivalence classes in Rules are considered in order to abstract from the order of parameter positions. $rs' \in [rs]_{ss} \subset \mathsf{Rules}$ means that rs' can be transformed into some $rs'' = rs$ by only changing consistently the order of parameter positions of elements e with $\pi_{\mathsf{Name}}(e) \in ss$ all over rs'. Note that changing the order of parameter positions does not introduce technical problems as long as uniqueness for sorts on input and output positions is assumed.[5]

2.3 Functional meta-programs

To obtain a meta-programming language, it is proposed to embed the data types for meta-programming into a typed λ-calculus. Functional meta-programs are preferred because of the applicability of equational reasoning for proving properties and the suitability of higher-order functional programming to write abstract program manipulations.

Furthermore, the following specification language constructs are assumed in the resulting calculus:

- *foldl* / *foldr*, non-recursive / recursive *let*,
- the Boolean data type and the conditional $b \to e_1, e_2$,
- products ($\times$), sequences (*), sets ($\mathcal{P}$),
- maybe types ($D? = D + \{?\}$),
- an error element $\top$ for strict error propagation,
- impure constructs to generate fresh variables etc.

The error element $\top$ is regarded as an element of any type. Embedding the data types for meta-programming the application of a basic operation, e.g. for the construction of a fragment, returns $\top$ whenever the underlying operation is not defined. Evaluating a term is strict w.r.t. $\top$ with the common exception of the conditional.

To reason about (un)definedness it is assumed in the sequel that $\mathcal{DEF}(t)$ means that neither the term t is evaluated to $\top$ nor the evaluation of t diverges.

2.4 Properties of meta-programs

Certain properties of meta-programs which are useful to characterize operators and to facilitate well-founded program manipulation are considered. In the sequel the term *transformation* refers to functions on Rules. The type definition $\mathsf{Trafo} = \mathsf{Rules} \to \mathsf{Rules}$ is assumed.

The first definition concerns (α[6]-) total transformations. In general, a transformation does not need to be total because of partial fragment constructors and $\top$. However, for many operators, we can show that they are total.

Definition 3 A transformation $f \in \mathsf{Trafo}$ is α-*total* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$: $\mathcal{DEF}(f(rs))$. ♢

Let us consider now a very simple preservation property, that is to say type preservation. It is often desirable to keep the output of a transformation compatible (i.e. interchangeable as far as the profiles of the symbols are concerned) with the input.

Definition 4 A transformation $f \in \mathsf{Trafo}$ is α-*type-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \mathcal{TYPE}_{\mathsf{Rules}}(f(rs))).$$

♢

Another simple preservation property is skeleton preservation. Skeleton-preservation is a valuable property in several ways. Consider for example transformations on attribute grammars, where the term skeleton corresponds (almost) to the term underlying CFG. Obviously, it is a desirable property for transformations focusing on attributes and semantic rules that they do not modify the skeleton. Moreover, the property facilitates composition based on superimposing rules with the same shape; refer to Subsection 3.3. Furthermore, skeleton preservation is necessary to be able to abstract from the structure of target programs as far as their skeletons are concerned. Our instance of weaving, for example, first accumulates a skeleton from all the components. Transformations modelling aspects may depend on the skeleton. To make sense the skeleton must be preserved during weaving.

Definition 5 A transformation $f \in \mathsf{Trafo}$ is α-*skeleton-preserving w.r.t.* $ss \in \mathcal{P}(\mathsf{Name})$ if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{SKELETON}(rs, ss) = \mathcal{SKELETON}(f(rs), ss).$$

♢

Note that if f is α-skeleton-preserving w.r.t. ss then f is also α-skeleton-preserving w.r.t. $ss' \subseteq ss$.

Let us consider a more advanced preservation property. If a given declarative program is adapted, for example, to cope with some additional computational aspects, the original computational behaviour mostly must be preserved. The semantics of the original interpreter from Figure 1, for example, is preserved by the adapted version in Figure 3 coping with I/O because a "syntactical" preservation property holds, that is to say the original interpreter can be regarded as a *projection* of the adapted interpreter where projection means that some premises and parameter positions can be removed and some occurrences of variables can be replaced by fresh variables.

Definition 6 Let be $rs, rs' \in \mathsf{Rules}$. rs is a *projection* of rs' (rs' is an *extension* of rs) if

[5] Thinking of an instance of the framework for attribute grammars, for example, uniqueness for positions of grammar symbols is a natural assumption if attributes are modelled by sorts.

[6] If some property holds only for some inputs $\alpha \subseteq \mathsf{Rules}$, the property is qualified with α.

1. $\forall \tau \in \mathcal{TYPE}_{\mathsf{Rules}}(rs) : \exists \tau' \in \mathcal{TYPE}_{\mathsf{Rules}}(rs') : \tau$ is a projection of τ', i.e.

   if $\tau' = s\ \sigma_1^{\downarrow} \times \cdots \times \sigma_{n'}^{\downarrow} \to \sigma_1^{\uparrow} \times \cdots \sigma_{m'}^{\uparrow}$

   then $\exists in_1, \ldots, in_n, out_1, \ldots, out_m$ such that

   - the in_i are pairwise distinct,
   - the out_j are pairwise distinct,
   - each $in_i \in \{1, \ldots, n'\}$,
   - each $out_j \in \{1, \ldots, m'\}$ and
   - $\tau = s\ \sigma_{in_1}^{\downarrow} \times \cdots \times \sigma_{in_n}^{\downarrow} \to \sigma_{out_1}^{\uparrow} \times \cdots \times \sigma_{out_m}^{\uparrow}$

   for $i = 1, \ldots, n$, $j = 1, \ldots, m$.

2. $|rs| = |rs'|$,

3. For every rule r in rs, there must be a rule r' in rs' as follows: $\pi_{\mathsf{Tag}}(r) = \pi_{\mathsf{Tag}}(r')$ and there is a type-consistent substitution θ such that $\theta(c) = \Pi(c')$ and $\theta(p_1) = \Pi(p'_{w_1})$, ..., $\theta(p_l) = \Pi(p'_{w_l})$, where

$$\begin{array}{rcl} c & = & \pi_{\mathsf{Conclusion}}(r) \\ c' & = & \pi_{\mathsf{Conclusion}}(r') \\ \langle p_1, \ldots, p_l \rangle & = & \pi_{\mathsf{Premise}^*}(r) \\ \langle p'_1, \ldots, p'_{l'} \rangle & = & \pi_{\mathsf{Premise}^*}(r') \end{array}$$

   and w_1, ..., w_l are some natural numbers with $1 \leq w_1 < \cdots < w_l \leq l'$ and $\Pi : \mathsf{Element} \to \mathsf{Element}$ projects parameters in elements according to (1.).

♢

Definition 7 A transformation $f \in \mathsf{Trafo}$ is *α-projection-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow rs \text{ is a projection of } f(rs).$$

♢

The term projection *preservation* makes sense here because if rs is a projection of $f(rs)$ all the projections of rs will be projections of $f(rs)$ as well. For several "sensible" instances of the framework projection-preserving transformations preserve computational behaviour because in some sense the given behaviour is extended and possibly further constrained but not adapted in any more specific sense. Kirschbaum, Sterling et al. have shown in [KSJ93], for example, that program maps—a tool similar to our projection-preserving transformations—preserve the computational behaviour of a Prolog program, if we assume that behaviour is manifested by the SLD computations of the program. It is also easy to observe that the notion is applicable to attribute grammars and natural semantics. In more general terms projections can be considered as one kind of data refinement (data transformation) [Heh93]. Obviously, not all interesting transformations are projection-preserving.

Finally, some properties concerning $\mathcal{DFC}$ are in place. Consider a transformation which preserves $\mathcal{DFC}$ in the sense that $\forall rs \in \mathsf{Rules} : \mathcal{DFC}(rs) \Rightarrow \mathcal{DFC}(f(rs))$ provided the result is defined. Such a preservation property is too weak to characterize transformations w.r.t. $\mathcal{DFC}$ because it does not apply to situations where $\mathcal{DFC}(rs)$ is not satisfied in intermediate results within a compound transformation. The following definition is useful to characterize transformations w.r.t. $\mathcal{DFC}$ in a more general sense.

Definition 8 Let be $f, f' \in \mathsf{Trafo}$, $\alpha \subseteq \mathsf{Rules}$, $\mathcal{F}$ a family $\{f_i \in \mathsf{Trafo}\}_{i \in \mathcal{I}}$ of transformations.

- The transformation f is *α-$\mathcal{DFC}$-preserving w.r.t. $\mathcal{F}$* if $\forall rs \in \alpha$: $\forall i_1, \ldots, i_n \in \mathcal{I}$:

$$\begin{array}{l} (\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f^\star(f(rs)))) \Rightarrow \\ (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f^\star(f(rs)))), \end{array}$$

  where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$.

- The *α-$\mathcal{DFC}$-preservation w.r.t. $\mathcal{F}$ for f is recovered by f'* if $\forall rs \in \alpha$: $\forall i_1, \ldots, i_n \in \mathcal{I}$:$\forall k : 1 \leq k \leq n$:

$$\begin{array}{l} (\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f'^\star(f(rs)))) \Rightarrow \\ (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f'^\star(f(rs)))), \end{array}$$

  where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$, whereas $f'^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_{k+1}} \circ f' \circ f_{i_k} \cdots \circ f_{i_1}$.

♢

Note that the above weak characterization is captured by $\mathcal{DFC}$-preservation w.r.t. $\emptyset$. Recoverability of α-$\mathcal{DFC}$-preservation for f by f' means that f and f' can be composed in a sense to construct an α-$\mathcal{DFC}$-preserving transformation. This property is useful for non-$\mathcal{DFC}$-preserving transformations because it tells that f' compensates for the undefined variables introduced by f. Note that it is slightly more general to say that the α-$\mathcal{DFC}$-preservation w.r.t. $\mathcal{F}$ for f is recovered by f' than to say that $f' \circ f$ is α-$\mathcal{DFC}$-preserving. In the paper we assume that $\mathcal{F}$ in Definition 8 corresponds to the set of transformations derivable as instances from the operators introduced in the paper.

3 An operator suite

A few operators for program manipulation are introduced below. The actual selection is example-driven, i.e. the described operators suffice to describe some semantic aspects in our running interpreter example and a certain weaver. The emphasis is on the properties of the operators facilitating semantics-preserving transformation. In [Läm98] we have investigated a more expressive operator suite including the actual definition of the operators by means of meta-programs. In fact, all the operators presented in this paper can be rigorously defined by meta-programs in the framework from the previous section.

To approach to a classification, the operators are grouped to facilitate either program transformation, program analysis or program composition.

3.1 Program transformation

Four simple operators for program transformation are introduced in the sequel. To illustrate the effect of the transformations the adaptation which is necessary to transform the simple interpreter in Figure 1 into Figure 3 coping with I/O is performed in various small steps.

3.1.1 Adding positions

The operator **Add** _ : Position → Trafo with Position = Io × Name × Sort, Io = {**Input**, **Output**} is used to add parameter positions to symbols. Consider for example the transformation **Add** $\langle$**Input**, $s, \sigma\rangle$ applied to some rules $rs \in$ Rules. All the conclusions and premises in rs are transformed systematically as follows. An element $s'(p_1, \ldots, p_n) \to (p'_1, \ldots, p'_m)$ keeps unchanged if $s \neq s' \vee \pi_{\mathsf{Sort}}(p_1) = \sigma \vee \cdots \vee \pi_{\mathsf{Sort}}(p_n) = \sigma$. Otherwise it is transformed to

$s(p_1, \ldots, p_n, v) \to (p'_1, \ldots, p'_m)$, where v is a fresh variable of sort σ.

Example 6 One trivial step in the derivation of Figure 3 is to add the parameter position of sort OUT for the symbol do. Recall that the corresponding parameters are used to accumulate the output. The corresponding transformation is represented by **Add** $\langle$**Output**, do, OUT$\rangle$. ♢

An application of **Add** is always defined. The operator preserves computational behaviour and it does not change the skeleton of the input rules. The other preservation properties from Subsection 2.4 do not hold, i.e. the type is changed, since a position is added, and $\mathcal{DFC}$ is not preserved. Because of the latter we have to look for a way to compensate for the violation of $\mathcal{DFC}$ which traces back to an application of **Add**.

Proposition 1 Let be $io \in$ Io, $s \in$ Name, $\sigma \in$ Sort. The transformation **Add** $\langle io, s, \sigma\rangle$ is total, projection-preserving and skeleton-preserving w.r.t. Name. It is neither type- nor $\mathcal{DFC}$-preserving. ♢

The operator **Add** is overloaded to add several positions at once, i.e.:

$$\begin{array}{rcl}\textbf{Add}\ \langle\langle io_1, s_1, \sigma_1\rangle, \ldots, \langle io_n, s_n, \sigma_n\rangle\rangle & = & \textbf{Add}\ \langle io_n, s_n, \sigma_n\rangle \\ & \circ & \cdots \\ & \circ & \textbf{Add}\ \langle io_1, s_1, \sigma_1\rangle\end{array}$$

Moreover, an auxiliary operator **Positions** _ **For** _ **Of Sort** _ : Io × $\mathcal{P}$(Name) × Sort → Position* for the construction of positions all with the same Io and Sort component, i.e.:

$$\textbf{Positions}\ io\ \textbf{For}\ \{s_1, \ldots, s_n\}\ \textbf{Of Sort}\ \sigma = \langle\langle io, s_1, \sigma\rangle, \ldots, \langle io, s_n, \sigma\rangle\rangle$$

Example 7 We continue Example 6 by adding the auxiliary positions of sort IN, which are used in Figure 3 to propagate the remaining input. The following transformation adds these positions:

Add Positions Output For $\{do, eval\}$ **Of Sort** IN
∘ **Add Positions Input For** $\{do, eval\}$ **Of Sort** IN

The intermediate result reflecting the inserted fresh parameters is shown in Figure 8. Note the difference between the intermediate result and the final form in Figure 3. All the computations dealing with appending outputs and computing the empty output still have to be inserted. Moreover, the inserted positions of sort IN are not yet connected to encode the propagation of the input. ♢

3.1.2 Inserting constant computations

The operator **Default For** _ **By** _ : Sort × Name → Trafo facilitates the elimination of undefined variables by the insertion of "constant computations", i.e. premises with no inputs and one output. Consider the transformation **Default For** σ **By** s applied to the rule r. Let be $v_1, \ldots, v_n$ all the undefined variables of sort σ in r. The premises $s \to (v_1)$, …, $s \to (v_n)$ are inserted into r.

Example 8 The transformation **Default For** VUSES **By** nil is useful to add the computations for the inserted parameter of sort OUT in the rules [skip] and [assign]. The intermediate result looks as follows:

$$\frac{}{do(\mathsf{skip}, \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}, \boxed{\mathsf{IN}_2}, \boxed{\mathsf{OUT}_1})}\ [\mathsf{skip}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}, \boxed{\mathsf{IN}_2}) \to (\mathsf{ST}', \boxed{\mathsf{IN}_3}, \boxed{\mathsf{OUT}_1}) \ \wedge\ do(\mathsf{C}_2, \mathsf{ST}', \boxed{\mathsf{IN}_4}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}_5}, \boxed{\mathsf{OUT}_2})}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}'', \boxed{\mathsf{IN}_6}, \boxed{\mathsf{OUT}_3})}\ [\mathsf{concat}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \boxed{\mathsf{IN}_2}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}_3}) \ \wedge\ update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}')}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{ST}', \boxed{\mathsf{IN}_4}, \boxed{\mathsf{OUT}_1})}\ [\mathsf{assign}]$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \boxed{\mathsf{IN}_1}) \to (\mathsf{VAL}, \boxed{\mathsf{IN}_2})}\ [\mathsf{var}]$$

Figure 8: An intermediate step from Figure 1 to Figure 3

$$\frac{\boxed{nil \to \mathsf{OUT}}}{do(\mathsf{skip}, \mathsf{ST}, \mathsf{IN}_1) \to (\mathsf{ST}, \mathsf{IN}_2, \mathsf{OUT})}\ [\mathsf{skip}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}_2) \to (\mathsf{VAL}, \mathsf{IN}_3) \ \wedge\ update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{ST}') \ \wedge\ \boxed{nil \to \mathsf{OUT}}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \mathsf{IN}_1) \to (\mathsf{ST}', \mathsf{IN}_4, \mathsf{OUT})}\ [\mathsf{assign}]$$

Note the difference to the final form in Figure 3. The data flow concerning the parameter positions of sort IN still needs to be established. That is the topic of Example 9. ♢

Proposition 2 Let be $\sigma \in$ Sort, $s \in$ Name. The transformation **Default For** σ **By** s is α-total, type-preserving, α'-skeleton-preserving w.r.t. Name $\setminus \{s\}$, projection- and $\mathcal{DFC}$-preserving, where $\forall rs \in \alpha \subseteq$ Rules : $\mathcal{DEF}(\mathcal{TYP}\mathcal{E}_{\mathsf{Rules}}(rs) \sqcup \{s :\to \sigma\})$ and $\forall rs \in \alpha' \subseteq$ Rules : $s \notin \mathcal{DS}(rs)$. ♢

Proposition 3 Let be $io \in$ Io, add, $by \in$ Name, $\sigma \in$ Sort. The $\mathcal{DFC}$-preservation for **Add** $\langle io, add, \sigma\rangle$ is recovered by **Default For** σ **By** by. ♢

3.1.3 Inserting copy rules

The operator **From The Left** _ : Sort → Trafo facilitates propagation by *copying systematically defining occurrences of a certain sort to undefined variables from left to right.* In attribute grammar jargon we would say that copy rules are established. Note that an application of the operator corresponds to the insertion of a potentially unknown number of copy rules. The schema is sufficient to establish certain patterns of propagation, e.g. a bucket brigade, provided the necessary positions have been added in advance. Consider the transformation **From The Left** σ applied to the rule r. Any undefined variable v of sort σ in r is replaced by the first defining variable occurrence v' of sort σ to the left of v.

Example 9 Example 8 is continued. To establish a propagation of the input from left to right (in the sense of a proof tree) the transformation **From The Left** IN is useful. As far as the rules [skip], [assign], [var] are concerned, for example, the above transformation exactly corresponds to the missing step to arrive at the final form shown in Figure 3. The rule [concat] needs some further effort concerning the combination of the outputs returned by the premises. The corresponding adaptation is discussed in Example 10. ♢

Proposition 4 Let be $\sigma \in \mathsf{Sort}$. The transformation **From The Left** σ is total, type-preserving, skeleton-preserving w.r.t. Name, projection- and $\mathcal{DFC}$-preserving. ♢

3.1.4 Pairing unused variables

The operator **Reduce _ By _** : Sort × Name → Trafo is used to pair unused variables of a certain sort σ in a dyadic computation deriving a new defining position of sort σ. The purpose of these computations is to reduce any number > 1 of unused variables of sort σ to 1. Consider the transformation **Reduce** σ **By** s applied to the rule r. Let be $v_1, \ldots, v_n$ all the unused variables of sort σ in r (in the order of their defining occurrence in r). The computations $s(v_1, v_2) \to (v_{n+1})$, $s(v_{n+1}, v_3) \to (v_{n+2})$, ..., $s(v_{n+n-2}, v_n) \to (v_{n+n-1})$, are inserted into r, where the variables $v_{n+1}, \ldots, v_{n+n-1}$ are fresh variables of sort σ. Thus, v_{n+n-1} will be the only unused variable of sort σ in the output of the transformation.

Example 10 Example 9 is continued. The defining occurrences of sort OUT in rule [concat] can be combined by the transformation **Reduce** OUT **By** *append*. Thereby, the following intermediate form of the rule is obtained:

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}_0) \to (\mathsf{ST}', \mathsf{IN}_1, \mathsf{OUT}_1) \\ \wedge & do(\mathsf{C}_2, \mathsf{ST}', \mathsf{IN}_1) \to (\mathsf{ST}'', \mathsf{IN}_2, \mathsf{OUT}_2) \\ \wedge & \boxed{append(\mathsf{OUT}_1, \mathsf{OUT}_2) \to (\mathsf{OUT}_3)} \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}_0, \mathsf{IN}_0) \to (\mathsf{ST}'', \mathsf{IN}_2, \mathsf{OUT}_4)} \quad [\mathsf{concat}]$$

Note that the above rule is still not yet in the final form shown in Figure 3 because the variables OUT_3 and OUT_4 must be identified. This can be modelled by the transformation **From The Left** OUT. ♢

Proposition 5 Let be $\sigma \in \mathsf{Sort}$, $s \in \mathsf{Name}$. The transformation **Reduce** σ **By** s is α-total, type-preserving, α'-skeleton-preserving w.r.t. $\mathsf{Name} \setminus \{s\}$, projection- and $\mathcal{DFC}$-preserving, where $\forall rs \in \alpha \subseteq \mathsf{Rules} : \mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{s : \sigma \times \sigma \to \sigma\})$ and $\forall rs \in \alpha' \subseteq \mathsf{Rules} : s \notin \mathcal{DS}(rs)$. ♢

3.2 Program analysis

Program analysis obviously seems to be useful in AOP, since weaving of component and aspect code usually has to be controlled by properties of the components. The effect of the operator **Default**, for example, depends on the set $\mathcal{AO}(r) \setminus \mathcal{DO}(r)$ for a given rule r (i.e. the undefined variables in r), where the relations $\mathcal{AO}$ and $\mathcal{DO}$ to compute applied resp. defining occurrences should be considered as analyses here. As far as the paper is concerned a further simple analysis **From _ To _ In _** : $\mathcal{P}(\mathsf{Name}) \times \mathcal{P}(\mathsf{Name}) \times \mathsf{Skeleton} \to \mathcal{P}(\mathsf{Name})$ is needed. The auxiliary operator is concerned with taking the transitive closure of symbols in a skeleton based on reachability in the context-free sense. Taking such closures is an important tool because thereby program manipulations may abstract from the underlying skeleton of a target program.

Obviously, a skeleton $sk \in \mathsf{Skeleton}$ can be regarded as a CFG. Thus, it makes sense to consider the transitive closure $\Rightarrow^+_{sk}$ of the context-free direct derivation relation w.r.t. the grammar sk. **From** *from* **To** *to* **In** sk is assumed to compute the set of all symbols $s \in \mathsf{Name}$ satisfying the property $\exists f \in from, \exists t \in to : f \Rightarrow^+_{sk} s \Rightarrow^+_{sk} t$.

Example 11 Recall Example 6 and Example 7 which are meant to add the parameter positions for propagation and accumulation of inputs and outputs respectively. The positions used in the applications of **Add** are tuned towards the interpreter fragment in Figure 1. Now consider Figure 9 with the interpreter rules for *if*- and *while*-statements. The transformations from Example 6 and Example 7 cannot be adopted for this extension because the symbol *cond* must also contribute to propagation of the input and accumulation of the output. Let us paraphrase the applications of **Add** from Example 6 and Example 7 so that they are more generic:

λrs : Rules.
Let $sk = \mathcal{SKELETON}(rs, \emptyset)$ **In**
 Let $ss_{out} = (\mathbf{From}\ \{do\}\ \mathbf{To}\ \{do\}\ \mathbf{In}\ sk) \cup \{do\}$ **In**
 Let $ss_{in} = (\mathbf{From}\ \{do\}\ \mathbf{To}\ \{eval\}\ \mathbf{In}\ sk) \cup \{do, eval\}$ **In**
 (
  Add Positions Output For ss_{out} **Of Sort** OUT
  ∘ **Add Positions Output For** ss_{in} **Of Sort** IN
  ∘ **Add Positions Input For** ss_{in} **Of Sort** IN
) (*rs*)

Thus, it is only stated where propagation/accumulation starts and at which points access is needed. All the auxiliary symbols are derived from the skeleton of the input. ♢

$$\frac{eval(\mathsf{E}, \mathsf{ST}) \to (\mathsf{VAL}) \quad \wedge \quad cond(\mathsf{VAL}, \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')}{do(\mathsf{if}(\mathsf{E}, \mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{if}]$$

$$\frac{do(\mathsf{if}(\mathsf{E}, \mathsf{concat}(\mathsf{C}, \mathsf{while}(\mathsf{E}, \mathsf{C})), \mathsf{skip}), \mathsf{ST}) \to (\mathsf{ST}')}{do(\mathsf{while}(\mathsf{E}, \mathsf{C}), \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{while}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}) \to (\mathsf{ST}')}{cond(\mathsf{boolval}(\mathsf{true}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{true}]$$

$$\frac{do(\mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')}{cond(\mathsf{boolval}(\mathsf{false}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{false}]$$

Figure 9: *if*- and *while*-statements

3.3 Program composition

Two simple program compositions are adopted. _ ⊕ _ corresponds to a kind of *union* operator used in many frameworks, e.g. in [BMPT94, Bro93], whereas _ ⊗ _ models some kind of *pairing* (or tupling [Chi93, HITT97]) based on superimposing skeletons of the operands. Another auxiliary operator $\succ\_\prec$ for *squeezing* target programs in the sense of the identification of positions of the same sort and mode is discussed. Squeezing is a useful companion for pairing.

The following notation is needed. $rs|_{ts}$ selects the rules from $rs \in \mathsf{Rules}$ with tags in $ts \in \mathcal{P}(\mathsf{Tag})$. $\mathcal{TAGS}(rs)$ denotes the tags of the rules rs.

Let us consider the operators in detail. $rs_1 \oplus rs_2$ composes rs_1, $rs_2 \in \mathsf{Rules}$ in the sense of textual juxtaposition provided the process results in a proper fragment in the sense of Figure 5, i.e. the tags must be pairwise distinct and the types of the operands rs_1 and rs_2 must be compatible. The following slightly more flexible form is needed. $rs_1 \oplus_{ss} rs_2$ with $ss \in \mathcal{P}(\mathsf{Name})$ denotes $rs_1 \oplus rs_2'$, where $rs_2' \in [rs_2]_{ss}$ such that $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs_1) \sqcup \mathcal{TYPE}_{\mathsf{Rules}}(rs_2'))$ holds.

Example 12 The simple interpreter in Figure 1 is obviously not compatible with the interpreter rules for I/O constructs shown in Figure 2. However, the adapted version shown in Figure 3 is

compatible with Figure 2. The interpreter rules for *if*-and *while*-constructs shown in Figure 9 are compatible with the simple interpreter in Figure 1. Thus, the following properties hold:

- $\neg\mathcal{DEF}$(Figure 1 $\oplus$ Figure 2)
- $\mathcal{DEF}$(Figure 3 $\oplus$ Figure 2)
- $\mathcal{DEF}$(Figure 1 $\oplus$ Figure 9)

$\Diamond$

Proposition 6 Let be $rs_1, rs_2, rs_3 \in \mathsf{Rules}$ and $ss, ss' \in \mathcal{P}(\mathsf{Name})$. In (1.)—(3.) it is assumed that $\mathcal{DEF}(rs_1 \oplus_{ss} rs_2)$ holds.

1. $\mathcal{SKELETON}(rs_i, ss') = \mathcal{SKELETON}(y|_{ts_i}, ss')$
2. rs_i is a projection of $y|_{ts_i}$
   where $y = rs_1 \oplus_{ss} rs_2$ $ts_i = \mathcal{TAGS}(rs_i)$, for $i = 1, 2$
3. $\mathcal{DFC}(rs_1) \wedge \mathcal{DFC}(rs_2) \Rightarrow \mathcal{DFC}(rs_1 \oplus_{ss} rs_2)$
4. $rs_1 \oplus_{ss} (rs_2 \oplus_{ss} rs_3) = (rs_1 \oplus_{ss} rs_2) \oplus_{ss} rs_3$
5. $[rs_1 \oplus_{ss} rs_2]_{ss} = [rs_2 \oplus_{ss} rs_1]_{ss}$

$\Diamond$

Now let us consider pairing. $rs_1 \otimes rs_2$ composes rs_1, $rs_2 \in \mathsf{Rules}$ by superimposing conclusions and premises of rs_1 and rs_2. The parameters of superimposed elements are concatenated. This simple form of pairing is defined if the skeletons of the operands w.r.t. Name are equal. To avoid a confusion of variables from the different operands it is assumed that at least one operand is "refreshed" by means of a renaming substitution. The following more flexible variant is needed. $rs_1 \otimes_{ss} rs_2$ with $ss \in \mathcal{P}(\mathsf{Name}) \supseteq \mathcal{DS}(rs_1) \cup \mathcal{DS}(rs_2)$ superimposes all elements e with $\pi_{\mathsf{Name}}(e) \in ss$. All the other premises are adopted preserving their relative order in rs_1 and rs_2. Note that $rs_1 \otimes_{ss} rs_2$ is defined if:

- $\mathcal{SKELETON}(rs_1, ss) = \mathcal{SKELETON}(rs_2, ss)$ and
- $\forall p_1 \in \mathcal{TYPE}_{\mathsf{Rules}}(rs_1)$, $\forall p_2 \in \mathcal{TYPE}_{\mathsf{Rules}}(rs_2)$:
  $\pi_{\mathsf{Name}}(p_1) = \pi_{\mathsf{Name}}(p_2) \Rightarrow$
  $(\pi_{\mathsf{Name}}(p_1) \in ss \vee p_1 = p_2)$,
  i.e. the types of rs_1 and rs_2 must be compatible as far as symbols which are not superimposed are concerned.

Proposition 7 Let be $rs_1, rs_2, rs_3 \in \mathsf{Rules}$, $ss \in \mathcal{P}(\mathsf{Name})$. In (1.)—(3.) it is assumed that $\mathcal{DEF}(rs_1 \otimes_{ss} rs_2)$ holds.

1. $\mathcal{SKELETON}(rs_i, ss) = \mathcal{SKELETON}(y, ss)$
2. rs_i is a projection of y
   where $y = rs_1 \otimes_{ss} rs_2$ for $i = 1, 2$.
3. $\mathcal{DFC}(rs_1) \wedge \mathcal{DFC}(rs_2) \Rightarrow \mathcal{DFC}(rs_1 \otimes_{ss} rs_2)$
4. $rs_1 \otimes_{ss} (rs_2 \otimes_{ss} rs_3) = (rs_1 \otimes_{ss} rs_2) \otimes_{ss} rs_3$
5. $[rs_1 \otimes_{ss} rs_2]_{ss} = [rs_2 \otimes_{ss} rs_1]_{ss}$

$\Diamond$

The first statement is trivial. The second statement holds because to project $rs_1 \otimes_{ss} rs_2$ to rs_1 (resp. rs_2) it is sufficient to discard the parameter positions and premises arising from rs_2 (resp. rs_1). Thereby, pairing with one operand position fixed can be regarded as a projection-preserving transformation. The third statement holds because the set of defining and applied occurrences in $rs_1 \otimes_{ss} rs_2$ is just a kind of disjoint union of the corresponding sets in rs_1 and rs_2.

Finally, the operator $\succ\_\prec$ for squeezing is regarded, where squeezing means to identify parameter positions of elements of the same sort and mode. Squeezing is performed for all the conclusions and the premises in a rule. The parameters on positions of the same sort and mode (input versus output) are used to build equations. Consider for example the input positions in $s(p_1, \ldots, p_n) \rightarrow (\ldots)$. Suppose that there are some positions $1 \leq i_1 < \cdots < i_m \leq n$ of the same sort σ, i.e. $\pi_{\mathsf{Sort}}(p_{i_1}) = \sigma$, ..., $\pi_{\mathsf{Sort}}(p_{i_m}) = \sigma$. Then the following equations are derived: $p_{i_1} = p_{i_2}$, ..., $p_{i_1} = p_{i_m}$. In the same manner equations are derived for all elements for the two modes. Thus, for a given rule we get a set of equations on parameters. The solved form of the equations is computed[7], where the resulting substitution (i.e. a most general unifier is applied to the rule. Finally, all but the first position of the same sort and mode are eliminated. The following slightly more general variant is needed. $\succ rs \prec_{ss}$ with $ss \in \mathcal{P}(\mathsf{Name})$ squeezes only those elements e in rs with $\pi_{\mathsf{Name}}(e) \in ss$.

Proposition 8 The operator $\succ\_\prec\_$ is idempotent, skeleton-preserving w.r.t. Name, projection- and $\mathcal{DFC}$-preserving. It is not type-preserving. $\Diamond$

$$\frac{emptyset \rightarrow (\mathsf{VS})}{do(\mathsf{skip}) \rightarrow (\mathsf{VS})} \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1) \rightarrow (\mathsf{VS}_1) \\ \wedge & do(\mathsf{C}_2) \rightarrow (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2)) \rightarrow (\mathsf{VS})} \quad [\mathsf{concat}]$$

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \rightarrow (\mathsf{VS}_1) \\ \wedge & lhs(\mathsf{ID}) \rightarrow (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E})) \rightarrow (\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\ldots$$

$$\frac{rhs(\mathsf{ID}) \rightarrow (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID})) \rightarrow (\mathsf{VS})} \quad [\mathsf{var}]$$

$$\ldots$$

Figure 10: Accumulating variable accesses

Example 13 Consider the rules in Figure 10 covering the same skeleton as the simple interpreter in Figure 1. These rules are concerned with a kind of reflection property, that is to say with recording variable accesses. It should be assumed that the data structure used for parameters of sort VS is a pair of two lists (or multisets), where one list is used to record LHS accesses, whereas the other list is used to record RHS accesses. The accumulation of accesses is similar to the accumulation of the output. To combine Figure 1, i.e. the simple interpreter with the new functionality in Figure 10, pairing with subsequent squeezing is appropriate. Refer to Figure 11 for the result of the following composition:

$$\succ \text{Figure 1} \otimes_{\{do, eval\}} \text{Figure 10} \prec_{\{do, eval\}}$$

Note that squeezing is necessary to identify the positions related to the traversal of abstract syntactical terms. Without squeezing the rule [skip], for example, takes the following form which is in contrast to Figure 11:

$$\frac{emptyset \rightarrow (\mathsf{VS})}{do(\boxed{\mathsf{skip}, \mathsf{skip}}, \mathsf{ST}) \rightarrow (\mathsf{ST}, \mathsf{VS})} \quad [\mathsf{skip}]$$

$\Diamond$

[7] That is meant in the sense of computing most general unifiers in logic programming; refer e.g. to [NM95].

$$\frac{emptyset \to (\mathsf{VS})}{do(\mathsf{skip},\mathsf{ST}) \to (\mathsf{ST},\mathsf{VS})} \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1,\mathsf{ST}) \to (\mathsf{ST}',\mathsf{VS}_1) \\ \wedge & do(\mathsf{C}_2,\mathsf{ST}') \to (\mathsf{ST}'',\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1,\mathsf{C}_2),\mathsf{ST}) \to (\mathsf{ST}'',\mathsf{VS})} \quad [\mathsf{concat}]$$

$$\frac{\begin{array}{ll} & eval(\mathsf{E},\mathsf{ST}) \to (\mathsf{VAL},\mathsf{VS}_1) \\ \wedge & update(\mathsf{ST},\mathsf{ID},\mathsf{VAL}) \to (\mathsf{ST}') \\ \wedge & lhs(\mathsf{ID}) \to (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E}),\mathsf{ST}) \to (\mathsf{ST}',\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\frac{\begin{array}{ll} & apply(\mathsf{ST},\mathsf{ID}) \to (\mathsf{VAL}) \\ \wedge & rhs(\mathsf{ID}) \to (\mathsf{VS}) \end{array}}{eval(\mathsf{var}(\mathsf{ID}),\mathsf{ST}) \to (\mathsf{VAL},\mathsf{VS})} \quad [\mathsf{var}]$$

...

Figure 11: Squeezing and pairing Figure 1 and Figure 10

4 Aspect-oriented programming

We instantiate all the central notions of AOP. Components are open declarative programs. Aspects are implemented by program transformations. One form of weaving is represented as a certain program composition.

4.1 Properties

According to the AOP terminology properties are certain decisions a program must implement. It is hard to make that term more concrete. In terms of declarative programs covered by our framework we might think of parts of the computational behaviour, certain non-functional properties such as efficiency in some sense etc.

Example 14 For the interpreter in our running example, the following properties can be isolated:

AXIOM	starting interpretation
SEQUENCE	sequencing statements
SELECTION	*if-then-else* for statements
ITERATION	iterating statements
VARIABLE	assignments and variable expressions
DATA	basic operations
READ	reading values from the input
WRITE	writing values to the output
RECORD	recording LHS/RHS variable accesses
AST	traversal of abstract syntax
EVAL	evaluation of expressions
STORE	propagation of stores
INPUT	input propagation
OUTPUT	output accumulation
ACCESS	accumulation of variable accesses

♢

A list of properties should be a proper partitioning of design decisions. However, there is no need for the properties to be atomic in some sense. It may, for example, increase modularity to subdivide *STORE* in the above example further into initialization (like in the rule [main] in Figure 2), store transformation (like in the relation *do*) and store inspection (like in the relation *eval*).

$$\frac{do(\mathsf{C})}{prog(\mathsf{C})} \quad [\mathsf{main}]$$

Implementation of *AXIOM* meeting *AST*

$$do(\mathsf{skip}) \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{ll} & do(\mathsf{C}_1) \\ \wedge & do(\mathsf{C}_2) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1,\mathsf{C}_2))} \quad [\mathsf{concat}]$$

Implementation of *SEQUENCE* meeting *AST*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VAL}) \\ \wedge & cond(\mathsf{VAL},\mathsf{C}_1,\mathsf{C}_2) \end{array}}{do(\mathsf{if}(\mathsf{E},\mathsf{C}_1,\mathsf{C}_2))} \quad [\mathsf{if}]$$

$$\frac{do(\mathsf{C}_1)}{cond(\mathsf{boolval}(\mathsf{true}),\mathsf{C}_1,\mathsf{C}_2)} \quad [\mathsf{true}]$$

$$\frac{do(\mathsf{C}_2)}{cond(\mathsf{boolval}(\mathsf{false}),\mathsf{C}_1,\mathsf{C}_2)} \quad [\mathsf{false}]$$

Implementation of *SELECTION* meeting *AST*, *EVAL*

$$\frac{do(\mathsf{if}(\mathsf{E},\mathsf{concat}(\mathsf{C},\mathsf{while}(\mathsf{E},\mathsf{C})),\mathsf{skip}))}{do(\mathsf{while}(\mathsf{E},\mathsf{C}))} \quad [\mathsf{while}]$$

Implementation of *ITERATION* meeting *AST*

$$\frac{\begin{array}{ll} & eval(\mathsf{E},\mathsf{ST}) \to (\mathsf{VAL}) \\ \wedge & update(\mathsf{ST},\mathsf{ID},\mathsf{VAL}) \to (\mathsf{ST}') \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E}),\mathsf{ST}) \to (\mathsf{ST}')} \quad [\mathsf{assign}]$$

$$\frac{apply(\mathsf{ST},\mathsf{ID}) \to (\mathsf{VAL})}{eval(\mathsf{var}(\mathsf{ID}),\mathsf{ST}) \to (\mathsf{VAL})} \quad [\mathsf{var}]$$

Implementation of *VARIABLE* meeting *AST*, *EVAL*, *STORE*

$$\frac{\begin{array}{ll} & head(\mathsf{IN}) \to (\mathsf{VAL}) \\ \wedge & tail(\mathsf{IN}) \to (\mathsf{IN}') \end{array}}{eval(\mathsf{read},\mathsf{IN}) \to (\mathsf{VAL},\mathsf{IN}')} \quad [\mathsf{read}]$$

Implementation of *READ* meeting *AST*, *EVAL*, *INPUT*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VAL}) \\ \wedge & list(\mathsf{VAL}) \to (\mathsf{OUT}) \end{array}}{do(\mathsf{write}(\mathsf{E})) \to (\mathsf{OUT})} \quad [\mathsf{write}]$$

Implementation of *WRITE* meeting *AST*, *EVAL*, *OUTPUT*

$$\frac{\begin{array}{ll} & eval(\mathsf{E}) \to (\mathsf{VS}_1) \\ \wedge & lhs(\mathsf{ID}) \to (\mathsf{VS}_2) \\ \wedge & union(\mathsf{VS}_1,\mathsf{VS}_2) \to (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID},\mathsf{E})) \to (\mathsf{VS})} \quad [\mathsf{assign}]$$

$$\frac{rhs(\mathsf{ID}) \to (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID})) \to (\mathsf{VS})} \quad [\mathsf{var}]$$

Implementation of *RECORD* meeting *AST*, *ACCESS*

Figure 12: Components for an interpreter

4.2 Components

Properties which can be implemented by open declarative programs, i.e. as collections of rules in the sense of the data type Rules, are regarded as components. In our running example it is obvious that all the properties *AXIOM*, *SEQUENCE*, *SELECTION*, *ITERATION*, *VARIABLE*, *DATA*,

READ, *WRITE* and *RECORD* are concerned directly with language constructs and thus they can be modelled by the corresponding interpreter rules. That is not possible for the properties *STORE*, *INPUT*, *OUTPUT* and *ACCESS*. However, components *implementing* a certain property might additionally *meet* some other properties. The component *VARIABLE*, for example, obviously has to meet the property *STORE*. To achieve modularity and thereby reusability it is necessary to minimize the properties met by components. In our running example, the rules dealing with a certain language construct should abstract from other properties, especially *STORE*, *INPUT*, *OUTPUT* and *ACCESS* whenever possible. All components but *VARIABLE*, for example, are not concerned with propagation of the store at all. Much better modularity is achieved if such components do not meet *STORE*. Thus, changing the property *STORE* (e.g. the migration to the two-level model based on environments and stores) will not affect most components. Figure 12 shows the components for our running interpreter example.[8]

4.3 Aspects

Properties which are not components are regarded as aspects. Aspects are modelled by suitable functional metaprograms the performance of which is intended to add or to adapt the computational behaviour or to modify the structure or "style" of components. Taking a component which does not yet meet a given aspect, the corresponding transformation should be sufficient to qualify the component so that the aspect is met. Note that if all components meet a certain aspect, there will be no need to specify the aspect at all. According to the simple form of weaving to be proposed below we restrict ourselves to a certain kind of transformations modelling aspects. They are of type Skeleton → Trafo and they are expected to be skeleton-preserving w.r.t. the symbols defined by the components. Furthermore, they should preserve computational behaviour, e.g. in the sense of projection-preserving transformations. The skeleton which can be accumulated from a given set components is useful to control the performance of some transformations, e.g. if closures of symbols must be computed based on the operator **From** ... **To** ... **In** .... Thus, a function implementing an aspect observes a skeleton.

Figure 13 shows the transformations associated with the aspects for our running interpreter example. Note that the aspects *AST* and *EVAL* are not associated with a transformation because these are such basic aspect that all components should meet them anyway.

4.4 Weaving

A program composition to be regarded as a kind of weaving is outlined in the sequel. There are probably several approaches to weaving based on open declarative programs as components and transformations as aspects. The most restrictive assumption about the instance of weaving described in this paper is that the transformations corresponding to the aspects are skeleton-preserving. The program composition modelling weaving is described as the operator *weaver*:

[8] The implementation of *DATA* dealing with the evaluation of basic operations (arithmetics, comparisons, etc.) is omitted.

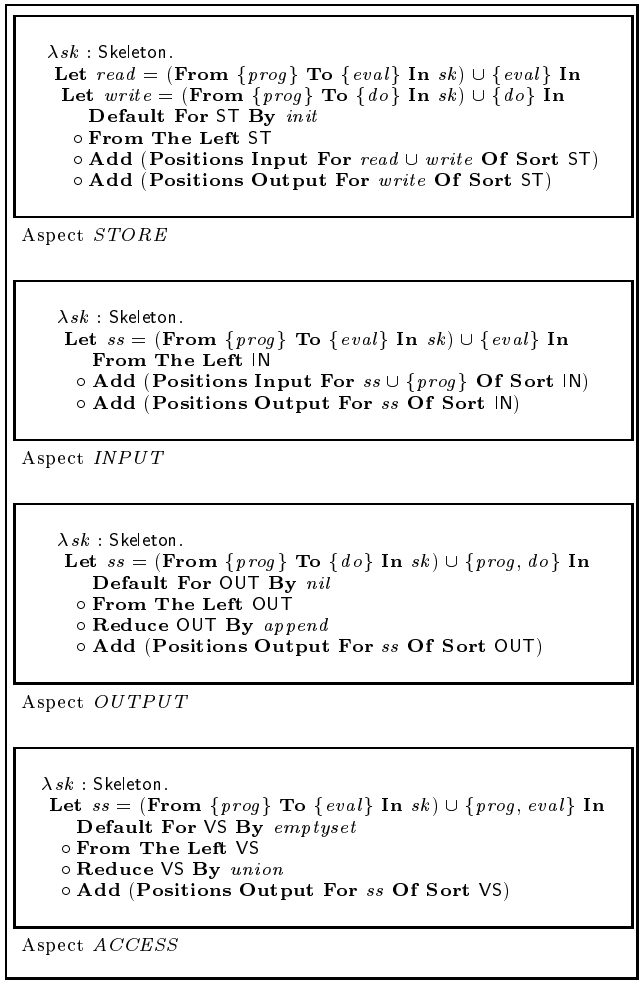

Figure 13: Aspect code for an interpreter

$$
\begin{aligned}
\mathit{weaver} &: \mathsf{Aspect}^\star \times \mathcal{P}(\mathsf{Component}) \to \mathsf{Rules} \\
\mathsf{Aspect} &= (\mathsf{Skeleton} \to \mathsf{Trafo})? \\
\mathsf{Component} &= \underbrace{\mathcal{P}(\mathcal{N})}_{\substack{\text{aspects}\\ \text{met}}} \times \underbrace{\mathcal{P}(\mathcal{N})}_{\substack{\text{irrelevant}\\ \text{aspects}}} \times \mathsf{Rules}
\end{aligned}
$$

Thus, the operator *weaver* expects two parameters, that is to say a list of aspects and a set of components. Aspects are implemented by transformations, where a maybe-type is used here for the case that no implementation can or should be provided, e.g. *AST* and *EVAL* in the running example. A component consists of some rules, the aspects met by the component and the aspects which are irrelevant for the actual component. Natural numbers are used to index the aspects. The purpose of *weaver* is to compute a combination of all the components so that all the aspects are satisfied by the result.

Definition 9 Let be $a_1, \ldots, a_n \in \mathsf{Aspect}$, $c_1, \ldots, c_m \in \mathsf{Component}$. It is assumed that $\pi_1(c_i) \cap \pi_2(c_i) = \emptyset$ and $(\pi_1(c_i) \cup \pi_2(c_i)) \subseteq \{1, \ldots, n\}$ for $i = 1, \ldots, m$. Weaving denoted by $weaver(\langle a_1, \ldots, a_n \rangle, \{c_1, \ldots, c_m\})$ is performed as follows:

1. The defined symbols ds and the skeleton sk are accumulated from the components as follows:
$$\begin{aligned} ds &= \bigcup_{i=1}^{m} \mathcal{DS}(\pi_{\mathsf{Rules}}(c_i)) \\ sk &= \bigcup_{i=1}^{m} \mathcal{SKELETON}(\pi_{\mathsf{Rules}}(c_i), ds) \end{aligned}$$
2. For each skeleton rule sh_i from sk a corresponding rule $R_i \in \mathsf{Rule}$ covering all aspects is derived as follows:
   (a) The components are filtered to obtain all the rules $r_{i,1}, \ldots, r_{i,q_i}$ with the shape sh_i. The component which contains $r_{i,j}$ is denoted by $c_{o_{i,j}}$ for $j = 1, \ldots, q_i$.
   (b) All the above rules $r_{i,1}, \ldots, r_{i,q_i}$ are paired and squeezed:
   $$\succ r_{i,1} \otimes_{ds} \cdots \otimes_{ds} r_{i,q_i} \prec_{ds}$$
   (c) The intermediate result from the previous step is transformed by f_i denoting the functional composition
   $$(a_{t_{i,w_i}}(sk)) \circ \cdots \circ (a_{t_{i,1}}(sk)),$$
   where the $t_{i,1}, \ldots, t_{i,w_i}$ are the indices of aspects not covered by the components, i.e. $\{1, \ldots, n\} \setminus \bigcup_{j=1}^{q_i} (\pi_1(c_{o_{i,j}}) \cup \pi_2(c_{o_{i,j}}))$.
3. The derived rules are composed:
$$\{R_1\} \oplus_{ds} \cdots \oplus_{ds} \{R_{|sk|}\}$$

♢

Note that the above composition fails if the union in step (1.) does not define a proper skeleton or if the aspects needed in step (2. (c)) are not proper transformations, i.e. $a_{t_{i,1}} =? \vee \cdots \vee a_{t_{i,w_i}} =?$. Step (3.) might also fail because of incompatible types. Note also that the aspects should be skeleton-preserving w.r.t. ss. Otherwise it does not make sense to accumulate a skeleton from the components and to observe this skeleton in step (2. (c)).

It is straightforward to represent the function *weaver* in our functional meta-programming framework.

Example 15 An interpreter covering the constructs from the components in Figure 12 and meeting the aspects in Figure 13 is derived by weaving as follows:

$$\begin{aligned} weaver \quad (\quad & \langle ?, ?, STORE, INPUT, OUTPUT, ACCESS \rangle, \\ & \{\langle \{1\}, \{2\}, AXIOM \rangle, \\ & \ \langle \{1\}, \{2\}, SEQUENCE \rangle, \\ & \ \langle \{1,2\}, \{\}, SELECTION \rangle, \\ & \ \langle \{1\}, \{2\}, ITERATION \rangle \\ & \ \langle \{1,2,3\}, \{\}, VARIABLE \rangle, \\ & \ \langle \{1,2,4\}, \{\}, READ \rangle, \\ & \ \langle \{1,2,5\}, \{\}, WRITE \rangle, \\ & \ \langle \{1,6\}, \{\}, RECORD \rangle \\ & \} \\ ) & \end{aligned}$$

The resulting "tangled" code is shown in Figure 14. The two question marks in the above application correspond to the aspects *AST* and *EVAL*. Note also that the aspect *EVAL* is irrelevant for the components *AXIOM*, *SEQUENCE* and *ITERATION* since they are not concerned with expression evaluation. The indices correspond otherwise to the captions in Figure 12. ♢

$$\frac{init \rightarrow (\mathsf{ST}) \quad \wedge \quad do(\mathsf{C}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{prog(\mathsf{C}, \mathsf{IN}) \rightarrow (\mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{main}]$$

$$\frac{nil \rightarrow (\mathsf{OUT}) \quad \wedge \quad emptyset \rightarrow (\mathsf{VS})}{do(\mathsf{skip}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}, \mathsf{IN}, \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{skip}]$$

$$\frac{\begin{array}{l} do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}_1, \mathsf{VS}_1) \\ \wedge \quad do(\mathsf{C}_2, \mathsf{ST}', \mathsf{IN}') \rightarrow (\mathsf{ST}'', \mathsf{IN}'', \mathsf{OUT}_2, \mathsf{VS}_2) \\ \wedge \quad append(\mathsf{OUT}_1, \mathsf{OUT}_2) \rightarrow (\mathsf{OUT}) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{concat}(\mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}'', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{concat}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}_1) \\ \wedge \quad cond(\mathsf{VAL}, \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}') \rightarrow (\mathsf{ST}', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS}_2) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{if}(\mathsf{E}, \mathsf{C}_1, \mathsf{C}_2), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}'', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{if}]$$

$$\frac{\begin{array}{l} do(\mathsf{if}(\mathsf{E}, \mathsf{concat}(\mathsf{C}, \mathsf{while}(\mathsf{E}, \mathsf{C})), \mathsf{skip}), \mathsf{ST}, \mathsf{IN}) \\ \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS}) \end{array}}{do(\mathsf{while}(\mathsf{E}, \mathsf{C}), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{while}]$$

$$\frac{\begin{array}{l} eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}_1) \\ \wedge \quad update(\mathsf{ST}, \mathsf{ID}, \mathsf{VAL}) \rightarrow (\mathsf{ST}') \\ \wedge \quad lhs(\mathsf{ID}) \rightarrow (\mathsf{VS}_2) \\ \wedge \quad union(\mathsf{VS}_1, \mathsf{VS}_2) \rightarrow (\mathsf{VS}) \end{array}}{do(\mathsf{assign}(\mathsf{ID}, \mathsf{E}), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{assign}]$$

$$\frac{eval(\mathsf{E}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS}) \quad \wedge \quad list(\mathsf{VAL}) \rightarrow (\mathsf{OUT})}{do(\mathsf{write}(\mathsf{E}), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}, \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{write}]$$

$$\frac{do(\mathsf{C}_1, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{cond(\mathsf{boolval}(\mathsf{true}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{true}]$$

$$\frac{do(\mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})}{cond(\mathsf{boolval}(\mathsf{false}), \mathsf{C}_1, \mathsf{C}_2, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{ST}', \mathsf{IN}', \mathsf{OUT}, \mathsf{VS})} \quad [\mathsf{false}]$$

$$\frac{apply(\mathsf{ST}, \mathsf{ID}) \rightarrow (\mathsf{VAL}) \quad \wedge \quad rhs(\mathsf{ID}) \rightarrow (\mathsf{VS})}{eval(\mathsf{var}(\mathsf{ID}), \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{VAL}, \mathsf{IN}, \mathsf{VS})} \quad [\mathsf{var}]$$

$$\frac{\begin{array}{l} head(\mathsf{IN}) \rightarrow (\mathsf{VAL}) \\ \wedge \quad tail(\mathsf{IN}) \rightarrow (\mathsf{IN}') \\ \wedge \quad emptyset \rightarrow (\mathsf{VS}) \end{array}}{eval(\mathsf{read}, \mathsf{ST}, \mathsf{IN}) \rightarrow (\mathsf{VAL}, \mathsf{IN}', \mathsf{VS})} \quad [\mathsf{read}]$$

Figure 14: Weaving Figure 12 and Figure 13

Finally, let us state an important property of the program composition *weaver* saying that it preserves the computational behaviour of the components because each component is a projection of some rules in the result of weaving. Furthermore, a sufficient condition for the data-flow completeness of the result of weaving is given.

Proposition 9 Let be $a_1, \ldots, a_n$, $c_1, \ldots, c_m$, ds, sk as in Definition 9. y denotes $weaver(\langle a_1, \ldots, a_n \rangle, \{c_1, \ldots, c_m\})$.

1. If $a_i \neq ?$ implies $a_i(sk)$ is projection-preserving for $i = 1, \ldots, n$, and $\mathcal{DEF}(y)$, then $y|_{\mathcal{TAGS}(c_j)}$ is a projection of c_j for $j = 1, \ldots, m$;
2. If $a_i \neq ?$ implies $a_i(sk)$ is $\mathcal{DFC}$-preserving for $i = 1, \ldots, n$ and $\mathcal{DFC}(c_j)$ for $j = 1, \ldots, m$ and $\mathcal{DEF}(y)$, then $\mathcal{DFC}(y)$,

♢

Proof It is assumed that $\mathcal{DEF}(y)$ holds.

1. It is easier to observe first that every single r from some component c_k is a projection of $y|_{\mathcal{TAGS}(\{r\})}$. Since sk

is computed as the union of all skeletons of all components, there must be an i such that r will contribute to R_i according to the steps (a)—(c) in Definition 9, i.e. r will be among the $r_{i,1}, \ldots, r_{i,q_i}$ retrieved in step (a). r is a projection of R_i computed in step (b) and (c) because the expression $f_i\ (\succ r_{i,1} \otimes_{ds} \cdots \otimes_{ds} r_{i,q_i} \prec_{ds})$ can be regarded as an application of a projection-preserving transformation to r. Refer to Proposition 7 and Proposition 8 as far as pairing and squeezing are concerned. f_i is projection-preserving because it is a functional composition of projection-preserving transformations; refer to the assumption for the $a_1, \ldots, a_n$.

It remains to show that y restricted to the tags from some component c_k can be projected to the entire component c_k at once according to the requirements (1.)—(3.) in Definition 6. The relationship for the profiles stated in (1.) exists because it carries over from projecting the rules in isolation since the signature of the entire component is just the union of the signatures of all component rules and the signature of y is just the union of the signatures of all the R_i. The 1-1 correspondence of rules stated in (2.) (concerning cardinality) and (3.) (concerning tags) holds for each component rule because there is an R_i in y with the same tag as the underlying component rules because R_i is the result of applying projection-preserving transformations to the underlying rules. The substitution σ needed in (3.) carries over from projecting the rules in isolation.

2. It suffices to show that $\mathcal{DFC}$ holds for all the single rules combined in step (3.); refer to Proposition 6. Consider again the steps (a)—(c) in Definition 9 to compute each R_i. $\mathcal{DFC}$ holds for $\succ r_{i,1} \otimes \cdots \otimes r_{i,q_i} \prec$; refer to the assumption for the $c_1, \ldots, c_m$ and to Proposition 7 and Proposition 8 as far as pairing and squeezing are concerned. f_i is $\mathcal{DFC}$-preserving because it is a functional composition of $\mathcal{DFC}$-preserving transformations; refer to the assumption for the $a_1, \ldots, a_n$.

$\Diamond$

5 Concluding remarks

First, the results of this paper are concluded. Afterwards, related work is considered in some depth. Finally, a few remarks on future work are provided.

5.1 Results

We have described an instance of aspect-oriented programming. It focuses on declarative languages as far as the components are concerned. Functional meta-programs are used to implement aspects as program transformations and to perform weaving of aspects and components in the sense of a program composition. We have developed a formal framework for functional meta-programs. It is not clear at the moment how much effort is necessary for an adaptation of the approach in order to cope with non-declarative languages.

One of the primary goals of introducing aspects and considering aspect languages is to abstract away the aspects from the components. Aspect code should be independent from components and from other aspects. This goal was addressed in several respects. First, some operators such as **Default**, **From The Left** and **Reduce** abstract from concrete symbols and concrete parameter positions. The operators only deal with occurrences of variables of certain sorts. Second, in dealing with propagation, accumulation etc. the symbols contributing to the corresponding process need not to be fixed in the aspect code but they can be derived from the skeleton of all components accumulated at weaving time. On the other hand, our aspect code still contains some details about components. Dealing with propagation, for example, the nodes which need access to the propagated data must be specified. It should be possible to abstract further in that respect by accumulating more information than just the skeleton from the components. It is a subject for further research to abstract further. Different aspects can be specified by separate program transformations. That does not mean, of course, that they are independent. Our current framework does not provide any support to detect dependencies. If there is some order on the aspects it would be easy to adapt our weaving process in such a way that this order is preserved.

Another serious problem with our instance of AOP is that there is no effective means to typecheck components w.r.t. the assertions about properties implemented and met by them (refer e.g. to the captions in Figure 12). We could try to associate a kind of type constructor with each aspect. The type constructors should be derivable from the aspect code and they could be used for typechecking components.

A very attractive property of our instance of AOP is that our weaver preserves computational behaviour of the components provided that the transformations implementing aspects preserve computational behaviour.

The framework, the operator suite for program transformation, program analysis and program composition and the operator *weaver* have been implemented in $\Lambda\Delta\Lambda$ [HLR97, Läm98] with applications in formal language definition similar to the running interpreter example.

5.2 Related work

Fradet and Südholt suggest in their recent position paper [FS98] to describe aspects as static source-to-source program transformations. Before, aspects have been usually described and implemented in an ad hoc way. Our work can be seen as an instance and refinement of this proposal because we offer a detailed framework for functional meta-programs and an operator suite facilitating the well-founded derivation of transformations implementing aspects. The proposal in [FS98] does not focus on the declarative paradigm. Their is no correspondence to using effectively sorts and modes as in our proposal. Their approach to weaving is based on fixpoint computation using program transformations as a rewriting system.

The Demeter Research Group (Karl J. Lieberherr et al.) has developed an extension of object-oriented programming, that is to say adaptive (object-oriented) programming (AP) [Lie95, PPSL96]. The Demeter method proposes *class dictionaries* for defining the structure of objects and *propagation patterns* for implementing the behaviour of the objects. Our approach is similar to AP in that transformations are independent from the actual skeleton and a reachability notion is used to establish computational behaviour schematically in concrete target programs.

Monads and monad transformers are a popular tool in

functional programming and denotational semantics [Wad92, Mog89, Esp95] to achieve extensibility. Monads rely on higher-order functions. Thus, the approach is not applicable to the representatives of the declarative paradigm addressed by our framework for functional meta-programs. A more general restriction of the monadic style which it has in common with many other concepts is that monadic programming relies on a suitable parameterization. In the terminology of AOP we had somehow to anticipate some properties to find a suitable parameterization. Note that one set of monad parameters would be in general not sufficient. Furthermore, the kind of aspects implementable by monads are restricted by the monad laws. Meuter suggests in his recent position paper to consider monads as a theoretical foundation of AOP [Meu98]. Mosses' and Watt's Action semantics [Mos96] is another approach to extensible semantics descriptions. They do not resort to higher-order features, but they rather build support for many semantic concepts such as transient, scoped, stable and permanent information into the notation; refer to [Läm98] for a detailed comparison. Thereby, such concepts need not to be coded in low-level λ- and domain-notation.

Sterling's et al. *stepwise enhancement* [Lak89, KMS96] advocates developing logic programs from skeletons and techniques. Skeletons are (in contrast to our terminology) simple logic programs with a well-understood control flow, whereas techniques are common programming practices. Applying a technique to a program yields a so-called enhancement. Our framework for functional meta-programs and our program manipulations are effective means to develop and to reason about techniques. Kirschbaum, Sterling et al. have shown in [KSJ93] that program maps—a tool similar to our projection-preserving transformations—preserve the computational behaviour of a logic program, if we assume that behaviour is manifested by the SLD computations of the program. Note that our approach is generic and that we make vital use of modes and sorts. Skeletons in the sense of our framework are not used in stepwise enhancement. Finally, there is no concept corresponding to weaving. The composition of separate enhancements of the same skeleton is considered [Jai95, KSJ93] in similarity to pairing+squeezing required for weaving components with overlapping skeletons.

Definite clause grammars (DCGs) [PW80] and some variants and extensions (e.g. the extended DCG notation in [Roy90]) support an important (Prolog) programming technique, that is to say the accumulator [SS94]. Only the access to the accumulator is specified. Otherwise, programs had to be written in a way that accumulators are modelled by means of auxiliary arguments to be chained. Such an expressive power does not provide, however, a means to adapt programs in systemic ways. The schematic adaptations, for example, according to the operators **Default**, **Reduce** and $\succ\_\prec$ are not concerned with adding and chaining arguments.

In the attribute grammar (AG) community quite a few related concepts to improve modularity have been suggested. Watt's Partitioned AGs [Wat75], for example, support decomposition and a corresponding kind of composition of AG modules which is similar to our kind of composition based on pairing and squeezing.[9] Dueck and Cormack suggest a kind of AG templates [DC90]. An instance of a template w.r.t. a CFG is obtained by a matching algorithm. One serious problem of the approach is that it is impossible to abstract sufficiently from the underlying CFG (skeleton). The AG specification language *Lido* [KW94] offers a number of concepts to describe computations abstracting from the underlying CFG. In [Läm98] we show how to simulate and to improve the above mentioned and some other approaches including object-oriented concepts by meta-programming.

[9] Actually, the term *superposition* is used in several frameworks (not only in AGs) for a similar purpose.

5.3 Future work

The class of program transformations considered in the paper focuses on data-flow aspects. We should consider aspects and corresponding operators dealing with control flow.

A projection-preserving transformation can be considered as an effective means to refine[10] a target program. Obviously, one could try to adopt a more general notion of preservation of computational behaviour. In [Läm98] we illustrate, for example, non-projection-preserving transformations to interleave premises, to reschedule some data flow and to install a new sum domain. In that case reasoning about the preservation of computational behaviour must be based on specific arguments. Note that our weaver does not rely on projection-preserving transformations.

A rather severe assumption of our weaver is that weaving is performed rule-wise and the only non-local information used in the loop body of the weaver is the skeleton to deal with reachability. Other non-local information such as component type information could be taken into consideration.

Another approach to make components compatible to each other and to allow us to implement further properties is based on "structural" transformations. A simple example concerns the implementation of optimization aspects based on folding/unfolding strategies; refer e.g. to [PP94] in the context of logic programming. Other examples are concerned with the elimination of certain forms of recursion, CPS conversion and transmutation from big-step to small-step (semantics). The relationship between such adaptations and AOP has not been investigated so far. One specific question regarding these adaptations is what are the basic roles to derive the corresponding transformations.

The scope of the framework should be extended to further representatives of the declarative paradigm, e.g. higher-order functional programming. It should also be considered if the approach can be adopted for procedural (object-oriented) programming languages.

Another issue concerns the correctness of the operator implementations. Although we represent our transformations as functional programs, it is apparently not trivial to provide rigorous proofs for all the propositions we are interested in. We want to investigate what properties can be proved automatically, e.g. by using a theorem prover. Instead of considering just functions on Rules, the derivable properties can be possibly used to establish a more powerful type system for the framework. The properties might be useful to control the weaving process.

Acknowledgement

The paper has benefitted from discussions with Isabelle Attali and Günter Riedewald. Many thanks to the anonymous referees for their constructive review. Many thanks also to Jacques Malenfant and Kenichi Asai for several advices during the preparation of the final version of the paper.

[10] (Data) refinement is usually applied in *reasoning* about programs and specifications; refer e.g. [BR94, Heh93].

References

[AOP97] *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Workshop Report published in Workshop Reader of the ECOOP, Finland, Springer-Verlag*, June 1997.

[AOP98] *Position papers of the Aspect-Oriented Programming Workshop at ECOOP'98, http: // wwwtrese . cs . utwente . nl / aop-ecoop98*, July 1998.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(3):225–237, 1994.

[Boy96] Johan Boye. *Directional Types in Logic Programming*. PhD thesis, University of Linköping, 1996.

[BR94] E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1994.

[Bro93] Antonio Brogi. *Program Construction in Computational Logic*. PhD thesis, University of Pisa, 1993.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Denmark, 14-16 June 1993.

[DC90] G.D. Dueck and G.V. Cormack. Modular Attribute Grammars. *The Computer Journal*, 33(2):164–172, 1990.

[Esp95] David A. Espinosa. *Semantic Lego*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995.

[FS98] Pascal Fradet and Mario Südholt. AOP: towards a generic framework using program transformation and analysis. In AOP98 [AOP98].

[Heh93] E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data transversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, 9–11 June 1997.

[HLR97] Jörg Harm, Ralf Lämmel, and Günter Riedewald. The Language Development Laboratory ($\Lambda\Delta\Lambda$). In Magne Haveraaen and Olaf Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248, ISBN 82-7368-163-7*, pages 77–86, May 1997.

[Jai95] Ashish Jain. Projections of Logic Programs using Symbol Mappings. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, June 13-16, 1995, Tokyo, Japan*. MIT Press, June 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[KMS96] M. Kirschenbaum, S. Michaylov, and L.S. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. In *Proceedings ACSC'96, Australian Computer Science Communications, 18(1)*, pages 516–524, 1996.

[KSJ93] M. Kirschenbaum, L.S. Sterling, and A. Jain. Relating logic programs via program maps. In *Annals of Mathematics and Artifical Intelligence, 8(III-IV)*, pages 229–246, 1993.

[KW94] Uwe Kastens and W.M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.

[Lak89] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement*. PhD thesis, Case Western Reserve University, 1989.

[Läm98] Ralf Lämmel. *Functional meta-programs towards reusability in the declarative paradigm*. PhD thesis, University of Rostock, Department of Computer Science, 1998. submitted.

[Lie95] Karl J. Lieberherr. *Adaptive Object-Oriented Software — The Demeter Method*. PWS Publishing Company, 1995.

[Meu98] Wolfgang De Meuter. Monads as a theoretical foundation for AOP. In AOP98 [AOP98].

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989.

[Mos96] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lect. Notes in Comp. Sci.*, pages 37–61. Springer-Verlag, 1996.

[NM95] U. Nilsson and J. Maluszynski. *Logic Programming and Prolog (2 ed)*. John Wiley, 1995.

[PP94] Alberto Pettorossi and Maurizio Proietti. Transformation of Logic Programs: Foundations and Techniques. *The Journal of Logic Programming 19, 20*, pages 261–320, 1994.

[PPSL96] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *6th European Symposium on Programming, Linköping, Sweden, April 1996, Proceedings of ESOP'96*, volume 1058, pages 280–295. Springer-Verlag, April 1996.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[Roy90] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, UC Berkeley, December 1990.

[SS94] L.S. Sterling and E.Y. Shapiro. *The Art of Prolog*. MIT Press, 1994. 2nd edition.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, January 1992.

[Wat75] David A. Watt. Modular Description of Programming Languages. Technical Report A-81-734, University of California, Berkeley, 1975.

[WM77] D.A. Watt and O.L. Madsen. Extended attribute grammars. Technical Report no. 10, University of Glasgow, July 1977.

Partial evaluation of shaped programs: experience with **FISh**

C.B. Jay
School of Computing Sciences,
University of Technology, Sydney
P.O. Box 123, Broadway NSW 2007, Australia;
email: cbj@socs.uts.edu.au

Abstract

FISh is an array-based programming language that combines imperative and functional programming styles. Static shape analysis uses partial evaluation to convert higher-order polymorphic programs into simple, efficient imperative programs. This paper explains how to compute shapes statically, and uses concrete examples to illustrate its several effects on performance.

1 Introduction

Partial evaluation uses limited information about inputs to optimise a program. Common instances are datum values, e.g. integers and booleans, and the shapes of data structures, e.g. the length of a list or the number of rows and columns of a matrix. Datum values can be used to unwind a recursion or evaluate a conditional, while shape information can be used to simplify data layout and memory management, e.g. by unboxing data. Shape information may be provided explicitly in a program, e.g. using types such as `int[2][3]`, but this approach severely limits program reuse. Conversely, polymorphic programming languages, such as ML and Haskell, tend to focus on inductive types, e.g. lists and trees, but do not provide any support for inferring shapes. Shape theory [Jay95] provides a formal account of data types as shaped entities, which supports programming with shapes. It has been used to guide the types, terms and compilation strategy of the **FISh** programming language [FISh] [JS98].

The explicit use of shapes in **FISh** supports several advantages not currently possible in other languages, namely: new forms of polymorphism, especially *polydimensionality* (the ability to apply a program to an array with an arbitrary number of dimensions) [Jay98b]; static detection of shape errors e.g. many array-bound errors [Sek98]; and, program optimisations. All of these advantages are achieved using (static) shape analysis of programs during off-line partial evaluation. This paper will focus on its use in optimisation.

The most obvious benefit of shape information is in improved memory management. This is of crucial importance in parallel and distributed programming, but is also a significant issue in sequential implementations. For example, unboxing eliminates a level of indirection in accessing data, e.g. replacing an array of pointers to floats by an array of floats, but then access to entries requires that their size be known in order to compute offsets. When the entries are of datum type then this can be inferred from the type [HM95, Ler97] but in general, if the entries are themselves structured, e.g. arrays, then type inference is insufficient, and a proper shape analysis is required. **FISh** is already able to handle polydimensional arrays, and is being extended to cope with inductive types, such as lists.

A more subtle benefit of shapes arises from improved separation of denotational and operational issues. This can be seen most clearly by comparing lists and vectors (one-dimensional arrays). It is common to distinguish these *operationally*: a vector typically indicates some combination of a named block of storage, constant access time to all entries and in-place update; while a list typically indicates a pointer to the heap, linear access time and referential transparency. Shape theory distinguishes vectors and lists *denotationally*: a vector is a list whose entries all have the same shape. For example, the entries in a vector of vectors must all have the same length, so that the whole corresponds to a matrix rather than an arbitrary list of lists. This *regularity* of vectors (and arrays generally) supports the operational features mentioned above, but they are not inherent.

FISh exploits this by allowing both assignable arrays of type `var` α and array expressions of type `exp` α. The former support assignment, and hence in-place update, while the latter can only be used once, and so may be re-used for other purposes. Conversely, one can envisage assignable lists, where each entry has different, but fixed, memory requirements. This distinction between `var` and `exp` types is inherited from Reynolds' Algol-like languages [Rey81] but the use of shape analysis means that it can be applied to structured data types as well as datum types. The relationship can be captured by the following slogan, from which the name "**FISh**" is derived:

$$\text{Functional} = \text{Imperative} + \text{Shape}$$

That is, higher-order, referentially transparent, functional programs can be constructed from efficient imperative procedures combined with shape information. The latter is used to control creation of local variables to which the procedure can be applied. Partial evaluation computes all of the shape information, reducing the higher-order functions to imperative procedures. Without further effort, this approach generates too many duplicate data structures, and pointless copying. Further optimisations, based on shape and free-variable analysis, eliminate most unnecessary structures.

A third source of efficiency is that shapes can be used by the programmer to optimise some algorithms. We will use folding (or reduction) over arrays as our example.

These benefits are augmented by an aggressive approach to function in-lining, which is the default choice for all (non-recursive) functions. This works well with the data-centric approach, and its support for while- and for-loops, where code copying is not a problem. Future versions are likely to pass some control over in-lining back to the programmer.

Aspects of these techniques are already familiar in partial evaluation. Shape theory provides a unified framework which selects these techniques from the range currently available, and presents them in a more general form than was previously possible. In combination they allow higher-order, polymorphic programs to be converted into simple, efficient imperative programs. A variety of small-to-medium sized programs have been written in **FISh**. Typical performance of polymorphic **FISh** programs is many times faster than equivalent programs in other polymorphic functional languages, and comparable to corresponding monomorphic programs in C (the target language of the current implementation). Even where C is polymorphic, **FISh** is typically faster. For example, polymorphic quicksort (`qsort` in C) is twice as fast in **FISh** on large arrays of floats.

The sections of the paper address the following topics: introduction; review of the **FISh** language; partial evaluation in **FISh**; examples of optimisation; and, conclusions.

2 The FISh language

This section reviews the types and terms of the **FISh** language. A large amount of background material can be found at the **FISh** web-site [FISh] including an introduction to shape theory [Jay95], introductory articles on **FISh** [JS98, Jay98b] a formal definition of the language, including partial evaluation and execution rules, a tutorial, sample programs and benchmarks.

2.1 Types

The raw syntax for the **FISh** types is given by

$$\begin{array}{rcl} \delta : \mathsf{D} & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{float} \mid \mathsf{char} \mid \ \ldots \\ \alpha : \mathsf{A} & ::= & X \mid \delta \mid [\alpha] \\ \sigma : \mathsf{Sh} & ::= & \tilde{}\delta \mid \#\alpha \\ \tau : \mathsf{T} & ::= & \alpha \mid \sigma \\ \theta : \mathsf{P} & ::= & U \mid \#U \mid \mathsf{comm} \mid \mathsf{var}\ \alpha \mid \mathsf{exp}\ \tau \mid \theta \to \theta \\ \phi : \mathsf{S} & ::= & \theta \mid \forall X : \mathsf{A}.\ \phi \mid \forall U : \mathsf{P}.\ \phi \end{array}$$

Following after Reynold's account of Algol-like languages [Rey81, OT97] **FISh** distinguishes the *data types* (meta-variable τ), which represent storable values, from the *phrase types* (meta-variable θ), which represent meaningful program fragments. The data types are further divided into the *array types* (meta-variable α) which are used to store arrays of data, and the *shape types* or *static types* (meta-variable σ) whose values are computed during compilation. These are used for static constants, and for describing the shape or structure of arrays, e.g. how many atoms of data an array will hold.

The array types are either *array types variables* (meta-variables $X, Y : \mathsf{A}$), *datum types* (meta-variable δ) or *arrays* $[\alpha]$ of α. Datum types represent atoms of data; currently, **FISh** supports datum types for integers int, booleans bool, reals or floats, float, and characters, char. The array type $[\alpha]$ represents regular arrays of α's. Here *regular* means that the arrays are finite-dimensional, rectangular, and their entries all have the same shape. For example the entries in an array of type $[[\mathsf{float}]]$ must all be arrays that have the same number of dimensions, i.e. the same *rank*, and size in each dimension. This means that array programs are able to act on arrays of arbitrary rank and size, i.e. are *polydimensional* programs.

Every datum type δ has a corresponding shape type, called $\tilde{}\delta$, whose values are computed statically, as compile-time constants. This distinction is similar to that in two-level languages, as in [NN92, BM97]. Here are some typical uses. The type $\mathsf{size} = \tilde{}\mathsf{int}$ of sizes is used to represent the length, or size, of an array in a given dimension. The type $\mathsf{fact} = \tilde{}\mathsf{bool}$ is used for static booleans, or *facts*, which are useful for expressing properties of shapes required during compilation. The type $\mathsf{cost} = \tilde{}\mathsf{float}$ is used for static floats, useful for static cost analysis. The type $\mathsf{mark} = \tilde{}\mathsf{char}$ may be used for labels.

The other shape types are of the form $\#\alpha$ which represents the shapes of arrays of type α. The values of such a type correspond to lists of sizes (one for each dimension, outermost first) paired with the common shape of the array entries. These types of array shapes are a new idea. Partial evaluation of an array of type α will include the complete evaluation of its shape of type $\#\alpha$ without any explicit separation of inputs into static and dynamic parts.

Take care not to confuse $\tilde{}\delta$ and $\#\delta$. The former type has many values, one for each value of type δ, representing sizes, facts, etc. The latter has only one value, representing the common shape of all δ-values. For example, $\tilde{}3 : \mathsf{size}$ compared to $\mathsf{int\_shape} : \#[\mathsf{int}]$.

Many of the type distinctions above originate in the semantics of arrays introduced in [Jay94] and further developed in [Jay99]. However, their motivation from a programming perspective is not so compelling. Future versions of **FISh** may simplify the type system, and hence the term structure, but this will produce fresh semantic challenges.

Now let us consider the phrase types. Phrase type variables (meta-variable $U : \mathsf{P}$) are used to express polymorphism. Each such has a *shape* $\#U$ (see below).

The type comm of commands represents operations that modify the store, such as assignments.

Data types are used to construct phrase types in two distinct ways. Each array type α yields a type $\mathsf{var}\ \alpha$ of *array variables* of type α. Terms of this type have mutable values. Each data type τ yields a type $\mathsf{exp}\ \tau$ of *expressions* of type τ whose values are immutable. Array variables represent stored quantities, much as reference types do in ML.

Unlike earlier Algol-like languages, which could only store atomic data, **FISh** also supports storable arrays. Consequently, one is able to define polymorphic array operations, such as mapping and reducing, which work for arrays of arbitrary shape, without having to fix the shape in advance. This appears to be in conflict with the well-known incompatibility of references and polymorphism ([Tof88] and also [OK93]) but in **FISh** all polymorphism is instantiated statically, before execution.

The *function type* $\theta_1 \to \theta_2$ represents functions from θ_1 to θ_2. When $\theta_2 = \mathsf{comm}$ the result is a procedure. A *ground type* is a phrase type which is not a function type.

Note that although **FISh** supports functions of arbitrarily high type, and that functions are first-class citizens as phrases (i.e. they can be passed as arguments to functions, and returned as results) they are not storable values because their shape, and hence their storage requirements, are un-

known. In particular, the shape of a function is a function (of shapes) for which no equality test is available. Hence the regularity condition for array entries cannot be established.

Every phrase type θ has an associated phrase type shp θ (or $\#\theta$) which is its *shape* (see the language definition for details). The key point for our discussion is that the shape of a function is a function of shapes, i.e.

$$\#(\theta_0 \to \theta_1) = \#\theta_0 \to \#\theta_1$$

This property of the types reflects the idea that the shapes of all data structures can be computed statically, e.g. if f : exp [int] $\to$ exp [α] is a function on arrays of integers and a is such an array then the shape of $f\ a$ is $\#(f\ a) = \#f\ \#a$ which can be computed from the knowledge of f and the shape of a.

Also, commands are not allowed to change the shape of the store, and hence all well-shaped commands have the same shape which is, by convention, the true fact ~true.

FISh supports Hindley-Milner style polymorphism using *type schemes* (meta-variable ϕ) obtained by quantifying over array and phrase type variables. The scheme $\forall X : \mathsf{A}.\phi$ represents quantification by an array type variable X and $\forall U : \mathsf{P}.\phi$ represents quantification by a phrase type variable U.

2.2 Terms

The raw syntax for **FISh** terms is the same as that for the Hindley-Milner type system:

$$t ::= x \mid c \mid \lambda x.t \mid t\ t \mid t \text{ where } x = t$$

except that where-expressions are preferred over let-expressions as evaluation will be call-by-name, not by value. x and y range over term variables. c ranges over constants. Type inference follows a modified version of the standard algorithm W [Mil78].

The **FISh** constants are given in Figure 1. They are arranged in the families, according to the kind of their result type. Binary datum operations are written infix.

Commands skip is the command that does nothing. abort terminates computation. assign $x\ t$ or $x := t$ updates the value of the array variable x to be t. The command seq $C_0\ C_1$ or $C_0; C_1$ performs the command C_0 and then C_1. The command cond $b\ C_0\ C_1$ or

$$\text{if } b \text{ then } C_0 \text{ else } C_1$$

is a conditional, branching according to the value of the boolean expression b. The for-loop forall $m\ n\ f$ or

$$\text{for } m \le i < n \text{ do } f\ i \text{ done}$$

iterates the command $f\ i$ as i ranges over the integers from m to $n-1$. Similarly, whiletrue $b\ C$ or

$$\text{while } b \text{ do } C \text{ done}$$

is a while loop. fix is a fixpoint constructor for the command type. The *command block* newvar $sh\ f$ or

$$\text{new } \#x = sh \text{ in } f\ x \text{ end}$$

introduces a local variable x of shape sh, executes the command $f\ x$ and then de-allocates x. Note that while it is necessary to supply the shape of a newly declared variable, it is not necessary to initialise its entries. output takes an array expression and returns a command. Its intended action is to compute the value of the expression, and output the result as a side effect.

Figure 1: **FISh** Constants

Commands

skip	:	comm
abort	:	comm
assign	:	$\forall X$: A. var $X \to$ exp $X \to$ comm
seq	:	comm $\to$ comm $\to$ comm
cond	:	exp bool $\to$ comm $\to$ comm $\to$ comm
forall	:	exp int $\to$ exp int $\to$ (exp int $\to$ comm) $\to$ comm
whiletrue	:	exp bool $\to$ comm $\to$ comm
fix	:	(comm $\to$ comm) $\to$ comm
newvar	:	$\forall X$: A. exp $\#X \to$ (var $X \to$ comm) $\to$ comm
output	:	$\forall X$: A. exp $X \to$ comm

Array variables

get	:	$\forall X$: A. var $[X] \to$ var X
sub	:	$\forall X$: A. var $[X] \to$ exp int $\to$ var $[X]$

Essential datum constants

n{int}	:	exp int for every integer n
$+${int, int, int}	:	exp int $\to$ exp int $\to$ exp int
$=${int, int, int}	:	exp int $\to$ exp int $\to$ exp bool
true{bool}, false{bool}	:	exp bool

Array expressions

var2exp	:	$\forall X$: A. var $X \to$ exp X
$d\{\delta_0, \ldots, \delta_k\}$	:	exp $\delta_0 \to \ldots \to$ exp $\delta_{k-1} \to$ exp δ_k
getexp	:	$\forall X$: A. exp $[X] \to$ exp X
subexp	:	$\forall X$: A. exp int $\to$ exp $[X] \to$ exp $[X]$
condexp	:	$\forall X$: A. exp bool $\to$ exp $X \to$ exp $X \to$ exp X
newexp	:	$\forall X$: A. exp $\#X \to$ (var $X \to$ comm) $\to$ exp X
dyn{δ}	:	exp ~$\delta \to$ exp δ

Shape expressions

~$d\{\delta_0, \ldots, \delta_k\}$	:	exp ~$\delta_0 \to \ldots \to$ exp ~$\delta_{k-1} \to$ exp ~δ_k
δ_shape	:	exp $\#\delta$
zerodim	:	$\forall X$: A. $\#X \to \#[X]$
succdim	:	$\forall X$: A. exp size $\to \#[X] \to \#[X]$
undim	:	$\forall X$: A. $\#[X] \to \#X$
lendim	:	$\forall X$: A. $\#[X] \to$ exp size
preddim	:	$\forall X$: A. $\#[X] \to \#[X]$
numdim	:	$\forall X$: A. $\#[X] \to$ exp size
equal	:	$\forall X$: A. $\#X \to \#X \to$ exp fact

Phrase polymorphic constants

condsh	:	$\forall U$: P. exp fact $\to U \to U \to U$
primrec	:	$\forall U$: P. (exp size $\to U \to U$) $\to U \to$ exp size $\to U$
error	:	$\forall U$: P. U
shape	:	$\forall U$: P. $U \to \#U$

Array variables The unique entry of a zero-dimensional array is named by get. Similarly, sub $x\ i$ names the variable which is the ith subarray of x (i is the *index*). For

example, if x is a matrix then the variable y given by sub x i is a vector, and sub y j is a zero-dimensional array, whose unique entry is named by applying get. We write

$$A[i_0, i_1, \ldots, i_k]$$

for get (sub (... (sub A i_0) ... i_k)).

The *primitive array variables* are those whose construction only uses primitive expressions of integer type (see next paragraph) as indices. All others are *civilised* array variables.

Let x be an occurrence of an array variable in a term. It is *assigned* if its immediate context is assign x t and is *evaluated* if its immediate context is var2exp x (see next paragraph).

Datum constants Datum constants are expressions $d\{\delta\}$: exp δ and datum operations $d\{\delta_0, \ldots, \delta_k\}$: exp $\delta_0 \to \ldots \to$ exp $\delta_{k-1} \to$ exp δ_k used to represent datum values and operations . We will often write d for $d\{\delta_0, \ldots, \delta_k\}$ when the choice of datum types is clear. Also binary operations may be written infix, e.g. $t_1 + t_2$ for $+$ t_1 t_2. The precise choice of operations does not materially affect the language design. Only those specifically required below are included in the figure.

Array expressions Each array variable x has a value given by the expression var2exp x also written as $!x$. Datum constants may be used to construct array expressions. getexp and subexp are expression analogues of get and sub. The conditional expression condexp b t_1 t_2 or

$$\text{ife } b \text{ then } t_1 \text{ else } t_2$$

evaluates t_1 if b is true, and t_2 if b is false. The needs of shape analysis impose a side-condition on the formation of such terms: both t_i must have the same shape, which is then the inferred shape of the whole expression. The *expression block* newexp sh f or

$$\text{new } \#x = sh \text{ in } f\ x \text{ return } x$$

is like a command block except that the value of the local variable x is returned before x is de-allocated.

The constants var2exp and datum constants (both expressions and functions) are called *primitive data constants*. Expressions built from these terms, term variables of type exp δ and primitive array variables are called *primitive expressions*. The constants getexp, subexp, newexp and condexp are the *civilised expression constants*.

For each datum type δ there is a coercion from static to dynamic values:

$$\text{dyn}\{\delta\} : \text{exp } \tilde{}\delta \to \text{exp } \delta.$$

Shape expressions Every datum constant d has a corresponding shape constant $\tilde{}d$. For example $\tilde{}+$ is addition on sizes. Every value of datum type δ has the same shape, given by δ_shape : exp $\#\delta$. For example, every integer n has shape int_shape. Note that, by contrast, the shape of $\tilde{}n$ is $\tilde{}n$. That is, values of shape type are their own shape.

There are six constants which manipulate array shapes. zerodim sh is a constructor that takes an array shape sh as argument, and returns the shape of a 0-dimensional array whose sole entry has shape sh. succdim is a constructor that takes a size $\tilde{}n$ and an array shape sh, and returns another array shape, of one higher dimension, whose size in that dimension is $\tilde{}n$ and whose subarrays all have shape sh. For example, succdim $\tilde{}3$ int_shape is the shape of a vector of integers of length three. A more convenient syntax for array shapes represents zerodim by a colon and succdim's by a comma separated list of integers, enclosed in braces. For example, $\{2, 3 :$ int_shape$\}$ denotes

$$\text{succdim } \tilde{}2 \text{ (succdim } \tilde{}3 \text{ (zerodim int\_shape))}$$

which is the shape of a 2×3 matrix of integers.

undim is a selector corresponding to zerodim. Similarly, lendim and preddim are selectors corresponding to succdim. Finally, the selector numdim determines the number of dimensions of an array, e.g. numdim $\{\tilde{}2, \tilde{}3 :$ int_shape$\}$ reduces to $\tilde{}2$. The remaining constant in this group is equal which checks for equality of shapes, returning a fact. We may write equal x y as $x \#= y$.

It will be useful below to distinguish the *shape constructors* $\tilde{}d\{\delta\}$, zerodim and succdim from the *shape destructors*, $\tilde{}d\{\delta_0, \ldots, \delta_k\}$ undim, lendim, preddim and equal. Terms constructed solely from shape constructors are called *shape values*.

Phrase polymorphic terms The *shape conditional* condsh b t_0 t_1 or

$$\text{ifsh } b \text{ then } t_0 \text{ else } t_1$$

branches according to the value of the *fact* b. As the value of b will be known during shape analysis, there is no requirement for the branches to have the same shape, as occurs in a shape conditional. The syntactic sugar

$$\text{check } b\ t$$

stands for condsh b t error. It is used extensively during shape analysis to check for errors.

$$\text{primrec } f\ x\ t$$

represents primitive recursion. If t is $\tilde{}n$ then this primitive recursion unwinds to

$$f\ \tilde{}n\ (f\ \tilde{}(n-1)(\ \ldots\ (f\ \tilde{}0\ x)\ldots))\ .$$

The term error represents shape errors, which result from, say, attempting to multiply matrices whose shapes don't match. The constant shape or # returns the shape of its argument.

We will abuse notation and allow a data type to stand for the corresponding expression type whenever the context makes this clear. Thus, we have 3 : int for 3 : exp int. Also, references to polymorphic constants will always refer to phrase polymorphic constants rather than data polymorphic ones.

3 Partial Evaluation

A **FISh** program is a closed term of type comm. (Array expressions can be converted to commands by applying output : exp $\alpha \to$ comm.) Static shape analysis reduces **FISh** programs to programs constructed in a simple sublanguage, called **Turbot**, whose raw syntax of terms is given

by

$$
\begin{array}{rcl}
t & ::= & x \mid \mathsf{skip} \mid \mathsf{abort} \mid \mathsf{assign}\ t_0\ t_1 \mid \mathsf{seq}\ t_0\ t_1 \mid \\
& & \mathsf{cond}\ t_0\ t_1\ t_2 \mid \mathsf{forall}\ t_0\ t_1\ \lambda x.t_2 \mid \mathsf{whiletrue}\ t_0\ t_1 \mid \\
& & \mathsf{fix}\ \lambda x.t \mid \mathsf{newvar}\ t_0\ \lambda x.t_1 \mid \mathsf{output}\ t \mid \\
& & \mathsf{get}\ t \mid \mathsf{sub}\ t_0\ t_1 \mid\ !t \mid d\{\delta_0 \ldots\ \delta_k\}\ t_0\ t_1\ \ldots\ t_{k-1} \mid \\
& & \tilde{}d\{\delta\} \mid \delta\_\mathsf{shape} \mid \mathsf{zerodim}\ t \mid \mathsf{succdim}\ t_0\ t_1
\end{array}
$$

where term variables x must be of type $\mathsf{exp}\ \mathsf{int}, \mathsf{var}\ \alpha$ or comm. Turbot evaluation is given by a structured, or big-step operational semantics in which commands are treated as store-transformers.

Note that **Turbot** does not support functions of higher type or phrase polymorphic constants, and expressions are limited to shape constructors and primitive data constants. The other functions and constants must be eliminated by partial evaluation. This is achieved by the reduction rules given in Figures 6 – 10. This section will survey the rules with examples of their application and further optimisations in the following section. A more detailed account can be found in the language definition.

The key goal is to compute all shapes, which necessarily involves evaluation of shape functions, i.e. beta-reduction. Rather than try to separate out the shape functions for special treatment, **FISh** in-lines all non-recursive function calls statically (Figure 7). Usually, in-lining is a mixed blessing with the benefits of closure elimination offset by the cost of code explosion [JW96, Ash97]. **FISh** avoids most of the disadvantages by promoting the use of for-and while-loops, in which code only appears once, the use of local variables whose initialisation is eager, and optimisations which eliminate unnecessary copying of data structures.

The rules for eliminating phrase polymorphic constants are given in Figure 6. This includes all explicit shape computations, resolving all shape conditionals and unwinding all primitive recursion. There is not space here to go discuss all of the explicit shape computations but let us consider two of the most interesting. The reduction

$$\#(x := t) \rightarrow^* \#x\ \#=\ \#t$$

shows that an assignment is well-shaped if both sides have the same shape. Many array-bound errors are caused by failures of this condition.

$$\#(\mathsf{ife}\ b\ \mathsf{then}\ x\ \ \mathsf{else}\ \ y) \rightarrow^* \mathsf{check}\ (\#b\ \#=\ \#b)\ \mathsf{check}\ (\#x\ \#=\ \#y)\ \#x$$

This rule reflects the requirement that both branches of an expression conditional must have the same shape. This constraint on conditionals is necessary for effective static shape analysis. Where the branches have different shapes the programmer is required to supply a condition that can be evaluated statically, using a shape conditional.

By unwinding all primitive recursions, we run the risk of code explosion, but its typical use is for supporting polydimensional programming, where recursion over the number of array dimensions is required, so that the number of iterations is usually no more than three.

Figure 8 gives rules for computing static quantities of datum type, and elimination rules for array shape destructors.

Figure 9 addresses the issue of shaping local variables. Their shape is known at creation, and these rules allow this information to be used when simplifying the body of the block, even though it contains free variables. That is, the context is allowed to carry shape information as well as type information about local variables.

Figure 10 describes reductions for simplifying array expressions. The first eight rules involve auxiliary functions vtc and vte which are used to handle local variables that appear within array indices. They are included here for completeness but will not affect the further discussion. The final four rules are used to convert expression conditionals and blocks to their command forms. Typical is the following assignment to an array variable x

```
x := new #y = sh in C return y
```

If y is of datum type, e.g. is an integer, then the returned value can be stored in a register, but if it is a proper array then it is not clear where to put its value. The solution is to expand the scope of y to contain the assignment, as in

```
new #y = sh in C ; x:= !y end
```

Note that there is no return value now, as indicated by the keyword `end`. Often there is a more efficient solution, as shown in Section 4.2.

After partial evaluation of a **FISh** program, the shape of resulting **Turbot** program is computed to check for shape errors, e.g. assigning an array of the wrong shape.

Shape analysis has some novel characteristics compared to standard partial evaluation techniques, e.g. [JGS93], as its techniques all derive from a single semantic insight. In this it is more like the parametrized partial evaluation described by Consel and Khoo [CK93] but requires even less intervention by the programmer. A fortiori, it can also be seen as a form of staged evaluation [ST97]. In **FISh**, however, the distinction between static and dynamic is based on the division between shape and data rather than an analysis of the properties of the particular program at hand. Also, it is able to work with partial information about a single input, e.g. the length of a vector, as well specialising with respect to total information about some inputs. Thus, shape analysis can be fully automatic, without requiring selection of variables to be handled statically, or code re-organisation. Nevertheless, a significant fraction of variables suitable for static treatment are either of datum type, or describe shapes of data structures, and so can be handled in **FISh**.

4 Examples

Now let us consider the impact of partial evaluation on program performance. The examples will illustrate the three effects listed in the introduction, namely, unboxing, array expressions, and explicit use of shapes.

4.1 Unboxed data: quicksort

Polymorphism is usually handled by boxing the data, i.e. by using pointers. Shape analysis determines the shape of the arguments statically, so that all data can be unboxed. Let us consider quicksort, as it is one of the few standard C library functions that is polymorphic, so that comparison becomes possible. A **FISh** program for polymorphic quicksort, of type

```
quicksort : (a -> a -> bool) -> [a] -> [a]
```

is given in Figure 3 (the `let rec` syntax is a sugared form of fix). The array type `a` can be instantiated to be any datum type, or nested array type. Nevertheless, comparisons are always made directly using the array entries.

By contrast, C's standard polymorphic quicksort function `qsort` uses pointers and typecasts to control polymorphism. An example comparison function for floats is

```
int cmp(const void *i, const void *j) {
  int res ;
  if (*(double*)i - *(double*)j > 0.0 )
  {res = 1 ;}
  else {res = -1 ;}
return res; }
```

Figure 2 shows user times for quicksort on a random array of 200,000 **FISh** floats (C doubles). Two kinds of program are tested. Monomorphic programs are specialised to handle floats, while polymorphic programs must be able to work with arbitrary data types and comparison functions. For C, the standard `qsort` function was used in the polymorphic case. This function achieves polymorphism by using pointers to locate array entries, and then de-referencing them to make the comparison. All of this creates longer, more complex programs, and also slows down execution by a factor of three. Similar problems are likely with the OCaml polymorphic program. **FISh** avoids pointer manipulations through shape analysis (and performs function inlining) so that the polymorphic program is as fast as its monomorphic one, twice as fast as `qsort`, and over six times faster than OCaml. This, in turn, is significantly faster than a corresponding Haskell program. Details of the experimental technique are given in Section 5.

	OCaml	C	**FISh**
poly	9.04	3.59	1.69
mono	2.22	1.29	–

Figure 2: User times (seconds) for quicksort (polymorphic and monomorphic) on a random array of 200,000 floats (doubles)

4.2 Array expressions: mapping

FISh supports both array variables (which can be assigned) and array values. This is counter to the approach in most programming languages, where all arrays are assignable, this being their raison d'etre. This added flexibility allows us to introduce additional optimisations on array expressions.

Consider an assignment to an array variable x

$$x := e$$

If e is the value $!y$ of some other array variable y then a bulk copy of memory (e.g. memcpy in C) is the simplest approach. This is safe because shape analysis guarantees that x and y have the same shape. A more frequent occurrence is that e is given by an expression block new $\#y = sh$ in C return y in which x does not appear free in C. Then there is no need to create the local variable y at all, merely to copy its result to x. Rather we can use x directly. The resulting optimisation

Figure 3: quicksort.fsh

```
let quicksort_pr (cmp:  exp a -> exp a -> bool)
   (array:  var [a]) =
 let rec qs m n =
   if m>=n then skip
   else
     new pivot = array[(m + n) div 2]
     and i = m
     and j = n in
       while cmp array[i] pivot do incr i done;
       while cmp pivot array[j] do decr j done;
       while i < j do
        new aux = array[i] in
          array[i] := array[j];
          array[j] := aux
        end;
        incr i; decr j;
        while cmp array[i] pivot do incr i done;
        while cmp pivot array[j] do decr j done
       done;
       (if i=j then incr i; decr j else skip);
       qs m (!j) ; qs (!i) n
     end
 in qs 0 (lendim #array -1)
;;
let quicksort cmp arg =
 new aux = arg in
   quicksort_pr cmp aux
 return aux ;;
```

is thus

$$x := \mathsf{newexp}\ sh\ f \quad > \quad \mathsf{check}\ (\#x\ \#\ =\ sh)\ (f\ x) \quad \text{if } fv(x) \cap fv(f) = \{\}.$$

In words, if x and the expression block have the same shape, and x is not free in the body of the block then use it as the local variable.

Although this optimisation looks fairly trivial, its correctness is dependent on a number of design features that are unique to **FISh**. (Previous Algol-like languages have not supported array *data types*.) First, the ability to manipulate whole arrays in this way, without using pointers into a heap, depends on shape analysis to ensure that copying occurs between structures of equal size and shape. Second, the check that x is not free in C would be inadequate if aliasing were allowed [Rey78, Rey89].

This optimisation eliminates many of the space leaks that confront implementers of functional languages, while maintaining a high degree of referential transparency in the source code (using `newexp`). The effect can be illustrated by looking at the action of polymorphic mapping

```
map : (a -> b) -> [a] -> [b]
```

on an expression block.

`map` is defined in the standard prelude for **FISh** and was explained in detail in [JS98] as a canonical application of the **FISh** slogan. It is defined as

```
proc2fun map_pr map_sh
```

When applied to a function `f` and an array expression `e` a local variable of shape `map_sh #f #e` is created and then the

procedure `map_pr f` is used to assign appropriate values to its entries. Rather than review the details of the construction let us consider an example, and see the effect of the optimisation on the resulting C code.

Here is a short **FISh** session. The `fill ...with ...` syntax allows one to build an array from its shape and a list of its entries.

Figure 4: Unoptimised C code generated for mapping

```
/* translated by fish */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include "fish.h"

int _argc;
char **_argv;

int main(int argc,char *argv[]) {
   _argc = argc;
   _argv = argv;

   { int A[2][3];
     { int B[2][3];
       { int C[2][3];
         C[0][0] = 0; C[0][1] = 1;
         C[0][2] = 2; C[1][0] = 3;
         C[1][1] = 4; C[1][2] = 5;
         memcpy(B,C,sizeof(B));
       }
       { int i;
         for (i = 0; i < 2; i++) {
           { int j;
             for (j = 0; j < 3; j++) {
               A[i][j] = (2*B[i][j]);
   }}}}}
   fish_print(_argc,_argv,INT_SHAPE,2,3,
   ARRAY_BOUNDARY,END_OF_SHAPE,(char *)A);
   }
   return 0;
}
```

```
let mat = fill {2,3:int_shape} with [0,1,2,3,4,5];;
let f x = 2*x;;
%show - assign_opt;;
let mat2 = map f mat;;
%show + assign_opt;;
let mat3 = map f mat;;
let mat4 = selfmap f mat;;
%run mat2;;
%run mat3;;
%run mat4;;
```

In each case the output is the same, namely

```
fill { 2,3 : int_shape }
with [
  0,2,4,
  6,8,10 ]
```

However, the first program has the assignment optimisation switched off, and so uses three local variables. The C program generated by the **FISh** compiler for `mat2` is given in Figure 4. The variable `C` is used to construct `mat` which is then copied to `B`. `B` holds the input to the mapping, whose result is stored in the variable `A` representing `mat2`. Note that the program for `map` has been written to ensure that the computation of `mat` is only performed once, outside the for-loops. Note, too that `memcpy` is used to copy `C` to `B`. This is perfectly safe as the shape analyser has already checked that the two variables have the same shape.

Of course, this copying is unnecessary, and is eliminated by the optimisation applied to the program for `mat3`. Its C code only has two local variables `A` and `B` representing `mat3` and `mat` respectively, with the central assignment being

```
A[i][j] = (2*B[i][j]);
```

Of course, one can object that a single variable should suffice, since the shapes of `mat` and `mat3` are the same. This can be achieved by using

$$\texttt{selfmap} : (\alpha \to \alpha) \to [\alpha] \to [\alpha]$$

If the result has the same shape as its input then it may store the result in the same location as the argument. This is the case in our example, where `selfmap` is used to define `mat4` whose central assignment is

```
A[i][j] = (2*A[i][j]);
```

Unfortunately, the current version of **FISh** does not allow the type of `selfmap` to be generalised to that of `map` (whose function argument may produce a result of different type) as the test for shape equality requires arguments of the same type. This should be generalised in future.

4.3 Shape-based optimisation: reduction versus folding

Shape analysis allows us to customise algorithms during compilation according to the shapes that arise even though the source code is fully polymorphic. For example, operations such as summing or taking the product of a list or array of numbers can be defined as a reduction using a primitive binary operations, e.g. addition or multiplication. An efficient algorithm uses a single auxiliary variable to hold all of the intermediate values. This is safe because all of the intermediate values have the same shape. Reduction is often identified with the polymorphic operation of folding of type

$$(a \to b \to a) \to a \to [b] \to a$$

However, for general data types the intermediate values of type a may have different shapes, e.g. be arrays of different lengths, so that one is forced to create fresh storage for each intermediate value. The **FISh** standard prelude supports both `reduce` and `fold` on arrays. The latter is implemented as `reduce` if all of the intermediate values have the same shape, but will create multiple storage locations on those rare occasions when it is necessary to do so. Here is a fragment of the code for `fold` taken from the **FISh** standard prelude.

```
let fold f x y  =
  if #f #x (zeroShape #y) #= #x
  then reduce f x y
  else ...
```

The shape conditional tests whether the shape of `f` applied to the shape of the auxiliary variable and the common shape of the array entries is the same as that of the auxiliary.

If the array types involved are actually datum types, e.g. `int`, then the type determines the shape, and so reduction (or quicksort) can be specialised without recourse to shape analysis, as in TIL [HM95]. However, the approach given here works for *all* data types, not just the datum types. For example, to add the columns of a matrix may be given as `fold (zipop plus)`. Type analysis would not allow any simplification, but shape analysis allows this to become a reduction.

5 Benchmarks

This section compares the run-time speed of compiled **FISh** programs with a number of other polymorphic languages for several array-based problems, especially OCAML which is one of the best of such other languages. All tests were run on a Sun SparcStation 4 running Solaris 2.5. C code for **FISh** was generated using GNU C 2.7.2 with the lowest optimization level using the `-O` flag and all floating-point variables of type `double` (64 bits). For OCAML code, we used ocamlopt, the native-code compiler, from the 1.07 distribution, using the flag `-unsafe` (eliminating arrays bounds checks), and also `-inline 100`, to enable any in-lining opportunities. OCAML also uses 64 bit floats.

As in [JS98] the times for **FISh** are often faster than OCAML, usually at least twice as fast, and sometimes significantly better than that. The results are summarised in Figure 5. Note, however, that OCAML requires all arrays to be initialised, while **FISh** does not.

We timed four kinds of array computations: mapping division by a constant across a floating-point array, reduction of addition over a floating-point array, multiplication of floating-point matrices, and quicksort of a floating-point array. None of the benchmarks includes I/O, in order to focus comparison on array computation.

Matrix multiplication used two different algorithms, here called "loops" and "semi-combinatory" (code omitted). The loops algorithm uses an assignment within three nested `for`-loops. This algorithm is the usual one written in an imperative language. The semi-combinatory algorithm closely follows the usual definition of matrix multiplication, with a double-nested `for`-loop containing an inner-product.

6 Conclusions

This paper has shown how knowledge of shapes supports a combination of higher-order polymorphic programming with efficient, imperative implementations. In particular, knowledge of shapes during compilation supports a wide range of program optimisations, such as unboxing of data, re-use of local variables and explicit uses of shape. These techniques all constitute a form of partial evaluation, but they emerge out of a single semantic approach, rather than being adapted to individual programs.

In particular, it is not necessary for the user to determine which inputs should be static and which dynamic, as this is determined from general principles. Where user intuition can yield further benefits, this can often be captured within the programming constructs of the language itself, as occurs in the conversion of `fold` into `reduce`, rather than by annotations.

All of the work described here has been implemented, with the source code made publically available, and is supported by a formal definition.

Current work is proceeding in two directions. One is to combine the ideas of **FISh** with those of Functorial ML [JBM98] to create a language that supports both array types and inductive data types. In developing this, many of the idiosyncracies of the **FISh** language appear to be falling away, leaving a simpler programming language but a more complicated semantics. If successful, this program may also reduce the distance between **FISh** and other, better known, programming languages, so that shape ideas could be incorporated within them.

The other development is that of a portable parallel version of **FISh** called **Goldfish**[JCSS97, Jay98a]. It will use shape analysis to guide data distribution and support a static cost model.

There are also many opportunities for further partial evaluation and optimisation based on shape information, e.g. the further elimination of dynamic array bound checks.

Overall, the **FISh** language demonstrates in concrete terms the benefits that can be extracted by incorporating shape ideas into the computational framework.

Acknowledgements

I would like to thank Neil Jones for his constructive criticism and support, and the anonymous referees and Gabriele Keller for helping me clarify these ideas.

References

[Ash97] J.M. Ashley. The effectiveness of flow analysis for inlining. In *Proc. 1997 ACM SIGPLAN International Conf. on Functional Programming (ICFP '97)*, pages 99–111, June 1997.

[BM97] G. Belle and E. Moggi. Typed intermediate languages for shape analysis. In P. de Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications. Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 1–10. Springer Verlag, 1997.

[CK93] C. Consel and S.C. Khoo. Parametrized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[FISh] Fish web-site. http://www-staff.socs.uts.edu.au/~cbj/FISh.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995.

[JW96] S. Jagannathan and A. Wright. Flow-directed inlining. In *Proc. ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation*, pages 193–205, 1996.

[Jay94] C.B. Jay. Matrices, monads and the fast Fourier transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80, 1994.

[Jay95] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jay98a] C.B. Jay. Costing parallel programs as a function of shapes. invited submission to Science of Computer Programming, September 1998.

[Jay98b] C.B. Jay. Poly-dimensional array programming. http://www-staff.socs.uts.edu.au/~cbj /Publications/polydimensional2.ps.gz, August 1998.

[Jay99] C.B. Jay. Denotational semantics of shape: Past, present and future. In A. Scedrov and A. Jung, editors, *Fifteenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XV) Tulane University New Orleans, LA USA April 28 - May 1, 1999: Proceedings*, Electronic Notes in Computer Science. Elsevier, 1999.

[JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, to appear.

[JCSS97] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August 1997.

[JS98] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98 Held as part of the joint european conferences on theory and practice of software, ETAPS'98 Lisbon, Portugal, March/April 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 139–53. Springer Verlag, 1998.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall International, 1993.

[Ler97] X. Leroy. The effectiveness of type-based unboxing. In *Abstracts from the 1997 Workshop on Types in Compilation (TIC97).* Boston College Computer Science Department, June 1997.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978.

[NN92] F. Nielson and H.R. Nielson. *Two-level functional languages.* Cambridge University Press, 1992.

[OK93] A. Ohori and K Kato. Semantics for communication primitives in a polymorphic language. In *POPL '93: The 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

[OT97] P.W. O'Hearn and R.D. Tennent, editors. *Algol-like Languages, Vols I and II.* Progress in Theoretical Computer Science. Birkhauser, 1997.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[Rey81] J.C. Reynolds. The essence of ALGOL. In J.W. de Bakker and J.C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland Publishing Company, 1981.

[Rey89] John C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722, Berlin, 1989. Springer-Verlag.

[Sek98] M. Sekanina. *Shape Analysis.* PhD thesis, University of Technology, Sydney, 1998.

[ST97] T. Sheard and W. Taha. Multi-stage programming with explicit annotations. In *Proceedings of PEPM '97*, 1997.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988. available as CST-52-88.

Figure 6: **FISh** reductions: phrase polymorphic constants

condsh ˜true	>	$\lambda x, y.\ x$
condsh ˜false	>	$\lambda x, y.\ y$
primrec f x ˜0	>	x
primrec f x ˜$(n+1)$	>	f ˜n (primrec f x ˜n)
error t	>	error
$c\ t_0\ ..\ t_{k-1}$ error	>	error for any combinator c except condsh, $k \neq 0$ or primrec, $j \neq 2$
$\#x$	>	sh if , $(x) = (sh, \theta)$
$\#(t\ t_1)$	>	$\#t\ \#t_1$
$\#\ (\lambda x.t_2)$	>	$\lambda y.t_3$ if $\#t_2 \rightarrow^*_{\Gamma'} t_3$ where $x \notin fv(t_3)$ and , $' =$, , $y : \#U, x : (y, U)$
#skip	>	˜true
#abort	>	˜true
#assign	>	equal
#seq	>	˜=
#cond	>	$\lambda x, y, z.$ check (equal x x) check y z
#forall	>	$\lambda x, y, z.$ check (equal x y) z int_shape
#whiletrue	>	$\lambda x, y.$ check (equal x x) y
#fix	>	$\lambda x.\ x$ ˜true
#newvar	>	$\lambda x, y.\ y\ x$
#output	>	$\lambda x.$ check (equal x x) ˜true
#get	>	undim
#sub	>	$\lambda x, y.$ check (equal y y) preddim x
$\#d\{\delta_0, \ldots, \delta_k\}$	>	$\lambda x_0.$ check (equal x_0 x_0) ... $\lambda x_{k-1}.$ check (equal x_{k-1} x_{k-1}) δ_k_shape
#getexp	>	undim
#subexp	>	$\lambda x, y.$ check (equal y y) preddim x
#condexp	>	$\lambda x, y, z.$ check (equal x x) check (equal y z) y
#newexp	>	$\lambda x, y.$ check (y x) x
#dyn$\{\delta\}$	>	$\lambda x.$ check (equal x x) δ_shape
#var2exp	>	$\lambda x.x$
#shape	>	$\lambda x.x$
#succdim	>	$\lambda x.$ check (x˜≥˜0) succdim x
$\#c$	>	c otherwise

Figure 7: **FISh** reductions: Beta and where

$(\lambda x.t)\ a$	>	$t[a/x]$
t where $x = a$	>	$t[a/x]$

Figure 8: **FISh** reductions: shape expressions

dyn$\{\delta\}$ ˜$d\{\delta\}$	>	$d\{\delta\}$
˜$d\{\delta_0, \ldots, \delta_k\}$ ˜n_0 ... ˜n_{k-1}	>	˜$p\{\delta_k\}$ where $p = d\ n_0..n_{k-1}$
undim (zerodim t)	>	t
undim (succdim s t)	>	error
lendim (zerodim t)	>	error
lendim (succdim s t)	>	s
preddim (zerodim t)	>	error
preddim (succdim s t)	>	t
numdim (zerodim t)	>	˜0
numdim (succdim s t)	>	numdim t ˜+ ˜1
equal δ_shape δ_shape	>	˜true
equal (zerodim t_0) (zerodim t_1)	>	equal t_0 t_1
equal (zerodim t_0) (succdim s_1 t_1)	>	˜false
equal (succdim s_0 t_0)(zerodim t_1)	>	˜false
equal (succdim s_0 t_0) (succdim s_1 t_1)	>	check (s_0˜= s_1) equal t_0 t_1

Figure 9: **FISh** reductions: shape contexts

newvar sh λx.error	>	error
forall t_2 t_3 λx.error	>	error
fix λx.error	>	error
newexp sh λx.error	>	error

Let , $' =$, , $x : (sh, \theta)$ and $t_0 \rightarrow_{\Gamma'} t_1$.

newvar sh $\lambda x.t_0$	$>_\Gamma$	newvar sh $\lambda x.t_1$
When $(sh, \theta) =$ (int_shape, exp int)		
forall t_2 t_3 $\lambda x.t_0$	$>_\Gamma$	forall t_2 t_3 $\lambda x.t_1$
When $(sh, \theta) =$ (˜true, comm)		
fix $\lambda x.t_0$	$>_\Gamma$	fix ($\lambda x.t_1$)
newexp sh $\lambda x.t_0$	$>_\Gamma$	newexp sh $\lambda x.t_1$

Figure 10: **FISh** reductions: data reduction

$$
\begin{array}{rcl}
\mathsf{assign}\ t\ e & > & \mathsf{vtc}\ (\lambda x.\ \mathsf{assign}\ x\ e)\ t \\
 & & \quad \text{if newexp or condexp in } t \\
!t & > & \mathsf{vte}\ (\lambda x.!x)\ t \quad \text{if newexp or condexp in } t \\
\mathsf{vtc}\ f\ y & > & f\ y \quad \text{if } y \text{ is a term variable} \\
\mathsf{vtc}\ f\ (\mathsf{get}\ t) & > & \mathsf{vtc}\ (\lambda y.\ f\ (\mathsf{get}\ y))\ t \\
\mathsf{vtc}\ f\ (\mathsf{sub}\ t\ i) & > & \mathsf{vtc}\ (\lambda y.\ \mathsf{newvar\ int\_shape}\ \lambda j. \\
 & & \quad j := i; f\ (\mathsf{sub}\ y\ j))\ t \\
\mathsf{vte}\ f\ y & > & f\ y \quad \text{if } y \text{ is a term variable} \\
\mathsf{vte}\ f\ (\mathsf{get}\ t) & > & \mathsf{vte}\ (\lambda y.\ f\ (\mathsf{get}\ y))\ t \\
\mathsf{vte}\ f\ (\mathsf{sub}\ t\ i) & > & \mathsf{vte}\ (\lambda y.\ \mathsf{newexp}\ (\#f\ (\mathsf{preddim}\ \#t)) \\
 & & \quad \lambda z.\ \mathsf{newvar\ int\_shape}\ \lambda j. \\
 & & \quad j := i; z := f\ (\mathsf{sub}\ y\ j))\ t \\
\mathsf{getexp}\ !t & > & !(\mathsf{get}\ t) \\
\mathsf{subexp}\ !t_1\ t_2 & > & !(\mathsf{sub}\ t_1\ t_2)
\end{array}
$$

Let g be a term and n be a natural number. If (g, n) is one of $(\mathsf{assign}\ t, 0)$, $(\mathsf{cond}, 2)$, $(\mathsf{forall}, 2)$, $(\mathsf{forall}\ t, 1)$, $(\mathsf{whiletrue}, 1)$ or $(\mathsf{output}, 0)$ then

$$
\begin{array}{rcl}
g\ (\mathsf{newexp}\ sh\ f)\ t_1\ \ldots\ t_n & > & \mathsf{newvar}\ sh\ \lambda x_0. \\
 & & \quad f\ x_0; g\ !x_0\ t_1\ \ldots\ t_n \\
g\ (\mathsf{condexp}\ s_0\ s_1\ s_2)\ t_1\ \ldots\ t_n & > & \mathsf{cond}\ s_0\ (g\ s_1\ t_1\ \ldots t_n) \\
 & & \quad (g\ s_2\ t_1\ \ldots t_n)
\end{array}
$$

Let h be a term and n be a natural number. If (h, n) is one of $(d\{\delta_0, \ldots, \delta_k\}\ s_0\ \ldots s_j, k-1-j)$, $(\mathsf{getexp}, 0)$, $(\mathsf{subexp}, 1)$ or $(\mathsf{subexp}\ s, 0)$ then

$$
\begin{array}{rcl}
h\ (\mathsf{newexp}\ sh\ f) & & \\
\quad t_1\ \ldots\ t_n & > & \mathsf{newexp}\ (\#h\ sh\ \#t_1\ \ldots \#t_n) \\
 & & \quad \lambda x.\ \mathsf{newvar}\ sh\ \lambda x_0. \\
 & & \quad f\ x_0; \\
 & & \quad x := h\ !x_0\ t_1\ \ldots\ t_n \\
h\ (\mathsf{condexp}\ s_0\ s_1\ s_2) & & \\
\quad t_1\ \ldots\ t_n & > & \mathsf{condexp}\ s_0\ (h\ s_1\ t_1\ \ldots t_n) \\
 & & \quad (h\ s_2\ t_1\ \ldots t_n)
\end{array}
$$

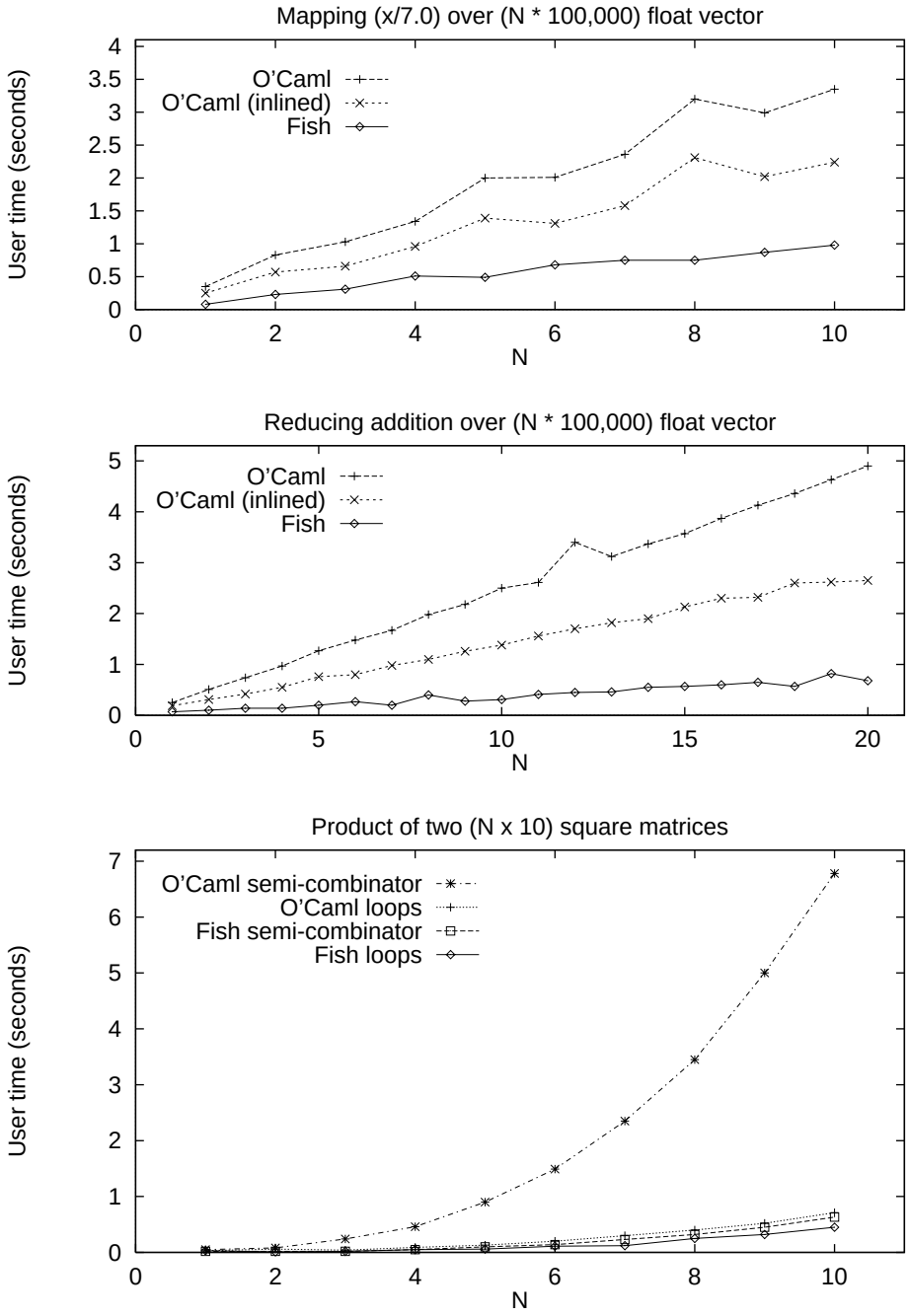

Figure 5: Benchmark results.

Rehabilitating CPS

Olin Shivers
MIT Artificial Intelligence Laboratory
shivers@ai.mit.edu

In the eighties, if one set out to write a compiler for a lambda-calculus based language, the odds are one would have chosen an intermediate representation based on continuation-passing style—"CPS." Compilers such as Orbit, Kelsey's transformational compiler, and SML/NJ all traced their heritage back to the seminal influence of Steele's Rabbit compiler, which established the CPS-as-intermediate-representation thesis. While CPS by no means had an exclusive franchise, it staked out a large niche in the functional-language arena—at many institutions, it was simply the accepted and expected representational framework.

However, at the turn of the decade, researchers began increasingly to experiment with alternative low-level, lambda-based frameworks. Representations such as nqCPS and A-normal form allowed compiler writers to factor the features provided by CPS, such as serialisation of primitive operations, and the naming of all intermediate results, without having to commit to the explicit machinery of exposed continuations. The new and interesting compilers, such as Morrisett and Tarditi's TIL compiler, were written using "direct style" lambda-calculus intermediate representations. CPS has faded from view somewhat; I am not aware of a serious CPS-based compiler being written in the last five to eight years.

Part of the reason academe's attention turned away from CPS was a series of influential papers by Felleisen and Sabry establishing some deep formal connections between the analytic power of the two frameworks. The message of these papers was that introducing explicit continuations provided no extra theoretical benefit—so why bother with them?

As another decade prepares to turn over, we have the advantage of ten years of experience working with these alternative frameworks. I suggest that it is now time to reexamine the benefits of CPS, in the light of these lessons learned.

In this talk, I will discuss the benefits of CPS from a modern perspective. I'll show some interesting examples of recent work on operating-system concurrency and transducer composition that rely critically on the ability to represent continuations explicitly. I will discuss how we can apply the lessons of the last decade to CPS, and *vice versa.* I'll address the issue of formal equivalences between continuation-based and direct-style representations, and point out limits in our current understanding and use of continuation-based compiler frameworks.

Recent BRICS Notes Series Publications

NS-99-1 **Olivier Danvy, editor.** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,*** **(San Antonio, Texas, USA, January 22–23, 1999), January 1999.**

NS-98-8 **Olivier Danvy and Peter Dybjer, editors.** ***Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,*** **(Gothenburg, Sweden, May 8–9, 1998), December 1998.**

NS-98-7 **John Power.** ***2-Categories*****. August 1998. 18 pp.**

NS-98-6 **Carsten Butz, Ulrich Kohlenbach, Søren Riis, and Glynn Winskel, editors.** ***Abstracts of the Workshop on Proof Theory and Complexity, PTAC '98,*** **(Aarhus, Denmark, August 3–7, 1998), July 1998. vi+16 pp.**

NS-98-5 **Hans Hüttel and Uwe Nestmann, editors.** ***Proceedings of the Workshop on Semantics of Objects as Processes, SOAP '98,*** **(Aalborg, Denmark, July 18, 1998), June 1998. 50 pp.**

NS-98-4 **Tiziana Margaria and Bernhard Steffen, editors.** ***Proceedings of the International Workshop on Software Tools for Technology Transfer, STTT '98,*** **(Aalborg, Denmark, July 12–13, 1998), June 1998. 86 pp.**

NS-98-3 **Nils Klarlund and Anders Møller.** MONA ***Version 1.2 — User Manual*****. June 1998. 60 pp.**

NS-98-2 **Peter D. Mosses and Uffe H. Engberg, editors.** ***Proceedings of the Workshop on Applicability of Formal Methods, AFM '98,*** **(Aarhus, Denmark, June 2, 1998), June 1998. 94 pp.**

NS-98-1 **Olivier Danvy and Peter Dybjer, editors.** ***Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,*** **(Gothenburg, Sweden, May 8–9, 1998), May 1998.**

NS-97-1 **Mogens Nielsen and Wolfgang Thomas, editors.** ***Preliminary Proceedings of the Annual Conference of the European Association for Computer Science Logic, CSL '97*** **(Aarhus, Denmark, August 23–29, 1997), August 1997. vi+432 pp.**

NS-96-15 **CoFI.** ***CASL – The CoFI Algebraic Specification Language; Tentative Design: Language Summary*****. December 1996. 34 pp.**