Some Approaches to Automatic Compiler Generation

E. David Burke
B.S.M.E, Lehigh University, 1983

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

January 1998

The thesis "Some Approaches to Automatic Compiler Generation" by E. David Burke has been examined and approved by the following Examination Committee:

John Launchbury
Associate Professor
Thesis Research Adviser

Tim Sheard
Associate Professor

Andrew Black
Professor

Dedication

To Kelly

Acknowledgements

I'll always be indebted to my parents; their passion for education, and their decency as human beings have made them the best role models I could wish for. Everything I am I owe to them. Thanks to my family: Kerri, Kelly, and Allison, for their love, support, and patience while I was in graduate school. I would like to acknowledge the other members of the "Center for Unsupervised Learning" – Rob Prouty, Mike Stupak, and Michael Gerlek, for all their help and friendship. I appreciate the guidance and financial support from my two thesis advisors, Charles Consel, and John Launchbury. Lastly, thanks to Andrew Black and Tim Sheard, for carefully commenting on the thesis drafts.

Contents

List of Figures

Abstract

Some Approaches to
Automatic Compiler Generation

E. David Burke

Supervising Professor: John Launchbury

Although it is well-known that the program optimization technique called partial evaluation can be used to do compilation, it is less well-known how the interpreters ought to be structured so that the target programs execute efficiently. Most of the existing work in this area fails to address the issue of compiling from a high-level language specification down to machine code; the resulting target code is much too inefficient, making this technique impractical for "Real World" compilation.

In this work, we address the feasibility issue by using partial evaluation to produce efficient, low-level code for the Scheme programming language, starting from the official language definition. We also detail another, complementary approach to compilation. This "calculational" approach is based on structuring the interpreter in such a way that it can be mathematically manipulated, and a compiler derived from the properties of the source language, and run-time machine. After demonstrating how both methods work, and discussing our experience in trying to efficiently compile Scheme programs, we compare and contrast the differing techniques, as well as suggesting some ideas for future research.

Chapter 1

Introduction

1.1 Opening Statment

In this chapter, we briefly discuss the concepts of compilation and, in particular, the automatic generation of compilers. This chapter provides the context in which to set our work in automatic compiler generation.

1.2 The Idea of Compilation

The very first compiler, for the programming language Fortran, was developed by John Backus and his team at IBM in the late 1950's. It is interesting to note that the original goal of this project was the elimination of programming! That is, given a mathematical formulation of the problem, a correct program would be generated to solve that problem. This goal has to be evaluated in context. At the time, only assembly language existed; because this is so close to the operational details of a machine, it turns out to be a very time-consuming and error-prone way to program. What Backus and his group hoped to do was to enable the user to input mathematical formula (the source program) that would be translated into machine code (hence the name "Fortran" from *For*mula *tran*slation).

It is also important to note that correctness was much more of a concern than efficiency (which is very ironic, given Fortran's reputation today as a very fast language); the goal of Backus and his team was to demonstrate that the very idea of compilation was feasible [1]. One way to measure the success of the project is to note that Fortran is now considered a very "low-level" language. Programming language and compiler design became legitimate fields of study due to Fortran's example, and the state of the art has advanced considerably since that time.

Once Fortran compilers had been successfully developed, it was natural to wonder whether the process of developing a compiler itself could be automated in some way. Consider this: a high-level language abstracts away details of a target machine, enabling us to write software that uses data structures and algorithms that are "natural" to the problem. The program, then, can be thought of as a somewhat higher-level specification of the problem, for which the compiler generates a correct target program. By correct, we mean that the target program produced by the compiler preserves the meaning of the source program.

We can take this process one level higher in abstraction. Why not do essentially the same thing for compilers as was done for programs? That is, specify the source language in which programs are to be written, and the target language to translate to, and automatically generate a compiler for the source-target pair. Such a tool, which automatically generates a compiler, can be thought of as a "meta-compiler" Compilers map the set of possible source programs (given a well-defined source language) to a set of target programs. The "meta-compiler" maps the set of all possible source languages (again, those that are "well-defined") to a set of compilers.

Compilers are complex objects; whereas the idea of automating the process of compiler generation was a very attractive one to computer science researchers, progress came slowly, and in pieces. As parts of the compilation process became well-understood, they could be formalized, and once formalized, amenable to automation. The first fruits of this labor were in what is usually referred to as the "front end" of compilation: the tasks of lexical and syntactic analysis. A lexer converts a program (as a string) into a stream of tokens. A parser takes a token stream, and converts it into a parse tree. Work in this area produced programs like *Lex* and *Yacc*, which, given a specification of tokens and productions, respectively, generate lexers and parsers. The conventional wisdom nowadays is that the front end of compilation is pretty much under control.

The crux of the compilation process, though, has proven to be extremely difficult to automate. This is the problem of translating the parse tree into a new, more operational representation (somewhat similar to machine language) while at the same time preserving the "meaning" of the source program. Unlike lexing and parsing, which are syntactically-driven, this translation (or "middle-end" compilation) is *semantically-driven.* This raises the question of just what is the "meaning" of a program is, and has led to a variety of methods for assigning semantics to programming languages. The most successful of all of these approaches is *denotational semantics.* In this approach, we map program text to mathematical structures, which we can then manipulate formally. With the rise of, and dissemination of information about denotational semantics in the 1970's, there was a concurrent increase in efforts in the new field of "semantics-directed compiler generation". Systems such as Mosses' *Semantic Implementation System* (SIS), and Wand's *Semantic Prototyping System* (SPS), [31, 44] demonstrated that it was indeed possible to generate compilers, given a language specification encoded with denotational semantics.

However, these, and other similar systems, have not been completely successful. Code produced by these systems tends to execute very slowly – often a couple of orders of magnitude slower than target programs translated by handwritten compilers. Not too surprisingly, these denotationally-driven approaches tend to be more successful in the context of a functional language, since both these languages, and denotational semantics itself, are computational realizations of the lamba calculus.

More recent work in the field has moved away from a strict denotational approach to automatic compiler generation. A good example of this is Mosses' Action Semantics [33], where the primitives correspond much more closely to the kinds of primitives that actually exist in target machines – fetches and updates, for example. The idea behind action semantics is that the primitives (or actions) should correspond very closely to the gradual, step-wise nature of computation. We discuss this, and other methods of semantic definition, in more detail in the next chapter.

1.3 Overview of Thesis

In this work, we investigate the use of partial evaluation in the automatic generation of compilers. It is well known that given an interpreter for a language, and a source program, a partial evaluator can compile the source program by specializing the interpreter with respect to the source program. Existing work, however, has dealt mainly with toy languages, and does not demonstrate that this technique scales up to "Real World" languages.

In our work, we write an interpreter that is an encoding of the denotational definition of the Scheme programming language that appears in the Revised^4 Report on the Algorithmic Language Scheme (R4RS) [7]. We demonstrate that it is possible, through an analysis of the basic forms of the language, to compile source programs that execute roughly as fast as hand-compiled programs. In addition, we show how a technique of making multiple passes with the partial evaluator enables us to control the structure of the compiled programs; we can iteratively expose operational details, making the code more low-level with each pass. This approach has the benefit of further modularizing the compilation process. Chapter 2 introduces partial evaluation, and Chapter 3 details our work.

Despite the semantic basis of our work, some of the binding-time improvements that we make to control the structure of residual programs are based on intuition and iterative experimentation. It would be better if these improvements were formally motivated. Another possible drawback to using partial evaluation for automatic compilation is that any correctness proof of the compiler is dependent on having a proof of correctness of the partial evaluator itself. In order to address these challenges, we look at an alternative, more formal approach to compiler generation, which we call the Calculational Approach, in which a compiler is derived from the structure of the source language. Our belief is that the two methods complement each other; the formal methods give us more confidence in the correctness of the partial evaluation approach, as well as generating useful intuitions about the structure of the source language and how it can best be compiled. In Chapter 4 we introduce the calculational approach, and in Chapter 5 we suggest how it can be used. Chapter 6 draws some conclusions, and mentions some possibilities for future research.

Chapter 2

Introduction to Partial Evaluation

2.1 Partial Evaluation

Partial evaluation is a program optimization technique that has received much attention from the research community over the last decade, because of its generality and power [13, 8, 19, 22]. The concept of partial evaluation unifies the ideas of compilation, interpretation, and program transformation.

A partial evaluator can be thought of as a "smart" interpreter. Given a program, an interpreter, and *some* of the program's inputs, a partial evaluator will produce a new program that takes the remaining inputs, and produces the same answer that the original program would have, given all the inputs. This new program is called a *specialized* version of the original program. Hence, a partial evaluator performs a "mix" of computations [23]. It executes code corresponding to the known inputs, and generates code corresponding to the unknown inputs. We call those inputs known at partial evaluation time *static*; those that are unknown are referred to as *dynamic*. We can think of partial evaluation as removing layers of interpretation corresponding to the static inputs. For this reason, partial evaluators are often described as *static semantic processors*.

Partial evaluation has already proved itself to be a practical tool for such diverse applications as scientific computation [2], neural nets [20], database queries [35], and automatic compiler generation [14, 18, 24, 26, 12, 23], the latter being of particular interest to us.

2.1.1 What a Partial Evaluator Does

The output generated by the partial evaluator, given a program and some of its inputs, is called a *residual* program, and is referred to as a *specialization* of the original program. The hope is that investment in the computation involved in the partial evaluation process will be more than made up for by a residual program that is more efficient in time and/or space than the original.

The following simple example illustrates what a partial evaluator does. Consider a program that calculates exponentials:

```
(define (power x n)
  (if (eq? n 0)
      1
      (* x (power x (- n 1)))))
```

Suppose that we know that n equals `5`, and that `x` is unknown. Hence, `n` is static, and `x` is dynamic. Feeding this program, and the input `n = 5` to a partial evaluator, we would find:

```
        (power x 5) = (* x (* x (* x (* x (* x 1)))))
```

However, it is important to note that in the case where `x` is static, and `n` is dynamic, we might not get any improvement to the original code (depending on the value of `x`; for example, a partial evaluator that "knows" some algebra might know that 1 raised to any power is 1, and specialize accordingly).

This example demonstrates one of the applications of partial evaluation to real–world code generation. During software development, there is usually a tension between the goals of efficiency on one hand, and generality and modularity on the other. Partial evaluation can be used as a way of satisfying both goals. Programs can be written in a general, modular fashion; efficient, specialized versions can be generated with a partial evaluator.

2.1.2 The Futumura Projections

One of the most significant properties of a partial evaluator is that specializing an interpreter with respect to a source program compiles the program[17]. The partial evaluator processes the static semantics of the interpreter, which is equivalent to performing all the compile-time actions, leaving the "executable code", which corresponds to run-time actions. Hence, to compile source programs for some language, we need only write an interpreter for it (given the existence of a partial evaluator for the language in which the interpreter is written). Interestingly, if the partial evaluator can take *itself* as an input, then the partial evaluator is called *self-applicable.*

A self-applicable partial evaluator offers a second method of compiling programs. The result of a self-applicable partial evaluator specializing *itself* with respect to an interpreter is a compiler for the language defined by the interpreter. Both approaches give the same result; a reason to take the second approach is that it is more efficient to generate a compiler whenever some of the programs we wish to compile share an interpreter as a language definition.

Let us consider the behavior of a partial evaluator in more detail, following the presentation given by Jones [22]. Given a partial evaluator PE for a language L, and a source program P written in L, with static inputs s, and dynamic inputs d, we feed P and s to PE, and take the resulting program, and run it on dynamic inputs d. The result is the same as if we had run P on s and d together. The following equation, which we can call the "PE equation" represents this state of affairs, where $[\![P]\!]$ indicate running a program P.

$$[\![[\![PE]\!]\ P\ s]\!]\ d \ = \ [\![P]\!]\ s\ d$$

What makes partial evaluation relevant to automatic compiler generation are the three equalities known as the *Futumura projections* [17]. Consider the PE equation; let the source program P be an interpreter, int. Let the static input s to the interpreter be a source program, sor, and the dynamic input d be the inputs to the source program, $input$. Then the PE equation becomes:

$$[\![[\![PE]\!] \ int \ sor]\!] \ input \ = \ [\![int]\!] \ sor \ input$$

We want a target machine program with the dynamic input to give the same result as we would get from running the interpreter on both the static and dynamic inputs:

$$[\![target]\!] \ input \ = \ [\![int]\!] \ sor \ input$$

Then, by the PE equation above, we have the following equality, the first Futumura projection:

$$target \ = \ [\![PE]\!] \ int \ sor$$

The second and third Futumura projections demonstrate the possibility of generating both compilers and compiler generators through the use of partial evaluation. The second Futumura projection shows us how to produce a compiler by using the partial evaluator to specialize *itself* with respect to an interpreter. This concept is called *self-application*, and requires the partial evaluator to be written in the same language that it processes.

To derive the second Futumura projection, we start with the PE equation again, this time letting the source program P be the partial evaluator PE itself, with static and dynamic inputs the interpreter int, and a source program sor, respectively. Then, by the first Futumura projection, and this version of the PE equation, we have:

$$\begin{aligned} target &= [\![PE]\!]\ int\ sor \\ &= [\![[\![PE]\!]\ PE\ int]\!]\ sor \\ &= [\![compiler]\!]\ sor \end{aligned}$$

From this derivation, we have the following equality, which is the second Futumura projection:

$$compiler = [\![PE]\!]\ PE\ int$$

The third Futumura projection gives us, in a similar fashion, the existence of a compiler generator, *cogen*, through partial evaluation:

$$cogen = [\![PE]\!]\ PE\ PE$$

When given an interpreter, *cogen* produces a compiler generator; a program that automatically translates interpreters into compilers. This result is much less intuitive than the previous two. However, we can derive this result by starting with the second Futumura projection, and applying the PE equation to the right hand side of the projection:

$$\begin{aligned} compiler &= [\![PE]\!]\ PE\ int \\ &= [\![[\![PE]\!]\ PE\ PE]\!]\ int \end{aligned}$$

Since $[\![[\![PE]\!] \ PE \ PE]\!]$ takes as input an interpreter, int, and produces a compiler, it is a compiler generator.

The Futumura projections show that partial evaluation is theoretically a very powerful and general program transformation technique. The goal of most research in the field is to make it a practical technique.

2.1.3 How a Partial Evaluator Works

We can make a distinction between *online* and *offline* partial evaluators. An online partial evaluator can be thought of as an interpreter that takes the source program, and the static inputs, and returns a target program that expects the dynamic arguments (as opposed to a standard interpreter, which expects all of its arguments at once). In contrast, in an offline partial evaluator, the partial evaluation occurs in two phases. The first phase typically takes the source program, and a description of the binding-time of its inputs, and propagates this information throughout the source program, in order to calculate whether an expression can be evaluated at partial evaluation time, or run-time. This is called a *binding-time analysis.* The second processing phase performs whatever evaluations have been identified during the binding-time analysis, producing the specialized program. Although online partial evaluators have the advantage of being more accurate than offline partial evaluators (because the partial evalutor is processing the actual values of the static inputs, not just the fact that an input is static), the latter are more common because they have the advantage that the binding-time analysis can be performed just once, and then reused for each set of actual inputs.

The basic operations used in the processing stage are constant propagation and constant unfolding. The treatment of function calls is particularly difficult. There are two pitfalls: infinite unfolding and infinite specialization.

A specialized function is created by interpreting the static inputs of that function at partial evaluation time. With specialization, the danger is that we can enter an infinite loop in the partial evaluator as it attempts to produce an infinite number of specialized functions. This occurs when the propagation of static arguments is controlled by dynamic arguments. For example, consider the following code:

```
(define (looper flag n)
  (if (eq? flag 0)
      'done
      (looper flag (+ n 1)))
```

Suppose that `n` is static, but `flag` is dynamic. Depending on its specialization strategy, the partial evaluator might begin producing an infinite number of specialized functions, each with a different value for `n`.

In the case of unfolding, too conservative a strategy leads to trivial optimizations, and residual programs that are poorly specialized with respect to the original. On the other hand, too liberal a strategy can lead to loops in which the same function is unfolded infinitely. One possible approach towards unfolding is to unfold only when all the arguments are static. However, in practice, this turns out to be too conservative a strategy, which means that it is necessary to unfold some functions when all values aren't known. There are several ways of choosing which function calls to unfold. We can statically annotate the functions to be unfolded in the preprocessing stage, or have the user decide interactively, or have some dynamic strategy. The approach taken by some partial evaluators is to only unfold when there is no risk of infinite expansion. This requires a termination analysis, in which we check that any recursive function has a static variable that is decreasing with respect to some partial ordering as function calls are made. For more information on the treatment of function calls, see references [23, 37, 4] and [10].

In the course of executing a program, we may evaluate a particular function many times, with different binding times for arguments . When we perform a binding time analysis, we have the option of associating either one binding time description with each function (a *monovariant* binding time analysis), or else multiple descriptions (a *polyvariant* binding time analysis) [11]. The advantage of a polyvariant binding-time analysis is that it is more accurate, but it does have the disadvantage that with recursive data structures or recursive functions, it may be difficult to ensure the finiteness of the binding-time description — the analysis may not terminate. This problem has been studied, and solutions are discussed in references [9, 11, 34] and [3]. Similarly, a monovariant partial evaluator produces one specialization per function, while a polyvariant partial evaluator can produce many specializations for each function.

2.2 Automatic Compiler Generation

Much of the activity in partial evaluation has centered on the application of partial evaluation to automatic compiler generation, and many languages have been studied, such as a subset of Algol [12], Prolog [14], lazy functional languages [24], and object-oriented languages [26]. This is because the technique of partial evaluation enables us to compile programs in a language when given only an interpreter for that language. This is clearly a promising approach; whereas compilers have advantages over interpreters in terms of speed of execution, they are much harder to write. There are several reasons for this:

1. When writing a compiler, we must keep in mind two binding times: compile-time and run-time. When writing an interpreter (a "normal" interpreter; not one for a partial evaluator!), we have only one binding time to deal with.

2. When writing a compiler, we have an implementation language L that translates the source program written in S to a target program written in T. When writing an interpreter, we have one less language to consider; we execute the source program written in S with an interpreter written in implementation language L.

This application of partial evaluation also ties in with current research in deriving programs from specifications. There is much interest in the research community in being able to automatically generate programs from executable specifications written in a denotational semantics. Using a partial evaluator, we may be able to go from specification to compiler without explicitly writing a compiler; the "ideal" automatic compiler generator would take a "naive" interpreter (that is, an interpreter that implements the specification, and was not written expressly for a partial evaluator), and use the partial evaluator to generate a compiler for a concrete target machine.

2.3 Related Work

Work in semantics-based automatic compiler generation is only about fifteen years old. The publication of Stoy's book on denotational semantics [38] in 1977 seems to have been instrumental in inspiring researchers to look at denotational semantics (or a variant of it) as not only a tool for the formal description of a language, but also an approach to its implementation. The idea is that the semantic equations of a language can be used to produce a compiler.

We will summarize three general approaches to semantics-based automatic compiler generation – denotational semantics, combinators, and action semantics – and discuss briefly how each of them can be used to generate compilers. In the denotational semantics approach, the semantic equations are written in lambda-calculus, and then the idea is to represent these equations as a functional program. The problem with most existing work using this approach is that the systems perform a *translation* from syntax to semantics [36, 12]. A compiler should *execute* all the compile-time actions, leaving only run-time actions for the executable. But without a way to differentiate between the static and dynamic semantics in a language, a denotational-semantics-directed implementation cannot do this. Since a partial evaluator processes static semantics, it can be used to automatically generate a compiler in which all the static actions are performed at partial evaluation time. An early paper by Jones and Schmidt [21] acknowledged this, concluding that their compiled code contained many constructs that could be evaluated at compile-time. It is possible to see the germination of partial evaluation in the solutions they propose for this problem: they discuss performing what amounts to a binding-time analysis on the source program.

In response to this obstacle, Wand developed a *combinator-based* approach to automatic compiler generation in a series of papers [41, 42, 43]. His method was to start with the semantic definition of the language in lambda-calculus, and define a set of combinators for the source language, replacing the lambda-calculus terms by the appropriate combinators. A source program, then, is translated to a tree of combinators. Using some associative and distributive laws, the tree is "rotated" to a linear form. The result is a series of combinators derived for the machine that represent the machine's instructions. The advantage claimed for this approach is that since the machine and the compiler are derived together, this makes correctness proofs for the compiler easier than the situation in which the machine and the language are independent. Unfortunately, the corresponding disadvantage of this approach is that it is more typical to be given a language to compile and a machine independently. This approach has also been criticized on the grounds that it requires "excessive cleverness" to define the combinator set [45]. That being said, we have been strongly influenced by Wand's work in deriving compilers from denotational semantics [42, 41, 43]. In particular, his recent development of a "higher-order abstract assembly language" as a machine language that is derived from CPS terms generated by the denotational semantics [46] is closely related to our approach. His technique also allows him to provide correctness proofs for the compilers. Similar ideas were used to derive compilers from interpreters [16]; the structure of the residual programs that we generate owe much to his approach. Our work differs from his in that we aren't as interested in deriving target architectures as we are in compiling to a given architecture, or abstract machine.

A third approach is another variant of denotational semantics called *action semantics*, developed by Mosses [32]. Action semantics is supposed to represent an operational version of denotational semantics, in which phrases from the source language are mapped to semantic entities called *actions*, which perform computations by "processing information". The goal of action semantics is to provide constructs that are superior to the higher-order functions of denotational semantics from a pragmatic standpoint, while retaining the rigor of the denotational approach. Bondorf and Palsberg have generated a compiler by partially evaluating an action interpreter [5]. Several changes had to be made to the naive interpreter by hand in order to get reasonably good code; specifically, η redexes were added to the interpreter in order to improve detection of early binding times. The resulting code was found to run an order of magnitude faster than target code produced by classical automatic compiler generators.

There are other approaches to compilation that are related to our work that don't fall neatly in one of the above categories. Using concepts from denotational semantics, Kelsey and Hudak [25] describe a method of realistic compilation in which the notion of compilation as a source-to-source transformation process is stressed. They are able to produce low-level code that can be mapped onto a concrete target machine, with good execution speeds. However, they don't address the issue of automating this process, given a language specification.

Clinger [6] developed a Scheme compiler and proved it correct with respect to the semantics of Scheme. The compiler algorithm generates code for an abstract "Scheme machine", whose architecture is influenced by the semantics of Scheme. Our separation of compilation into multiple passes was influenced by his observation that during compilation, it is desirable to have varying levels of representation at different abstraction levels, and the need to maximize the flexibility of representation at each level.

Although there have been many successes in compiling programs using partial evaluation, the compiled code typically executes much slower than a hand-compiled version. This is because the residual code generally is still a functional program (in the implementation language). To execute this compiled code requires the use of either an interpreter, or an existing compiler for the implementation language. In contrast, although our residual programs also are functional after the first pass, we will show that it is possible to use additional partial evaluation passes to generate native machine code.

Chapter 3

Using Partial Evaluation to Compile Scheme Programs

3.1 Introduction

Although it is well-known that the program optimization technique called partial evaluation can be used to do compilation, it is less well understood how the interpreters should be structured to derive compilers effectively. Partial evaluators are often described as *static semantics processors* [36], but there is no guarantee that the partial evaluator will be able to find and execute all the constructs in the source program that correspond to the static semantics (i.e, the compile-time operations). This is especially true if the interpreter is an encoding of a language specification, which will be written for descriptive purposes, not as an implementation guide. For example, a crucial feature of Scheme is that it has no explicit looping constructs. Hence, any implementation has to be properly tail-recursive, yet this feature is not explicit in the specification of Scheme [7].

In this chapter, we describe a promising technique in the direction of realistic compilation of languages by partial evaluation. The idea is to make multiple passes with the partial evaluator over source and residual programs. Our guiding principle here is a tried-and-true one in system design: separate policy from mechanism. We use the first pass through the partial evaluator as our "policy" pass; it is here that we decide *what* language operations are to be executed at compile-time, and which operations are to be done at run-time. The second, and any subsequent passes through the partial evaluator, are the "mechanism" passes, in which we determine the appropriate representations for the high-level operations in the residual program in terms of the low-level machine model. Each subsequent pass can further uncover operational details in terms of concrete target machines, thereby allowing implementations on varied architectures.

A very brief example will give the flavor of our approach. Given the following fragment of source code, and assuming a continuation `k`,

```
(set! x (+ y 4))
```

we specialize the R4RS interpreter with respect to this source code, which produces the following residual code. As expected, the location of variables has been computed (`loc1` is the location of `y` and `loc2` the location of `x`). Furthermore, the implementation-independent details of the computations represented by the source code are made explicit. Lower level operations such as store operations are left unspecified. They are dealt with by later partial evaluation passes.

```
(lookup-store loc1
  (lambda (exprval1)
    (add
      (Concat exprval1 (Concat (number 4) (Empty)))
      (lambda (exprval2 store)
        (update-store loc2 exprval2
          (lambda (store3)
            (k (Unspecified) store3))))
      store2))
  store1)
```

In this code, each lambda is a continuation. That is, the result of doing `lookup-store` on `loc1` in `store1` becomes `exprval1` in the continuation that follows. Similarly, the function `add` takes `exprval1` and the number 4, adds them, and passes the result as `exprval2` in the following continuation (along with `store2`). The innermost continuation gets passed the result of the `update-store` function.

In the second partial evaluation pass, the residual code is fed to the partial evaluator, along with the encoding of an abstract stack machine, and the result is below. `exprval`, which was bound to `loc1` during the `lookup-store` operation during the first pass, is here treated as the memory location of the original variable `y`.

```
(move-top-of-stack exprval1
 (lambda (state)
  (push-contents exprval1
   (lambda (state)
    (push (number 4)
     (lambda (state)
      (prim-add
       (lambda (state)
        (move-top-of-stack loc2))))))))
```

```
(move-top-of-stack exprval1)
(push-contents loc1)
(push (number 4))
(prim-add)
(move-top-of-stack loc2)
```

On the left is the actual residual program; on the right, we have removed the store, since the code is single-threaded. This representation displays the imperative nature of the second residual program. The second partial evaluation yields low-level compiled code that exposes many operational details. In fact, this code is very close to what a byte-code compiler would generate.

The residual code that we obtained after two partial evaluation passes can be properly described as an intermediate representation. This brings up the question of how to get from this representation to native code for a target machine, i.e., back-end compilation. We have identified two alternatives to explore. The first alternative is to make additional parital evaluation passes, with the final pass being to specialize a residual$_n$ program (i.e., the residual program generated after the *nth* partial evaluation pass) with respect to a concrete target machine encoded as an interpreter. This alternative has the benefit of being conceptually consistent with our multi-pass approach to compilation.

However, the second alternative is even more attractive: use off-the-shelf technology to achieve back-end compilation. Just as tools like *lex* and *yacc* have largely solved the problem of doing front-end compilation in practice, there exist a growing number of code generators and code generator generators that can do the same thing for back-end compilation.

3.2 Tools

In this section, we briefly discuss Schism, the partial evaluator used in this project, and R4RS, the specification of the programming language Scheme, the language for which we wish to write a compiler.

3.2.1 Schism

The partial evaluator we use in this work is Schism, an off-line, higher-order, polyvariant partial evaluator for pure, applicative languages [10]. Schism's ability to handle higher-order functions allows us to write interpreters in continuation passing style (CPS) [16], which allows for better program specialization than interpreters in direct style. Schism operates on source programs written in Scheme, enriched with a set of type operations, and a construct called a `filter`. The type operations allow us to define types with `defineType`, and with the operations `letType` and `caseType`, perform `let` and `case` on typed data. A filter in Schism is simply an annotation made to each function definition which tells the system whether to unfold or residualize a call to this function, and whether any of the static arguments should be propagated.

3.2.2 R4RS

Scheme is a statically-scoped, tail-recursive dialect of Lisp. Scheme is based on the untyped lambda calculus; it provides few basic forms, but allows for many ways of combining them. This flexibility allows Scheme to support varied programming paradigms, such as functional or object-oriented. Figure 3.1 describes the abstract syntax of Scheme. In this dissertation, as well as in R4RS, a name followed by the symbol * refers to a sequence. Hence, the definition says that an abstraction takes a sequence I^* (of arguments), a sequence of commands (which it evaluates), and then evaluates an expression (which is the value of the abstraction). Commands and expressions are the same thing, but we identify some expressions as commands if they are being evaluated only for their side effects.

$$\begin{array}{llll}
\mathrm{K} \in \mathrm{Con} & \text{constants} & \mathrm{E} \longrightarrow & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}_0\ \mathrm{E}^*) \\
\mathrm{I} \in \mathrm{Ide} & \text{identifiers} & & (\texttt{lambda}\ (\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0) \\
\mathrm{E} \in \mathrm{Exp} & \text{expressions} & & (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2) \mid (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1) \\
\Gamma \in \mathrm{Com} = \mathrm{Exp} & \text{commands} & & (\texttt{set!}\ \mathrm{I}\ \mathrm{E})
\end{array}$$

Figure 3.1: Abstract Syntax of the Scheme Language

The definition of the Scheme language is given in the Revised^4 Report on Scheme (R4RS). This description includes a denotational semantics for the language. Given Scheme's close relationship to the lambda calculus, the semantics of Scheme can be defined denotationally quite easily, and encoded as a Scheme program just as easily. R4RS not only contains the definitions of the basic forms that comprise Scheme, but also definitions of some of the common built-in procedures such as `car`, `cons`, `eqv?`, etc. One of the reasons that Scheme is a very appropriate language for our experimentation is that R4RS is only 50 pages long; the actual denotational semantic definition is only a handful of pages.

To keep the definition of Scheme from being metacircular, certain language primitives are introduced in the semantics. Lists, for example, are represented explicitly as sequences in R4RS. In our implementation, we use the Schism datatype constructors to define an explicit list type, which we call `Sequence`. In this definition, `Empty` and `Concat` are constructors, and `object` and `more-sequence` are just placeholders.

```
(defineType Sequence
  (Empty)
  (Concat object more-sequence))
```

Other primitive operations that appear in the R4RS semantics are given in Figure 3.2, along with the name of the operation in our Scheme interpreter that corresponds to each primitive.

sequence formation	$\langle\epsilon\rangle$	`form`* ϵ
length of sequence s	$\#s$	`length`* s
k th member of the sequence s (1-based)	$s \downarrow k$	`project`* $s\,k$
concatenation of sequences s and t	$s \,@\, t$	`concat`* $s\,t$
drop the first k members of sequence s	$s \dagger k$	`drop`* $s\,k$
McCarthy conditional	$t \rightarrow a, b$	`if` t `then` a `else` b
injection of x into domain `D`	x in `D`	*various type constructors*
projection of x to domain `D`	$x \mid$ `D`	`caseType`

Figure 3.2: Translation of Language Primitives to Interpreter Primitives

3.3 First Pass

Using R4RS as a specifications, we write a Scheme interpreter. Specializing this interpreter with respect to a Scheme source program gives us a compiled program. As we will see, writing a "naive" interpreter results in a poor-quality residual program (in terms of the sorts of operations we expect a compiler to carry out on a source program). In this section, we examine each of the Scheme basic forms, and discuss the transformations applied to the R4RS interpreter to obtain good quality residual code. We also discuss the language of residual programs, tail recursion, and present an example of compiled code generated by specializing the interpreter with respect to a Scheme source program.

3.3.1 Analysis and Transformation of the R4RS encoding

To compile a program, we want to remove all of the constructs that correspond to static data, and emit code for the constructs that rely on dynamic data. For example, in most imperative languages (and in the case of Scheme), the environment is a mapping from variable names to locations, and a store is a mapping from locations to values. When these languages are compiled, variable names can be replaced by memory locations: the environment can be considered static. The actual values held in these memory locations will not generally be known until run-time, and hence store operations (updates and lookups) can be considered dynamic. By using Schism's `filter` construct, we can residualize these operations; that is, mark them as dynamic. Conversely, references to the environment should not appear in the compiled code, and we expect to "unfold" these operations, i.e., to execute them at partial evaluation time. The process of assigning a static or dynamic tag to variables and functions is called a *binding-time analysis.*

In this chapter, we analyze and transform the R4RS interpreter based on its binding-time properties discovered by the binding-time analysis of Schism. These properties allow us both to identity any changes that must be made to the R4RS interpreter in order to get good-quality compiled code, and to characterize the structure of the emitted code for later partial evaluation passes.

Since R4RS was not written with the division between static and dynamic semantics in mind, it is not surprising that some binding-time improvements need to be made. However, we accomplish virtually all of them in a systematic manner, by using the `filter` construct to residualize function calls that operate on run-time values.

In the following transformation example, functions to be unfolded will be *underlined*, and functions that we residualize with a filter (because they are operations that we wish to appear in the compiled code) will be marked in bold-face type. In our analysis, the

$\Rightarrow_{pe}$ symbol denotes equivalence by unfolding by the partial evaluator, whereas the symbol $\Rightarrow_u$ denotes an equivalence that we have established. Figure 3.3 mentions the important domains and semantic functions that will appear in the analysis. Note that expression continuations take sequences of values, not single values, even though expressions in Scheme only return single values. This was done, the authors explain "to simplify the formal treatment of procedure calls and to make it easy to add multiple return values". We will see that although defining the semantics this way make make it easier to draft R4RS, it will make things more difficult to use R4RS as an implementation guide.

$\epsilon \in \mathtt{E}$ expressible values	$\mathcal{K} : \text{Constant} \rightarrow \mathtt{E}$
$\sigma \in \mathtt{S}$ stores	$\mathcal{E} : \text{Exp} \rightarrow \mathtt{U} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\rho \in \mathtt{U}$ environments	$\mathcal{E}^* : \text{Exp}^* \rightarrow \mathtt{U} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\kappa \in \mathtt{K}$ expression continuations	$\mathcal{C} : \text{Command}^* \rightarrow \mathtt{U} \rightarrow \mathtt{C} \rightarrow \mathtt{C}$
$\theta \in \mathtt{C}$ command continuations	$\mathtt{K} = \mathtt{E}^* \rightarrow \mathtt{C}$
$\mathtt{A}$ answers	$\mathtt{C} = \mathtt{S} \rightarrow \mathtt{A}$

Figure 3.3: Domain Names and Semantic Functions

Evaluation of Constants

Even though Scheme is a small language, some details of its description have been left out of R4RS. For example, the definition of constants, $\mathcal{K}$, is omitted because it would "complicate the semantics without being very interesting". Our approach is to assume that all constants are tagged as being so during syntactic analysis, and so evaluation of a constant consists of the following simplification: $\mathcal{K}[\![\mathrm{K}]\!] \Rightarrow_{pe} \mathrm{E}$, followed by the application of an expression continuation. Figure 3.4 describes this. In this, and following figures, the first appearance of a semantic equation for a basic form, plus any "support" functions, comes directly from R4RS. The second appearance of the semantic equation describes the

$$\mathcal{K}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,send\ (\mathcal{K}[\![\mathrm{K}]\!])\ \kappa$$

$$send : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$$
$$send = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$$

$$\begin{array}{lll} \mathcal{K}[\![\mathrm{K}]\!] & = & \lambda\rho\kappa\,.\,\underline{send}\ (\mathcal{K}[\![\mathrm{K}]\!])\ \kappa \\ & \Rightarrow_{pe} & \lambda\rho\kappa\,.\,\kappa\langle\mathrm{K}\rangle \end{array}$$

Figure 3.4: Transformation of Constants

transformations applied to that equation. Note that the process of applying an expression continuation is made explicit by the *send* operation, which takes an expressible-value as an argument, but applies the continuation to an expressible-value sequence.

Identifier Evaluation

The analysis of evaluation of identifiers is shown in Figure 3.5, and illustrates one of the drawbacks of trying to use R4RS as an implementation guide. We mentioned earlier that expression continuations take a sequence of values, instead of single values, even though Scheme returns only single values. Therefore, after any expression continuation is applied, it is passed to the function *single*, which makes an unnecessary check whether a single value is returned. Looking at the semantics of identifier evaluation, we see that $lookup\rho\mathtt{I}$ returns the location corresponding to `I`. The function *hold* does a store lookup on this location, and passes the value to *single*. If we do have a single value, *single* applies a new continuation, which performs a check to see whether the returned value is *undefined*.

As written, static and dynamic semantics are quite intertwined. The environment lookup is static, but the value of ϵ will not be known at compile-time. Hence, the conditional in *single* will appear in the residual program. This is undesirable, for two reasons. Firstly, the $undefined$ test will involve `caseType`, which means that the residual program

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,&\mathit{hold}\ (\mathit{lookup}\ \rho\ \mathrm{I})\\ &(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \rightarrow\\ &\qquad \mathit{wrong}\ \text{“undefined variable”},\\ &\qquad \mathit{send}\ \epsilon\ \kappa))\end{aligned}$$

$\mathit{wrong} : \mathtt{X} \rightarrow \mathtt{C}$ implementation-independent
$\mathit{single} : (\mathtt{E} \rightarrow \mathtt{C}) \rightarrow \mathtt{K}$
$\mathit{single} = \lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \rightarrow \psi(\epsilon^* \downarrow 1), \mathit{wrong}$ “wrong number of return values”

$\mathit{hold} : \mathtt{L} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{hold} = \lambda\alpha\kappa\sigma\,.\,\mathit{send}(\sigma\alpha \downarrow 1)\kappa\sigma$
$\mathit{hold} \Rightarrow_u \lambda\alpha\kappa\sigma\,.\,\textbf{lookup-store}\ \alpha\ (\lambda\epsilon\,.\,\mathit{send}\ \epsilon\kappa\sigma)$

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] \quad &= \quad \lambda\rho\kappa\,.\,\underline{\mathit{hold}}\ (\underline{\mathit{lookup}}\ \rho\ \mathrm{I})\ (\underline{\mathit{single}}(\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ \kappa))\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\textbf{lookup-store}(\underline{\mathit{lookup}}\,\rho\,\mathrm{I})\ (\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ (\underline{\mathit{single}}\ (\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ \kappa)))\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\textbf{lookup-store}\ \mathit{loc}_\rho\ (\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ (\underline{\mathit{single}}\ (\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ \kappa)))\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\textbf{lookup-store}\ \mathit{loc}_\rho\ (\lambda\epsilon\,.\,\underline{\mathit{send}}\ \epsilon\ (\lambda\epsilon^*\,.\,\#\epsilon^* = 1 \rightarrow\ (\underline{\mathit{send}}\ \epsilon\ \kappa)(\epsilon^* \downarrow 1)\ ,\\ &\qquad\qquad\qquad \underline{\mathit{wrong}}\text{“wrong number of return values”}\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\textbf{lookup-store}\ \mathit{loc}_\rho\ (\lambda\epsilon\,.\,\kappa\langle\epsilon\rangle)\end{aligned}$$

Figure 3.5: Transformation of Identifier Evaluation

will consist of code peculiar to our implementation of the Scheme interpreter (i.e., R4RS says nothing about `caseType`). Secondly, and more importantly, this test is something that should be abstracted away into the implementation of a store lookup.

The function *hold* is not written in CPS. The purpose of *hold* is to map a variable name to a expressible value, and this requires a store lookup (which is dynamic). By rewriting *hold* in CPS, and making the store lookup come after the environment lookup, we will be able to evaluate the environment lookup at partial evaluation time. The function *single* then, is unnecessary, since we can look at the definition of an environment lookup, and see that it returns a single value. What is left is just the `lookup-store` primitive. The result of these transformations is that, if we have an identifier evaluation in our source program, the following code fragment would be emitted during partial evaluation (modulo the variable names).

```
(lookup-store loc
  (lambda (exprval)
    (k exprval store))))
```

Evaluation of Applications

R4RS explicitly makes some aspects of Scheme implementation independent. For example, Scheme does not specify the order of evaluation of arguments in a function call. To simulate this in the specification, two functions called *permute* and *unpermute*, which are inverses of each other, are used to scramble and unscramble the arguments prior to and after the evaluation of the arguments.

$permute : \mathrm{Exp}^* \to \mathrm{Exp}^*$

$unpermute : \mathrm{E}^* \to \mathrm{E}^*$

Even this, the authors admit, is not the intended semantics, since it denotes an unspecified, yet constant, order of evaluation. Residualizing these two functions doesn't pass any useful information to later partial evaluation passes, since there is no real function to implement. Hence, we can think of these functions as identity functions. In practice, we "transform" them by simply removing them from the interpreter; they serve no purpose. Note, however, that we are not insisting on a left-to-right order of evaluation, but assuming that it will be taken care of during compiler optimizations.

$$
\begin{aligned}
\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \lambda\rho\kappa\,.\,\mathcal{E}^*(&\mathit{permute}\ (\langle \mathrm{E}_0\rangle\ @\ \mathrm{E}^*))\\
&\rho\\
&(\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\ (\epsilon^*\downarrow 1)\ (\epsilon^*\dagger 1)\ \kappa)\\
&\qquad(\mathit{unpermute}\ \epsilon^*)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
&\mathit{applicate} = \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon \in \mathtt{F} \to (\,\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\,\kappa, \mathit{wrong}\ \text{“bad procedure”}\\
&\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^* \quad (\mathit{identity})\\
&\mathit{unpermute} : \mathrm{E}^* \to \mathrm{E}^* \quad (\mathit{identity})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] \quad &= \quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\underline{\mathit{permute}}\ (\langle \mathrm{E}_0\rangle\ @\ \mathrm{E}^*))\\
&\qquad\qquad \rho(\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathbf{applicate}\ (\epsilon^*\downarrow 1)\ (\epsilon^*\dagger 1)\ \kappa)\\
&\qquad\qquad\qquad (\underline{\mathit{unpermute}}\ \epsilon^*)))\\
&\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\langle \mathrm{E}_0\rangle\ @\ \mathrm{E}^*)\ \rho(\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathbf{applicate}\ (\epsilon^*\downarrow 1)\ (\epsilon^*\dagger 1)\ \kappa)\ \epsilon^*))\\
&\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\langle \mathrm{E}_0\rangle\ @\ \mathrm{E}^*)\ \rho(\lambda\epsilon^*\,.\,\mathbf{applicate}\ (\epsilon^*\downarrow 1)\ (\epsilon^*\dagger 1)\ \kappa)
\end{aligned}
$$

Figure 3.6: Transformation of Function Application

Figure 3.6 contains the details of the transformation performed on a function application by Schism. We choose to residualize the procedure `applicate`, since its function is to perform a tag check on an expressible-value. This cannot be done until run-time, and if `applicate` is unfolded, we will have code peculiar to our implementation of the interpreter (i.e., a `caseType` operation during the tag check). Runtime tag checking is left abstract until we are ready to deal with specific implementations (via a later pass by a partial evaluator, or a code generator). Similar situations will occur with functions in other basic forms, and we justify residualizing them in the same way – the first partial evaluation compilation pass ought to produce code at a sufficiently high abstraction level so as to defer appropriate details to later phases.

Evaluation of Abstractions

Figure 3.7 contains the semantic definition of a Scheme lambda abstraction. In our experiments compiling Scheme programs, we found that abstractions were the most difficult of the basic forms to compile well without transformations being applied to the semantics. In this definition, notice that when the number of formals and actuals agree, a function called *tievals* is called. The job of *tievals* is to extend the environment, and update the store, with the formal-actual pairs. However, as R4RS is written, *tievals* cannot do its job at compile-time. Although we don't know the values of the actuals at partial evaluation time, we do know how many of them there should be. Hence, by counting the number of formals, we unroll *tievals* that many times, as shown in Figure 3.8. This accomplishes the following two things:

1. we have extended the environment by generating the proper number of new locations
2. we have emitted run-time code that checks that for each new location, we are not out of memory (the function *new*), and also generates an *update-store* instruction to update these new locations with the expressible-values expected at run-time.

The unrolled code should only be executed at run-time if the number of actuals is equal to the number of formals. Hence, we define a new function called *arity-check*. This function checks, at run time, whether to execute the code, or whether to return an error message.

Figure 3.8 describes the unrolling of *tievals*. We added one more argument to *tievals*; the arity of the formals, and rewrote *tievals* to make this the induction variable. This preserves the meaning of the semantics, since the branch containing *tievals* is only taken when the old and new induction variables agree (as shown in the definition of the *tievals* function. Since the new induction variable is static, Schism can unfold *tievals*, producing a series of *new, update-store* pairs, one for each location, followed by the function ψ applied to the location sequence. The ϵ^* extractors, $((\epsilon^* \dagger 1)^i \downarrow 1)$, are also residualized, so that the definition bodies won't appear in the compiled code. In the interpreter, these residualized `drop`* and `project`* functions are called `res-drop`* and `res-project`*, respectively.

$$
\begin{array}{l}
\mathcal{E}[\![(\mathrm{lambda}\ (\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0)]\!] = \\
\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathrm{L} \to \\
\qquad\qquad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathrm{L}, \\
\qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\,\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho'\,\kappa')) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{extends}\ \rho\ \mathrm{I}^*\ \alpha^*)) \\
\qquad\qquad\qquad\quad \epsilon^*, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}\rangle \\
\qquad\qquad\quad \text{in E}) \\
\qquad\qquad\quad \kappa \\
\qquad\qquad\quad (\mathit{update}\ (\mathit{new}\,\sigma \mid \mathrm{L}\ \mathit{unspecified}\,\sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\ \sigma
\end{array}
$$

$$
\begin{array}{l}
\mathit{arity\text{-}check} : \mathtt{E}^* \to \mathtt{N} \to \mathtt{Exp} \to \mathtt{C} \\
\mathit{arity\text{-}check} = \lambda\epsilon^* n\mathcal{E}\,.\,\#\epsilon^* = n \to \mathcal{E}\ ,\ \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}[\![(\mathrm{lambda}\ (\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0)]\!] & = & \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathbf{new}\,\sigma \in \mathrm{L} \to \\
 & & \qquad \underline{\mathit{send}}\,(\langle \mathbf{new}\,\sigma \mid \mathrm{L}, \\
 & & \qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
 & & \qquad\qquad \underline{\mathit{tievals}}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\,\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho'\,\kappa')) \\
 & & \qquad\qquad\qquad (\underline{\mathit{extends}}\ \rho\ \mathrm{I}^*\ \alpha^*))\ \epsilon^*, \\
 & & \qquad\qquad \underline{\mathit{wrong}}\ \text{“wrong number of args”}\rangle\ \text{in E}) \\
 & & \qquad\quad \kappa \\
 & & \qquad\quad (\mathbf{update\text{-}store}\ (\mathbf{new}\,\sigma \mid \mathrm{L}\ \mathit{unspecified}\,\sigma), \\
 & & \qquad \underline{\mathit{wrong}}\ \text{“out of memory”}\ \sigma \\
 & \Rightarrow_u & \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathbf{new}\,\sigma \\
 & & \qquad (\lambda\alpha\,.\,\mathbf{update\text{-}store}\ \alpha\ \mathit{unspecified} \\
 & & \qquad\quad (\underline{\mathit{send}}\,((\langle\alpha\ ,\ \lambda\epsilon^*\kappa'\,.\,\mathbf{arity\text{-}check}\ \epsilon^*\ (\underline{\#}\mathrm{I}^*) \\
 & & \qquad\qquad (\underline{\mathit{tievals}}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\,\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho'\,\kappa')) \\
 & & \qquad\qquad\quad (\underline{\mathit{extends}}\ \rho\ \mathrm{I}^*\ \alpha^*))\ \epsilon^*(\underline{\#}\mathrm{I}^*)\ \rangle\ \text{in E})))\ \kappa \\
 & \Rightarrow_{pe} & \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathbf{new}\,\sigma \\
 & & (\lambda\alpha\,.\,\mathbf{update\text{-}store}\ \alpha\ \mathit{unspecified} \\
 & & \quad (\kappa(\ \langle\alpha\ ,\ \lambda\epsilon^*\kappa'\,.\,\mathbf{arity\text{-}check}\ \epsilon^*\ n \\
 & & \qquad \mathbf{new}\ \sigma\ (\lambda\alpha_1\,.\,\mathbf{update\text{-}store}\ \alpha_1\ (\epsilon^* \downarrow 1) \\
 & & \qquad\quad \mathbf{new}\ \sigma\ (\lambda\alpha_2\,.\,\mathbf{update\text{-}store}\ \alpha_2\ ((\epsilon^* \dagger 1)^1 \downarrow 1) \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad\quad (\mathbf{new}\ \sigma\ (\lambda\alpha_n\,.\,\mathbf{update\text{-}store}\ \alpha_n\ ((\epsilon^* \dagger 1)^n \downarrow 1) \\
 & & \qquad\qquad\qquad (\mathcal{C}[\![\Gamma^*]\!]\,\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho'\,\kappa'))\ \ldots)\ \rangle\ \text{in E}))) \\
 & & \\
 & & \qquad \text{where } \rho' = (\underline{\mathit{extends}}\ \rho\ \mathrm{I}^*\ \alpha^*) \text{ and } n = (\underline{\#}\mathrm{I}^*)
\end{array}
$$

Figure 3.7: Transformation of Abstractions

$new\ :\ \mathtt{S} \rightarrow (\mathtt{L} + \{\text{error}\})$ implementation-independent

$$
\begin{aligned}
&tievals : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{C}\\
&tievals = \lambda\psi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \rightarrow \psi\langle\,\rangle\sigma,\\
&\qquad\qquad new\,\sigma \in L \rightarrow tievals\ (\lambda\alpha^* \,.\, \psi(\langle new\,\sigma \mid \mathtt{L}\rangle \mathbin{@} \alpha^*))\\
&\qquad\qquad\qquad (\epsilon^* \dagger 1)\\
&\qquad\qquad\qquad (update\ (new\ \sigma \mid \mathtt{L})\ (\epsilon^* \downarrow 1)\ \sigma),\\
&\qquad\qquad wrong\text{“out of memory”}\sigma
\end{aligned}
$$

$$
\begin{aligned}
&tievals : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{N} \rightarrow \mathtt{C}\\
&tievals \quad \Rightarrow_u \quad \lambda\psi\epsilon^* n\sigma \,.\, n = 0 \rightarrow \psi\langle\,\rangle\sigma,\\
&\qquad\qquad \mathbf{new}\ \sigma(\lambda\alpha \,.\, \underline{tievals}\ (\lambda\alpha^* \,.\, \psi(\langle\alpha\rangle \mathbin{@} \alpha^*))\\
&\qquad\qquad\qquad (\epsilon^* \dagger 1)\\
&\qquad\qquad\qquad (\textbf{update-store}\ \alpha\ (\epsilon^* \downarrow 1)\ \sigma)\\
&\qquad\qquad\qquad (n-1))\\
&\qquad \Rightarrow_{pe} \quad \lambda\psi\epsilon^* n\sigma \,.\, \mathbf{new}\ \sigma\ (\lambda\alpha_1 \,.\, \textbf{update-store}\ \alpha_1\ (\epsilon^* \downarrow 1)\\
&\qquad\qquad \mathbf{new}\ \sigma\ (\lambda\alpha_2 \,.\, \textbf{update-store}\ \alpha_2\ ((\epsilon^* \dagger 1)^1 \downarrow 1)\\
&\qquad\qquad \ldots\\
&\qquad\qquad (\mathbf{new}\ \sigma\ (\lambda\alpha_n \,.\, \textbf{update-store}\ \alpha_n\ ((\epsilon^* \dagger 1)^{n-1} \downarrow 1)\\
&\qquad\qquad\qquad (\psi\langle\alpha_1\,,\ \alpha_2\,,\ \ldots\alpha_n\rangle)))) \ldots)
\end{aligned}
$$

Figure 3.8: Transformation of the Tievals Function

One other issue is the *new* function, which is given as implementation-independent (that is, R4RS leaves it undefined). We rewrite *new* as a function that takes a store, and a continuation, and returns a location, and a new store (the old store plus a newly-allocated memory allocation). Since *new* is meant to be a run time operation, we residualize each appearance of it with the appropriate filter, and expect that an implementation pass will use *new* as a signal to generate code that performs a trap if we run out of memory.

Conditional Evaluation

The top of Figure 3.9 describes the semantics of a Scheme conditional. A test expression is evaluated, and the expressible value is passed to *single*. Similar to the situation for identifier evaluation, we have the appearance of the function *single* (which was removed after we applied our transformation, and showed that the rewritten interpreter would never pass anything but a single value to *single*). As noted earlier, this guarantee is easy to make, since only single expressible values are passed. Hence, our transformation is simply to remove *single*.

$$\mathcal{E}[\![(\text{if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = $$
$$\qquad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\ (single\ (\lambda\epsilon\,.\,truish\ \epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\ ,\ \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))$$

$$truish : \mathtt{E} \rightarrow \mathtt{T}$$
$$truish = \lambda\epsilon\,.\,\epsilon = false \rightarrow false\ ,\ true$$

$$\begin{aligned} \mathcal{E}[\![(\text{if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!] &= \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\ (\underline{single}\ (\lambda\epsilon\,.\,\mathbf{truish}\ \epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\ ,\ \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa)) \\ &\Rightarrow_{pe} \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\ (\lambda\epsilon\,.\,\mathbf{truish}\ \epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa\ ,\ \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa) \end{aligned}$$

Figure 3.9: Transformation of Conditionals

The function *truish* is residualized, to represent the notion of branching as the result of the evaluation of a test expression.

Evaluation of Set!

In our implementation of the Scheme interpreter, we have written all of our primitive store operations (which will be residualized) to take a continuation argument. In Figure 3.10, we have the original semantics of the `set!` function. Notice that the primitive store operation *update-store* doesn't take a continuation. Since the only purpose of *assign* is to apply a continuation to the result of *update-store*, we eliminate *assign* by making the continuation an argument to *update-store*. We then unfold *send*. The residual code fragment that corresponds to the code that comes after $\mathcal{E}[\![\text{E}]\!]$ looks like this:

```
(lambda (exprval store)
  (update-store loc exprval
    (lambda (store)
      (k (Unspecified) store)))))
```

Notice that `Unspecified` is the value returned by the `set!` operation. Evaluating any Scheme basic form is supposed to result in some return value, but `set!`, being Scheme's assignment statement, is evaluated just for the it's side effect (of updating the store). Hence, `set!` just returns the "don't care" value `Unspecified`.

$$\begin{aligned}\mathcal{E}[\![(\text{set! I E})]\!] = \lambda\rho\kappa\,.\,\mathcal{E}[\![\text{E}]\!]\,\rho\,(single\,(\lambda\epsilon\,.\,assign\,&(lookup\,\rho\,\text{I})\\ &\epsilon\\ &(send\ unspecified\ \kappa)))\end{aligned}$$

$$\begin{aligned}&assign : \mathtt{L}\rightarrow\mathtt{E}\rightarrow\mathtt{C}\rightarrow\mathtt{C}\\ &assign \quad = \quad \lambda\alpha\epsilon\Theta\sigma\,.\,\Theta(update\text{-}store\ \alpha\ \epsilon\ \sigma)\\ &assign \quad \Rightarrow_u \quad \lambda\alpha\epsilon\Theta\sigma\,.\,(update\text{-}store\ \alpha\ \epsilon\ \Theta\ \sigma)\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![(\text{set! I E})]\!] \quad &= \quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\text{E}]\!]\,\rho\,(\lambda\epsilon\,.\,\textbf{update-store}\ (\underline{lookup}\,\rho\,\text{I})\ \epsilon\ (\underline{send}\ unspecified\ \kappa))\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\text{E}]\!]\,\rho\,(\lambda\epsilon\,.\,\textbf{update-store}\ loc_\rho\ \epsilon\ (\underline{send}\ unspecified\ \kappa))\\ &\Rightarrow_{pe} \quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\text{E}]\!]\,\rho\,(\lambda\epsilon\,.\,\textbf{update-store}\ loc_\rho\ \epsilon\ (\kappa\ unspecified))\end{aligned}$$

Figure 3.10: Transformation of Set!

Built-In Primitives

R4RS has semantic definitions for a few of the procedures that are built-in to any implementation of Scheme For example, consider the definition of the primitive *add*[1] given in Figure 3.11. We don't want the partial evaluator to "go inside" the definition of *add*, so we use a filter to just residualize *add*. This means that all of the primitive function names will appear in the residual code. We are effectively deferring any kind of implementation decision for these primitives, which is the right thing to do; clearly, the R4RS definition of *add* bears no relation to its machine language implementation.

$$
\begin{aligned}
&add : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
&add = twoarg(\lambda\epsilon_1\epsilon_2\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
&\qquad\qquad send\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa\,,\\
&\qquad\qquad wrong\text{"non-numeric argument to +"})\\
&\\
&twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})\\
&twoarg = \lambda\zeta\epsilon^*\kappa\,.\,\#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa\,,\\
&\qquad\qquad\quad wrong \text{ "wrong number of arguments"}
\end{aligned}
$$

Figure 3.11: Example of a Built-In Procedure

3.3.2 Language of Residual Programs

A Scheme source program is a series of top-level definitions, followed by a Scheme expression. The value of the program is the value returned by that expression. To complete our picture of the structure of residual programs, we need to know the semanctics of top-level definitions. In Scheme, top-level definitions are equivalent to `set!` expressions; if the variable is not already bound, the variable is bound to a new location before the assignment is performed. We can statically analyze a source program to locate all the top-level definitions; the variable names corresponding to these definitions are collected in a sequence

[1]It is true that implementations of Scheme do not restrict procedures like *add* to two arguments, but R4RS specifies *add* as a binary function.

at the top of the source program. During compilation, locations are allocated for all of those variables, and then the top-level definitions are all executed as `set!` expressions.

$$
\begin{array}{rcl}
\mathrm{P} & \in & \text{Residual Programs} \\
\mathrm{Exp} & \in & \text{Expressions} \\
\epsilon & \in & \text{Expressible Values} \\
n & \in & \text{Integers} \\
\kappa_{\epsilon\sigma}\,,\ \kappa_{\alpha\sigma}\,,\ \kappa_{\sigma}\,,\ \kappa_{\epsilon}\,,\ \kappa_{a\kappa\sigma}\,,\ \kappa & \in & \text{Continuations} \\
\sigma & \in & \text{Stores} \\
\mathrm{Cont} & \in & \text{Continuation Bodies} \\
\mathrm{PrimOp} & \in & \text{Primitive Operators} \\
\alpha & \in & \text{Locations}
\end{array}
$$

$$
\begin{array}{rcl}
P & ::= & Exp \\
Exp & ::= & \mathbf{if}\ (\mathbf{truish}\ \epsilon)\ Exp_1\ Exp_2 \mid \mathbf{applicate}\ \epsilon\ \epsilon^*\ \kappa_{\epsilon\sigma}\ \sigma \\
 & & \mid \mathbf{new}\ S\ \kappa_{\alpha\sigma} \mid \mathbf{apply\text{-}k}\ \kappa\ \epsilon\ \sigma \\
 & & \mid \mathbf{update\text{-}store}\ \alpha\ \epsilon\ \kappa_{\sigma}\ \sigma \mid \mathbf{lookup\text{-}store}\ \alpha\ \kappa_{\epsilon}\ \sigma \\
 & & \mid PrimOp\ \epsilon^*\ \kappa_{\epsilon\sigma}\ \sigma \\
PrimOp & ::= & add \mid mult \mid car \mid cons\ \ldots \\
\kappa_{args} & ::= & \lambda\, args\, .\mathrm{Cont} \\
Cont & ::= & Exp \mid final\text{-}continuation \\
\epsilon & ::= & unspecified \mid undefined \mid true \mid false \\
 & & \mid Int \mid null \mid \text{procedure}\ \alpha\ \kappa_{\epsilon\kappa\sigma} \mid ((\epsilon^* \dagger 1)^i \downarrow 1) \mid \ldots \\
\kappa_{\epsilon\kappa\sigma} & ::= & arity\text{-}check\ \epsilon^*\ n\ \mathrm{Exp}
\end{array}
$$

Figure 3.12: Syntax of the residual language

By transforming the Scheme basic expressions as we have described, we create residual programs. These residual programs are not just transformations of Scheme programs – they can be regarded as being written in a new language. Figure 3.12 contains the BNF of the residual programs. This BNF is based on the analysis given previously of the binding time properties of the various Scheme constructs. Our experiments with compiling

```
(define (fact n)
  (if (less? n 1)
      1
      (* n (fact (- n 1)))))

(fact 4)
```

Figure 3.13: Source Code for the Factorial Function

Scheme programs confirm that residual programs do conform to the BNF given. A program consists of one expression, which is made up of various instructions that have been residualized during compilation of the basic forms, followed by an appropriate continuation. The subscripts in the continuations describe the formals of the lambda expression in that continuation. The body of a continuation is another expression, or else a final continuation (there is only one of the latter; a final continuation returns the value of the program evaluation).

3.3.3 An Example

As a simple example (to keep the size of the residual code readable), consider the source program in Figure 3.13. This program contains one top-level definition; that of the canonical recursive factorial function, followed by an expression that applies the factorial function to 4. Of course, factorial is an old and worn-out example, but it has the virtue of being both small and well-understood.

When this program is compiled with Schism, a residual program is produced containing the compiled source code, along with the definitions of any functions that were residualized during compilation. Figure 3.14 shows the compiled source code, i.e., the semantics of R4RS specialized to the factorial code in 3.13.

```
(define (evprogram.1 store)
 (new store
  (lambda (loc.3 store_1.2)
   (update-store loc.3 (unspecified)
    (lambda (store_2.4)
     (new store_2.4
      (lambda (loc.8 store_1.7)
       (update-store loc.8 (unspecified)
        (lambda (store_2.9)
         (update-store loc.3
            (procedure
             (lambda (exprval*.18 k1.17 store_3.16)
              (arity-check exprval*.18 1
                 (new store_3.16
                  (lambda (loc.20 store_1.19)
                   (update-store loc.20 (res-project* exprval*.18 1)
                    (lambda (store_2.21)
                     (lookup-store loc.20
                      (lambda (exprval.23)
                       (less-than
                        (concat exprval.23 (concat (number 1) (empty)))
                        (lambda (exprval.27 store_1.26)
                         (if (truish exprval.27)
                             (k1.17 (number 1) store_1.26)
                             (lookup-store loc.20
                              (lambda (exprval.28)
                               (lookup-store loc.3
                                (lambda (exprval.31)
                                 (lookup-store loc.20
                                  (lambda (exprval.34)
                                   (subtracts
                                    (concat exprval.34 (concat (number 1) (empty)))
                                    (lambda (exprval.38 store_1.37)
                                     (applicate
                                      exprval.31
                                      (concat exprval.38 (empty))
                                      (lambda (exprval.40 store_1.39)
                                       (multiplies
                                        (concat exprval.28 (concat exprval.40 (empty)))
                                        k1.17
                                        store_1.39))
                                      store_1.37))
                                    store_1.26))
                                 store_1.26))
                               store_1.26))
                             store_1.26)))
                         store_2.21))
                    store_2.21))
                store_1.19))))))
            (lambda (store_2.10)
             (lookup-store loc.3 (lambda (exprval.11)
              (applicate
               exprval.11
               (concat (number 4) (empty))
               (lambda (exprval_1.15 store_1.14) exprval_1.15)
               store_2.10))
             store_2.10))
          store_2.9))
        store_1.7))))
    store_1.2))))
```

Figure 3.14: Result of First Pass

3.3.4 Tail Recursion

A very important feature of Scheme is that all implementations of it must be properly tail-recursive. This means that even if an iterative computation is written recursively, the implementation must guarantee that it will execute in constant space. This requirement cannot be made explicit in the denotational semantics, as it is an implementation, or operational, issue. Fortunately, the residual code generated by the first pass makes it explicit when we have a recursive call that can be handled iteratively: whenever the continuation to a function call is trivial, we can handle it iteratively, otherwise, we treat it as a general recursive function. A trivial continuation is a variable like `k`; a nontrivial continuation is an explicit lambda abstraction in the residual code. Hence, the structure of the residual code enables us to identify those recursive functions that can be compiled as iterative functions.

3.3.5 Assessment of First Pass

Although traditional compile-time actions (such as replacing identifiers with locations) have been accomplished, the residual code is still too high-level: we have operations on a store, but no details on how these operations are implemented. The residual program is still a functional program; we would like to compile down to a concrete, imperative target architecture, for greater efficiency. Our strategy will be to use additional partial evaluation phases to accomplish this.

3.4 Creating a Second Pass

In this section, we discuss the construction of an abstract machine designed to execute residual programs, and show that specialization of this machine processor with respect to a particular source program generates a lower-level, imperative representation of Scheme source programs.

3.4.1 The Abstract Machine

When we encoded the R4RS specification as an interpreter, we were actually encoding a very high-level abstract machine. The result of partially evaluating this machine with respect to a source program was a residual program consisting of high-level operations. In fact, we tried to make the compilation process as implementation-independent as possible, in keeping with our approach to separate policy and mechanism: the first pass is designed to remove compile-time constructs; subsequent passes can implement the run-time constructs.

For the second pass, we encode an abstract, but lower-level machine as an interpreter. With each additional pass after the first, we make more operational details explicit. We imagine a "tree" of abstract machines, with the leaves being concrete architectures and instruction-sets. Then, the process of compilation is a sequence of partial evaluation steps, from a root (or roots) to a leaf.

Current imperative architectures tend to be register-based, but we choose our abstract machine to be a hybrid stack/register machine. The various memory locations in the residual code can be thought of as registers. However, we also have use for a stack – all arguments passed to function calls are just thrown on the stack. This is done so that a later partial evaluation pass can decide how many, and which registers are available for function arguments; by throwing all the arguments on a stack now, we will be deferring this kind of decision.

Our virtual machine is designed to take residual programs, which are in CPS, and emit new residual programs, which we refer to as residual$_2$ programs. These residual$_2$ programs are also in CPS, but are made up of instructions acting upon a stack machine, as opposed to a high-level store. Figure 3.15 describes the instructions that make up the abstract machine.

Some of the instruction-set choices deserve some explanation. When a primitive like `prim-add` or `prim-less`, or a user-defined function returns with a value, it is passed, along with the updated store, to the next instruction, by a $\kappa_{\epsilon\sigma}$ continuation, as in the following code fragment:

```
(prim-subtract
 (concat exprval.1 (concat (number 1) (empty)))
 (lambda (exprval.2 store.3) ...))
```

In this code, the function prim-subtract operates on the sequence consisting of `exprval.1` and `(number 1)` (i.e., it subtracts the latter from the former). The result is passed to the continuation, which means that it should bind it to `exprval.2`. Since we handle function calls with a stack, we generate a `move-top-of-stack` instruction, which moves it to a temporary location designated `exprval.2`. This handles a problem that all compilers must solve, that of deciding what to do with values returned by a function call. They

```
new-memory location
push arg
push-contents location
allocate-frame
deallocate-frame
procedure-tag? location
eval-applicate location
top-of-stack
move-top-of-stack location
eval-procedure-code code
arity-check integer
branch-on-contents location
prim-less, prim-add, etc.
contents loc
halt
```

Figure 3.15: Abstract Stack Machine Instructions

cannot stay where they are because they may be overwritten before they are used in the computation. Therefore, after any procedure call, we generate a `move-top-of-stack` instruction. In a similar way, we generate a `push-contents` instruction when an argument to a procedure is actually a location.

Other instructions map easily from the language of residual programs to the virtual machine. For example, the final continuation in the residual program becomes a `halt` instruction, and applying a trivial continuation becomes a `deallocate-frame` instruction. The virtual machine is designed so that each instruction operates on the current state, and passes the new state to its continuation.

3.4.2 An Example

The simple design of the virtual machine results in the language of residual$_2$ programs that is illustrated in Figure 3.4.2.

As an example of this, the following is a portion of the residual$_2$ program generated by using the partial evaluator to specialize the abstract stack machine processor with respect to the residual code generated for the program in Figure 3.14.

```
(define (evprogram.1 state)
 (new-memory loc.3 state_1.2
```

$$\begin{array}{rcl}
\text{P} & \in & \text{Residual}^2 \text{ Programs} \\
\text{Exp} & \in & \text{Expressions} \\
\epsilon & \in & \text{Expressible Values} \\
n & \in & \text{Integers} \\
\kappa_\sigma & \in & \text{Continuations} \\
\sigma & \in & \text{States}
\end{array}$$

$$\begin{array}{rcl}
P & ::= & Exp \\
Exp & ::= & Instruction\ \kappa_\sigma \mid Proc\text{-}body \\
Proc\text{-}body & ::= & \lambda\sigma\,.\,Instruction\ \kappa_{proc} \\
Instruction & ::= & \text{Listed in Figure 3.15} \\
\kappa_\sigma & ::= & \lambda\sigma\,.\,Exp \mid \text{halt} \\
\kappa_{proc} & ::= & \lambda\sigma\,.\,Instruction \\
\epsilon & ::= & unspecified \mid undefined \mid true \mid false \\
 & & \mid Int \mid null \mid \text{procedure } \alpha\ \kappa_{\epsilon\kappa\sigma} \mid ((\epsilon^* \dagger 1)^i \downarrow 1) \mid \ldots
\end{array}$$

Figure 3.16: Syntax of residual$_2$ Programs

```
(lambda (state_1.2)
 (push (unspecified) state_1.2
  (lambda (state_1.3)
   (move-top-of-stack loc.3 state_1.3
    (lambda (state_2.4)
     (new-memory loc.8 state_2.4
      (lambda (state_1.5)
       (push (unspecified) state_1.5
        (lambda (state_1.6)
         (move-top-of-stack loc.8 state_1.6
          (lambda (state_2.7)
           (eval-procedure-code
            (lambda (state_1.8)
             (arity-check 1 state_1.8
              (lambda (state_1.9)
                   ⋱
```

3.4.3 Assessment of the Second Pass

As expected, the code is extremely regular in structure, consisting of a state-transforming operation, followed by a lambda expression. This program is again functional, but it is easy to carry out a syntactic transformation to make it look like imperative-style abstract machine code. We note that since the program is trivially single-threaded, the state can be globalized, and therefore removed altogether. This allows us to remove all of the lambda expressions, too, since they take only the state as an argument. We are left with the imperative-style code in Figure 3.17. We don't claim that this code is in any way optimized; the careful reader will note that as the result of allocating a new memory location, which returns a value of **unspecified**, this hapless value gets thrown on the stack. A subsequent partial evaluation pass should get rid of useless instructions like this.

This code can be thought of as an intermediate representation; we have the option of compiling down to a target machine either by making more partial evaluation passes, or by hooking up the current representation to an off-the-shelf code generator.

3.5 Conclusions

3.5.1 Assessments

Our technique is based on the idea that compilation via partial evaluation may be addressed by compiling down a hierarchy of abstract machines, each revealing more operational details, as opposed to trying to make the leap between a high-level specification and low-level code all at once.

At first, we were not able to get good specialization of source programs because of the way certain semantic functions were defined. For example, R4RS lays out the semantics of Scheme in CPS, but not at the granularity we desire; some of the functions (the store primitives) do not take a continuation argument. We changed these so that they do. Also, functions like *send* and *single* were originally defined in order to make the addition of multiple return values easy to specify in the semantics; we were able to remove these function completely. None of these changes had any effect on the semantics of Scheme.

When the above changes had been made, we found that the additional transformations needed to turn the semantic definition into an interpreter that could be specialized well fell into two categories:

1. Residualization of dynamic operations. By annotating these operations with the filter construct, we were able to generate code with the proper level of abstraction.

2. Binding-time improvements. These are improvements such as the one that was made to the `tievals` function, in which we rewrote the interpreter so that we could expose all the static operations at partial evaluation time. The trick is to do this in such a way that we are still faithful to the intended meaning of the Scheme basic forms (if not to the particular denotational definition given in R4RS). Of course, as we have stated previously, since R4RS is a specification, not an implementation guide, we shouldn't be surprised that situations like this occur. We just hope that they are rare.

Fortunately, virtually all of the transformations made to the interpreter were of the first type; if there had been too many of the latter type, then our approach would be open to the criticism of being too ad-hoc to be useful.

The second pass converted the high-level residual program into a new residual program that was both more structured (all of the continuations had the same form, for example), and more low-level. Making the abstract machine a stack machine allowed us to defer much of the concrete implementation details, while still producing code that could be executed imperatively. This was demonstrated by the simple syntactic transformation of removing all of the continuations from the residual$_2$ code, leaving just the instructions. By making just two passes, we have been able to compile a high-level specification of the Scheme language into code that runs on a fairly low-level abstract machine.

The Scheme construct *call-with-current-continuation* (*cwcc*) is an important feature of the language that we have not yet discussed. R4RS does give a semantics for this function along with other Scheme primitives. Since the definition of *cwcc* contains only calls to run-time operations, unfolding it in our current framework would not gain us anything. Therefore, we residualize any calls to *cwcc* with a filter, deferring the implementation to a later partial evaluation pass. The problem of finding an appropriate level of abstraction for *cwcc* requires further investigation.

Like previous work in compilation via partial evaluation, we were able to perform some simple transformations and annotations to the R4RS interpreter in order to get good residual code. The risk is that, without any unifying principle, transformations can be criticized as being ad-hoc, and of no use the next time it comes to compile a language this way.

Generating native code using multiple partial evaluation passes does raise some interesting compilation issues. For each compilation phase, we must write an abstract machine to specialize with respect to its source programs. This means that if we wish to perform compiler optimizations, they must be encoded in an interpretive framework. Work has been done on the problem of expressing variousd compiler optimizations in a functional framework [15] , but is unclear whether this can be translated to an interpretive algorithm. For example, elimination of the common subexpressions F from the expression E can be characterized as follows, but this doesn't specify an appropriate algorithm.

$$E \rightarrow (\lambda x \,.\, E[x/F])\, F$$

Further work is necessary to determine what kinds of optimizations can be encoded in an interpreter. An interesting long-range goal would be to create a taxonomy of abstract machines that can be arranged in a hierarchy. These would be designed so as to promote sharing of abstract machines by languages families; for example, it is not unreasonable for Scheme and ML to share a representation through an abstract machine at some level.

Producing native code would then consist of selecting a compilation path from a root to a leaf (where the latter represents a concrete machine, or a hook to a code generator), allowing for fine control over the compilation process.

```
(new-memory loc.3)                     ; trap, or allocate new location
(push (unspecified))                   ; push on stack
(move-top-of-stack loc.3)              ; loc.3 unspecified
(new-memory loc.8)                     ; trap, or allocate new location
(push (unspecified))                   ; push unspecified on stack
(move-top-of-stack loc.8)              ; loc.8 contains unspecified
(eval-procedure-code                   ; code for factorial starts here
  (arity-check 1)                      ; trap if number of args on stack isn't 1
  (new-memory loc.19)                  ; trap, or allocate new location
  (move-top-of-stack loc.20)           ; move value of n to loc.20
  (push (number 1))                    ; put 1 on stack
  (push-contents loc.20)               ; put n on stack
  (push (args 2))                      ; info for ''wrong number of args'' trap
  (prim-less)                          ; compare values on stack
  (move-top-of-stack exprval.27-loc)   ; move result to exprval.27-loc (a temp)
  (branch-on-contents exprval.27-loc)  ; branch on boolean at exprval.27-loc
                                       ; false branch starts here
   (push (number 1))                   ; put 1 on stack
   (push-contents loc.20)              ; put value of n on stack
   (push (args 2))                     ; trap info for arity check
   (prim-sub)                          ; subtract 1 from n
   (move-top-of-stack exprval.38-loc)  ; move result to temp
   (push-contents exprval.38-loc)      ; put contents of temp on stack
   (push (args 1))                     ; trap info for arity check
   (allocate-frame)                    ; save stack frame
   (procedure-tag? loc.3)              ; trap if loc.3 isn't a procedure
   (eval-applicate loc.3)              ; make recursive call
   (move-top-of-stack exprval.40-loc)  ; move result to temp
   (push-contents exprval.40-loc)      ; put result on stack
   (push-contents loc.20)              ; put n on stack
   (push (args 2))                     ; trap info
   (prim-mult)                         ; multiply n and fact(n - 1)
   (deallocate-frame)                  ; restore stack frame
                                       ; true branch starts here
   (push (number 1))                   ; put 1 on stack
   (deallocate-frame)                  ; restore stack frame
                                       ; end of code for factorial procedure
(move-top-of-stack loc.3)              ; move factorial code to loc.3
(push (number 4))                      ; 4 goes on stack
(push (args 1))                        ; trap info for arity check
(allocate-frame)                       ; save stack frame
(procedure-tag? loc.3)                 ; trap if loc.3 isn't a procedure
(eval-applicate loc.3)                 ; call procedure at loc.3 with stack args
(halt)                                 ; end of computation
```

Figure 3.17: Result of Second Pass – Abstract Stack Machine Code

Chapter 4

A Calculational Approach to Compiler Generation

4.1 Introduction

As we have seen, partial evaluation is a useful technique for generating compilers from the semantics of a language. However, we have also seen that to do it well is not a trivial task. So far, it has been an iterative process, in which the interpreter is carefully crafted, and then tweaked repeatedly. We also are forced to make an explicit, manual, division between the static and dynamic inputs, which may require some insight into the structure of the source language that the implementor does not possess.

In this chapter, we introduce a calculational approach to compiler generation, which in some ways lies at the opposite spectrum of compilation techniques from partial evaluation. In this new, more mathematical technique, the interpreter is automatically factored into a run-time and compile-time component, based on the algebraic structure of the interpreter. This material is based on some influential papers by Erik Meijer [28, 29].

We also introduce in this chapter an abstraction method known as monads, which has recently become a subject of great research interest. It turns out that monads can be embedded in a calculational approach to compiler generation in a very natural and powerful way. We will show an example of this in the next chapter, using the R4RS example that we discussed previously in the context of partial evaluation.

We again start with an interpreter for a source language for which we wish to create a compiler. If we have access to both a denotational semantics for the source language, and an implementation of a functional language, then it becomes a straightforward task to write the interpreter by encoding the semantics in the functional language. If we work carefully and systematically, we will believe that the interpreter is a faithful representation of the semantics of the source language.

Since the interpreter is written in a functional style, it may execute less efficiently than an interpreter written in an imperative fashion. However, having a precise semantics for the source language gives a mathematical "handle" that we can use to map our inefficient interpreter to a more operational version.

Our goal, however, is to produce a compiler, not just a good interpreter. Therefore, our last step is to show how an interpreter can be "factored" into compile time and run-time components. Since each step has been a well-defined mathematical process, we can have confidence in the correctness of the compiler.

In this, and the next chapter, our meta-language changes from Scheme to Gofer, a popular dialect of the functional language Haskell. One of the major reasons that Gofer was developed was to investigate certain type classes, called constructor classes. These classes turn out to be a very useful abstraction technique when defining an interpreter that we wish to manipulate algebraically. The code given in `typewriter` font is exactly as it would appear written in Gofer.

To make this material easier to understand, we introduce the concepts by constructing a compiler for a toy language. Our example is adapted from Meijer[28].

4.2 Translating abstract syntax into categorical notions

In the world of category theory, we can define a *functor*, f, which is simply a type constructor, along with a map that takes a function $(a \to b)$ to $(fa \to fb)$. This map has the nice properties of preserving identities, and distribution over function composition:

$$\text{map id} = \text{id}$$
$$\text{map (f} \circ \text{g)} = \text{map f} \circ \text{map g}$$

We start by defining an functor that represents an abstract syntax tree for our source language. For our example, we define the following datatype:

```
data Expr a = Num Int | Add a a
```

We would like our source language to be represented as a tree, with integers at the leaves, and addition operations at the nodes. The datatype we have declared gives us one level of structure; to describe the recursive shape of an abstract syntax tree, we "tie the recursive knot" by defining:

```
data Rec f = Init (f (Rec f))
```

Note that the constructor Init takes objects of type `(f (Rec f))` to objects of type `(Rec f)`.

Now we declare a type synonym – our source language is the language of expressions, where the expressions are defined by recursion over the datatype `Expr`. This is done simply for ease of reference.

```
type Expression = Rec Expr
```

We can see how these pieces go together to create an abstract syntax tree for our source language of expression trees. Each branch of the Add constructor contains a copy of the type `Expression`

```
Expression = Rec Expr
           = Init (Expr (Rec Expr))
           = Init (Num Int) | Init (Add (Init (Expr (Rec Expr)))
                                        (Init (Expr (Rec Expr))))
           = Init (Num Int) | Init (Add Expression
                                        Expression)
```

Now imagine we were trying to define a simple function "eval" that would serve as an interpreter for source programs in the expression language we've described. It would probably look something like this:

```
eval (Init x) = case x of
         Num Int -> Int
                 Add e1 e2 -> (eval e1) + (eval e2)
```

Looking at the structure of this interpreter, we see that `eval` moves one level down the tree, while `+` collects terms that have been evaluated at a lower level. We will duplicate this functionality by explicitly defining two functions – one that moves down the tree, and one that moves up.

As mentioned above, functors are supposed to have a map. This is just what is needed to move down the tree one level:

```
map :: (a -> b) -> (Expr a -> Expr b)
map f e = case e of
            Num Int -> Num Int
            Add e1 e2 -> Add (f e1) (f e2).
```

`map` applies a function `f` one level down the tree. To define a function to move one level up the tree, we use the categorical notion of an *algebra*.

Given a functor `f`, a type `a`, and a function, `phi :: f a -> a`, we have an *f-algebra*. The type `a` is called the *carrier* of the f-algebra, and the functor `f` is its *signature*.

The language of expressions that we are using for an example has integer values at the leaves, so we define `phi` as follows:

```
phi :: Expr Int -> Int
phi a = case a of
          Num Int -> Int
          Add n m -> n + m.
```

One other algebra that will play a part in the sequel is called the *initial* algebra. Given any algebra `f`, we can define the following algebra (this explains the use of the constructr `Init` in the definition of the datatype `Rec f`):

```
init :: f (Rec f) -> Rec f.
```

Now we are ready to put all the pieces together to create an interpreter. Calling the interpretation function `eval`, we define:

```
eval :: Expression -> Int
eval (Init x) = phi (map eval x).
```

We see that `map` applies the interpretation function, `eval`, one level down the tree, while phi moves up the tree, collecting the values as it goes. Fortunately, the technique that we've just demonstrated is a general one. Given any other f-algebra with operation `phi :: f a -> a`, we can instantiate the following polymorphic type:

```
cata :: (f a -> a) -> (Rec f -> a)
```

to get a unique homomorphism:

```
cata phi :: Rec f -> a
```

This homomorphism we have just defined is called a *catamorphism*, and is simply a generalization of the fold operation over lists to a general datatype. This mapping is suggestive of an interpreter, which takes an abstract syntax tree (represented by `Rec f`), and returns an answer (of type `a`). That is, `cata phi = eval`, the normal top-level interpreter function.

The relationships can be depicted in the following diagram:

$$\begin{tikzcd}
\text{f(Rec f)} \arrow[r, "\text{Init}"] \arrow[d, "\text{map (cata phi)}"'] & \text{Rec f} \arrow[d, "\text{cata phi}"] \\
\text{f(a)} \arrow[r, "\text{phi}"] & \text{a}
\end{tikzcd}$$

Notice that this diagram is just the following equation (given a source program x):

```
cata phi (Init x) = phi (map (cata phi) x)
```

Substituting `eval` for `cata phi`, we get

```
eval (Init x) = phi (map eval x)
```

which is exactly the form of the interpreter we wrote for our language of expressions.

4.3 Calculating a Compiler

At this point, we have constructed a simple interpreter for a simple language of expressions, based on some concepts from category theory. However, the goal in this chapter is to calculate a compiler, and we have to show how to get there from here.

To build a compiler for a source language, we need to have a target language. To continue our example, we define a target language, and an interpreter for this language (i.e., a description of the run-time machine), using the same techniques we used for the source language.

Suppose that the target machine is a stack machine. We define a stack datatype, with three instructions. We can push integers on the stack, and perform an addition of the contents of the stack. Unlike the language of expressions, in which the evaluation order of subexpressions is undefined, we use `Seq` to control the order of evaluation.

```
data Stack a = Push Int | Add | Seq a a
```

We also define a `map` and an algebra `psi` for this datatype. In the definition of `psi`, let `push`, `add`, and `before` be primitive operations on a real stack machine (just as `+` was a primitive operation in the definition of `phi`.

```
map f x = case x of
            Push Int -> Push Int
            Add -> Add
            Seq s1 s2 -> Seq (f s1) (f s2)

psi x = case x of
          Push n -> push n
          Add -> add
          Seq s1 s2 -> s1 before s2
```

We will call the evaluation function for the stack language `run`, since it represents a run-time machine.

```
run :: Rec Stack -> Int
run (Init x) = psi(map run x)
```

We now have defined an interpreter for our source language, as well as our run-time machine. To produce a compiler for our source language, we need to define a function that allows us to map objects of type `Rec Expr` (i.e., expressions) to `Rec Stack`. The reason for developing the mathematical machinery of functors, algebras, catamorphisms, etc., is that once we have defined our source language and target machine this way, there exist theorems that enable us to derive a compiler.

Recall that the interpretation function for our source language can be written as a catamorphism:

```
eval = cata phi.
```

Intuitively, we'd like to divide the interpretation function into two parts: the compiler, and the run-time machine. That is, we'd like to find some function `t` such that:

```
phi = t psi
```

The idea is that just as phi evaluates one level of the source language, and psi evaluates one level of the target language, `t` converts one level of the target language into its source language representation. This is the "eureka" step for the whole process. Once `t` has been defined, there is a theorem that guarantees the existence of a mapping between `Rec Expr` and `Rec Stack`:

```
cata (t init) :: Rec Expr -> Rec Stack
```

This mapping, `cata (t Init)` is the compiler that we are looking for, and as the following diagram shows, we have factored the source language interpreter (`cata phi = cata (t psi)`) into a compiler, and a run-time machine.

$$\begin{tikzcd}
 & \text{Rec Stack} \arrow[dd, "\text{cata psi}"] \\
\text{Rec Expr} \arrow[ur, "\text{cata t Init}"] \arrow[dr, "\text{cata t psi}"'] & \\
 & \text{Int}
\end{tikzcd}$$

We can proceed to derive `(t init)` and `(t Psi)`. We know that `t` is a mapping between the stack and expression algebras; as our "eureka" step, we write the definition of `t` applied to an algebra `h`:

```
t :: (Stack a -> a) -> (Expr a -> a)

t h x = case x of
           Num n -> h (Push n)
           Add a b -> h (Seq a (h (Seq b (h Add))))
```

Now instantiate `h` with the algebra `psi`. We know that `(t psi) = phi`; what we get is an interpreter that evaluates expressions using the stack machine's instructions (this is easily confirmed by plugging in the definition of `psi` into the equation above.

```
psi :: Stack Int -> Int

t psi :: Expr Int -> Int

t psi x = case x of
             Num n -> push n
             Add e1 e2 -> e1 before (e2 before add)
```

Lastly, we derive the compiler itself, `(t init)`:

```
init :: Stack (Rec Stack) -> Rec Stack

t init :: Expr (Rec Stack) -> Rec Stack

t init x = case x of
             Num n -> Init (Push)
             Add e1 e2 -> Init (Seq a (Init (Seq b (Init Add)))).
```

We have shown how to develop a compiler starting from a functorial representation of a recursive datatype. However, there is a problem in trying to scale up this technique to a real programming language. Specifically, recall that in our formulation, an interpreter takes only one argument – the expression to be evaluated. "Real" interpreters take other arguments, such as environments, or stores, or continuations. In order to scale our techniques up, we will have to introduce a new abstraction method, called monads.

4.4 Monads

There has been a great deal of interest recently in the use of *monads* to help structure functional programs. Much of this interest is due to some influential papers written by Wadler [40, 39] popularizing their use. One of the drawbacks of working in a functional paradigm is that it is difficult to emulate "impure" constructs such as state and exceptions. Monads provide a way to do this, without sacrificing the benefits of functional programming.

Monads, beneath all of the hype, are simply a very useful form of data abstraction, and as we will see, common data constructs have very natural representation in monads. Our interest in monads is in using them as building blocks, in what are called monad transformers. The idea is use various monads in a modular fashion, as kind of a semantic (or monadic, if you like) Lego set. This allows us to build computational abstractions to reduce the complexity of an interpreter for a real programming language (like Scheme), to a very manageable size, enabling us to use the algebraic methods outlined earlier in the chapter.

A monad consists of a datatype `M`, and two operations, called *unit* and *bind*. The unit operation takes a value of type `a` into a computation over `a`, where a computation consists of a value and a set of effects (the latter being tied to the datatype `M`).

```
unit:: a -> M a
```

If the idea is to raise all values into the world of monads, or computations, then since effects happen in some particular order, we must have a way of specifying this order. This is where bind comes in.

```
bind:: M a -> (a -> M b) -> M b
```

We now look at some examples of monads. The simplest of them is just the identity monad:

```
type I a = a

unitI :: a -> I a
unitI a = a

bindI :: I a -> (a -> I b) -> I b
bindI m k = k m
```

This may not appear to do anything, but in the next chapter we will demonstrate its usefulness when we describe monad transformers.

We can define a monad for states (assuming that a `State` datatype has been defined):

```
type S a = State -> (a, State)
```

```
unitS :: a -> S a
unitS a = \s -> (a, s)

bind :: S a -> (a -> S b) -> S b
bindS m k = \s -> let (b, s1) be (m s) in (k b s1)
```

Note that \ stands for λ.. **unitS** returns a pair consisting of a value, and an unaltered state. **bindS** takes an initial state s and a value a, and returns a new value b, paired with a new state, s1.

Also of interest in the sequel is a monad **R** for environments, and for error messages, given the existence of datatypes for **Env** and **E**, respectively. In the environment monad, the environment is passed unaltered by the bind function. The error monad enables an error message to be propagated, say, to the top level of an interpreter.

```
type R a = Env -> a

unitR :: a -> R a
unitR a = \r -> a

bindR :: R a -> (a -> R b) -> R b
bindR m k = \r -> let b be (m r) in (\b -> k b r)

type E a = OK a | ErrMsg

unitE :: a -> E a
unitE a = OK (a)

bindE :: E a -> (a -> E b) -> E b
bindE m k = \a -> case a of
                     OK x -> k x
                     ErrMsg -> ErrMsg
```

In a similar fashion, monads can be defined for constructs like exceptions, continuations, and non-deterministic choice. Wadler gives a good description of these, and shows how they might be used in an interpreter [39].

4.5 Monad Transformers

When building an interpreter, we may have very complex computations to carry out. We could certainly try to build a complex monad custom-made for the problem at hand, but a better solution seems to be to develop a way that monads can be usefully combined. This idea is not a new one; Moggi [30] discussed structuring monads in such a way as to be able to extend them with other monads; he called these *monad transformers*. A paper by Liang, Hudak, & Jones [27] expands upon those ideas to show how interpreters can be constructed using monad transformers as modular building blocks, by embedding monads in each other to produce more complex monads. A monad transformer is simply a monad with a "hole" in it, so that another monad (or monad transformer) can be embedded in it. The result is a tower of monad transformers, in which each transformer is defined in terms of the embedded object. This technique can be demonstrated by a simple example. Consider again the state monad:

```
unitS a = \s -> (a, s)
bindS m k = \s -> let (b, s1) be (m s) in (k b s1)
```

We define a state monad transformer over the state monad by adding an extra parameter for the embedded monad, `M`. Let `unitM` and `bindM` be the unit and bind operations for the monad `M`. Then we define unit and bind for the state monad transformer as follows, using `unitSM` and `bindSM` to denote the state monad with monad `M` embedded:

```
unitSM a = \s -> unitM(a, s)
bindSM m k = \s -> bindSM (m s) (\(b, s1) -> (k b s1)
```

Note that this transformer has the property that if we take `M` to be the identity monad, `I`, as defined earlier, the `SM` monad becomes the state monad. The identity monad, then, gives us a "base case" for the embedding procedure. We build a complex monad by embedding a series of monad transformers over the identity monad. Obviously, we can invent an infinite number of monads, and monad transformers; the hope among researchers in this area is that the set of truly useful monads will be small enough that it doesn't require too much cleverness to combine them in ways that correspond to the language constructs that we wish to emulate.

4.6 Combining Approaches

What we would like to do is to combine the benefits of Meijer's calculational approach with the use of monad transformers. As we have seen, Meijer's technique is to define a recursive functor that captures the structure of source programs. In practice, an interpreter for any real language needs to carry information on environments and/or states, at the very least. What we would like is a way to "package up" all of this information in one parameter, and monad transformers offer us just that ability. In other words, the standard calculational approach is to produce a function `Eval` which takes a source program built recursively from functor `E` to the domain of, say, `Answers`:

```
Eval:: Rec E -> Answers
```

What we will do instead is to produce a function that maps instead to some monad transormer, `M`, over `Answers`:

```
Eval:: Rec E -> M Answers
```

By defining M as the proper set of monad transformers over the identity monad, we can incorporate information about the environment, state, etc., giving us the ability to write an interpreter for a non-trivial language. The next chapter discusses our experience writing a Scheme interpreter this way.

Chapter 5

Applying the Calculational Approach

5.1 Introduction

In this chapter, we write a Scheme interpreter using the structuring techniques set forth in the previous chapter. We start off by creating an appropriate monad, make some necessary type definitions, and then define a functor to represent Scheme source programs. We end with a brief discussion of how this interpreter could then be transformed into a compiler.

5.2 Building a Monad for a Scheme Interpreter

We first build a monad appropriate to the interpreter that we want to construct. In the R4RS specification, we see that the denotational semantics of Scheme requires inputs of a source text, an environment, a store, and a continuation. The definition also requires us to be able to do error reporting. We write a functor that captures the recursive shape of source text, and package the rest of the arguments in a monad. The only exception to this is the continuation argument; we handle sequencing explicitly by the bind function for the monad we define. Therefore, we must create a monad that is a combination of the monads for environment, store, and errors. To do this, we will use various monad transformers. We start by defining the identity monad:

```
type MonadI e = e

unitI :: a -> MonadI a
unitI x = x

bindI :: MonadI a -> (a -> MonadI b) -> MonadI b
bindI m k = k m.
```

We next embed the MonadI monad in an error monad transformer, creating a MonadEI monad:

```
data MonadEI e = OK e | Err Message

unitEI :: a -> MonadEI a
unitEI x = OK (unitI x)

bindEI :: MonadEI a -> (a -> MonadEI b) -> MonadEI b
bindEI m k = bindI m (\a -> case a of
                                 OK x -> k x
                                 Err msg -> unitI (Err msg)).
```

The MonadEI monad itself is now embedded in a state monad transformer, creating a MonadSEI monad:

```
type MonadSEI e = State -> MonadEI (e, State)

unitSEI :: a -> MonadSEI a
unitSEI x = \s -> unitEI (x, s)

bindSEI :: MonadSEI a -> (a -> MonadSEI b) -> MonadSEI b
bindSEI m k = \s -> bindEI (m s)  ( \(a, s1) -> k a s1).
```

Lastly, we take the MonadSEI monad, and embed it in an environment monad transformer, producing a MonadRSEI monad.

```
type MonadRSEI e = Env -> MonadSEI e

unitRSEI :: a -> MonadRSEI a
unitRSEI x = \r -> unitSEI x

bindRSEI :: MonadRSEI a -> (a -> MonadRSEI b) -> MonadRSEI b
bindRSEI m k = \r -> bindSEI (m r) ( \a -> k a r)
```

A little experimentation will show that monad transformers are not commutative, so the order in which we embed them is important. The environment monad was added last for a reason. Recall that at compile time, the environment is static. By putting it on the outside of the monad that we have created, it can be "stripped off" at compile-time, leaving a run-time monad of type MonadSEI.

5.3 Type Definitions

Our interpreter will make reference to structures such as identifiers, locations, formals, and environments. We provide type synonyms for these, as well as for states, and error messages, to enhance the readability of the interpreter. Environments are simply lists of identifier/location pairs, and States are lists of location/exprval pairs. Since the only constants we handle in this interpreter are integers, we make this explicit with a datatype definition.

```
type Id = String
type Loc = Int
type Env = [(Id, Loc)]
type State = [(Loc, Exprval)]
type Message = String
type Formals = [Id]
data Constant = Number Int
type Boolean = Exprval -- intended to be restricted to False and True
```

We define an expression datatype, E, that corresponds to Scheme's six basic forms (constant, variable, conditional, assignment, abstraction, and application), as well as to a couple of primitive functions that the interpreter recognizes (addition, and less-than).

```
data E e = Const Constant | Ident Id | Cond e e e
              | Add2 e e | Less e e | Set Id e
              | Lambda Formals (C e) e | Apply e (A e)
```

The definition of the expression datatype has references to two other datatypes: arguments (`A`), and commands (`C`). Each of these datatypes have essentially the same structure as a list, with the additional property that the last entry is tagged. Although both of these are expressions, we construct new functors for each of them that reflect the difference in the way that they are evaluated. That is, when we evaluate a sequence of arguments, we return a sequence with each evaluated argument, whereas when we evaluate a sequence of commands, we are only doing so for their side effects; the returned values are ignored. In R4RS, too, evaluation functions for both these datatypes are defined separately from expression evaluation.

```
data C e  = Com e (C e) | LastCom e | EmptyCom
data A e  = Arg e (A e) | LastArg e | EmptyArg
```

Next, we define an Exprval datatype that corresponds to the expressible values in the Scheme semantics.

```
data Exprval = Num Int | Closure (MonadRSEI Exprval) Env Loc
             | MyFalse | MyTrue | Undefined | Unspecified | Null
             | Args [Exprval]
```

These are the same expressible values used in the Scheme interpreter that we partial-evaluated; by using Gofer instead of Scheme as our metalanguage, we can make this datatype explicit.

Lastly, we "tie the recursive knot" with the following definitions, where Init is a dummy constructor.

```
data Rec f = Init (f (Rec f))
type Expr = Rec E
```

5.4 Defining Functors

Using Gofer's ability to define a constructor class, we create a class for functors, with the two operations mapf and phi. Note that instead of Exprval being an "atomic" type, we work with MonadRSEI Exprval as being our base type.

```
class Functor f where
       mapf :: (Expr -> MonadRSEI Exprval) -> f Expr -> f (MonadRSEI Exprval)
       phi :: f (MonadRSEI Exprval) -> (MonadRSEI Exprval)
```

Now we can create an instance of that functor that represents the structure of source programs. This means defining, for each Scheme basic form, the functions mapf (which traverses down the abstract syntax tree of the source program), and phi (which traverses back up the tree with the results of computations). Figure 5.1 contains the definition of this functor, E, which is the heart of the interpreter.

We can briefly describe the behavior of this functor. The function mapf is an indentity map for the two atomic Scheme basic forms: constants, and identifiers. For all of the other basic forms, as well as the two primtive binary routines (`Add2` and `Less`), we apply the function f to each subexpression.

The purpose of the function phi is to collect and sequence computations, and in fact, the code resembles CPS in some cases. Any subexpression in a basic form is assumed to be a computation. In the case of constants, we use unitRSEI to raise the constant to a computation. If we have a variable, we evaluate it, and in the process, raise the resulting value to a computation.

We can also define functors for argument lists, and lists of commands. They differ from each other, and from the expression functor, in how bind operates on them. For example, a list of commands is just evaluated for its side effects; a value of *unspecified* is supposed to be returned. In the context of our monadic approach, we use unitRSEI to raise *unspecified* from an Exprval to a computation over Exprval. Figure 5.2 shows the definition of these functors.

5.5 Auxiliary Functions

The functors that we have defined have referenced several auxiliary functions; since the monad used in this interpreter is made by combining monad transformers for environments, states, and error-reporting, it makes sense that we need routines to manipulate those three data structures. For example, we need to be able to do environment lookups, as well as extend the environment with new identifier/location pairs.

```
extendEnv :: Env -> [Id] -> Loc -> Env
extendEnv = \r formals freeloc -> case formals of
     [] -> r
     (x:xs) -> (extendEnv ((x,freeloc):r)  xs (next(freeloc)))

lookupEnv :: Id -> MonadRSEI Loc
lookupEnv idnt = \r -> \s -> case r of
       [] -> unitRSEI (-1) r s
```

```
instance Functor E where
 mapf f x = case x of
             Const const -> Const const
             Ident var -> Ident var
             Add2 e1 e2 -> Add2 (f e1) (f e2)
             Less e1 e2 -> Less (f e1) (f e2)
             Cond e1 e2 e3 -> Cond (f e1) (f e2) (f e3)
             Set idnt e -> Set idnt (f e)
             Lambda form coms e -> Lambda form (mapf f coms) (f e)
             Apply e args -> Apply (f e) (mapf f args)
 phi x = case x of
          Const (Number n) -> unitRSEI (Num n)
          Ident var -> evalIdent var
          Add2 e1 e2 -> bindRSEI e1
                         (\a -> case a of
                            Num int1 -> bindRSEI e2
                               (\b -> case b of
                                  Num int2 -> unitRSEI (Num (int1 + int2))
                                  _ -> makeErr "non-numeric arg to Add1")
                            _ -> makeErr "non-numeric arg to Add2")
          Less e1 e2 -> bindRSEI e1
                         (\a -> case a of
                            Num int1 -> bindRSEI e2
                               (\b -> case b of
                                  Num int2 -> unitRSEI
                                    (if (int1 < int2)
                                        then MyTrue
                                        else MyFalse)
                                  _ -> makeErr "non-numeric arg to Less1")
                            _ -> makeErr "non-numeric arg to Less2")
          Cond e1 e2 e3 -> bindRSEI e1
                            (\a -> case truish a of
                                   MyTrue -> bindRSEI e2
                                              (\b -> unitRSEI b)
                                   MyFalse -> bindRSEI e3
                                              (\b -> unitRSEI b))
          Set idnt e -> bindRSEI e (\v -> assign idnt v)
          Lambda formals coms e -> makeclosure
                                    (bindRSEI (phi coms)
                                      (\a -> bindRSEI e
                                             (\b -> unitRSEI b)))
                                    formals
          Apply e es -> bindRSEI (phi es)
                         (\a -> let (Args args) = a in
                                bindRSEI e
                                         (\clos -> applicate clos args))
```

Figure 5.1: The Expression Functor

```
instance Functor A where
 mapf f a = case a of
             Arg x y -> Arg (f x) (f y)
             EmptyArg -> EmptyArg
 phi a = case a of
          EmptyArg -> LastArg (unitRSEI (Args []))
          Arg x y -> bindRSEI x (\a -> bindRSEI y
                      (\b -> let LastArg (Args lst) = b in
                        LastArg (unitRSEI (Args (a:lst)))))

instance Functor C where
 mapf f  c = case c of
              EmptyCom -> EmptyCom
              Com x y -> Com (f x) (f y)
 phi c = case c of
          EmptyCom -> LastCom (unitRSEI Unspecified)
          Com x y -> bindRSEI x
                     (\a -> bindRSEI y
                      (\b -> LastCom (unitRSEI Unspecified)))
```

Figure 5.2: The Argument and Command Functors

```
        ((a,b):xs) -> if idnt == a
                         then unitRSEI b r s
                         else lookupEnv idnt xs s
```

Given these primitive environment functions, we define the function evalIdent. To mirror the structure of R4RS as closely as possible (at the risk of obfuscation), we have **evalIdent** call the function **hold**, a store lookup routine similar to the one defined in Chapter 3.

```
evalIdent :: Id -> MonadRSEI Exprval
evalIdent idnt = bindRSEI (lookupEnv idnt)
                  (\loc -> bindRSEI (hold loc)
                   (\a -> case a of
                     Undefined -> makeErr ("undefined " ++ idnt)
                     _ -> unitRSEI a))
                            lse holdhelper loc xs
```

Similarly, we have routines to do both store lookups and assignments:

```
updateState :: Loc -> Exprval -> MonadRSEI Exprval
updateState loc val = \r -> \s -> OK (Unspecified, (loc,val):s)

assign :: Id -> Exprval -> MonadRSEI Exprval
assign idnt val = bindRSEI (lookupEnv idnt)
                  (\loc -> if loc == -1
                             then makeErr "undefined variable"
                             else updateState loc val)

hold :: Loc -> MonadRSEI Exprval
hold loc = \r -> \s -> OK(holdhelper loc s,s)

holdhelper :: Loc -> State -> Exprval
holdhelper loc s = case s of
   [] -> Undefined
   ((a,b):xs) -> if loc == a then b else holdhelper xs
```

We also have a routine that takes an error message into a monad:

```
makeErr :: Message -> MonadRSEI Exprval
makeErr msg = \r -> \s -> Err msg.
```

Our interpreter also contains routines that make closures (`makeclosure`), and apply argument lists to closures (`applicate`).

Lastly, we define our top-level evaluation functions. The function `evalE` takes source program x (as an abstract syntax tree), traverses down the tree with `mapf`, and then back up with `phi`. To actually run the interpreter, though, we define a routine `interpE`, which takes the result of `evalE` (a very large function), and applies it to a actual environment and state. The result is a MonadEI containing the result of the computation, and the final state.

```
evalE :: Expr -> MonadRSEI Exprval
evalE (In x) = phi (mapf evalE x)

type Answer = MonadEI (Exprval, State)

interpE :: Expr -> Env -> State -> Answer
interpE m r s = (evalE m) r s
```

5.6 Discussion

By packaging up most of the interpreter's arguments in a monad, using a series of embedded monad transformers, we are able to write a compact Scheme interpreter that is both faithful to R4RS, and amenable to the algebraic manipulation discussed in the previous chapter. The interpretation function that we defined, `interpE`, simply applies an environment and a store to the expression evaluation function, `evalE`.

We would like to apply the transformation methods outlined in chapter 4 to the evaluation function, `evalE`, and calculate a Scheme compiler. However, there is still a obstacle to be overcome. Recall that the crux of the calculational method is defining an algebra for the source and target languages, and then creating a mapping between them, from which we can derive a compiler. In this technique, the signatures of the algebras are not required to be the same; in the simple example given in the previous chapter, the source, and run-time signatures were `Expression` and `Rec Stack`, respectively. However, it is necessary for the source and target algebras to have the same carrier; in our simple example, the type of the carrier was `int`.

Contrast this with the situation for the Scheme interpreter, in which the type of the carrier is `MonadRSEI Exprval`. We defined a source language algebra `phi :: Expr -> MonadRSEI Exprval`, and could define a run-time algebra `psi :: Machine -> MonadRSEI Exprval`, where `Machine` is the run-time machine functor. However, as we saw in the partial evaluation section of this paper, the environment is not an input to the run-time machine. The monad transformer that is appropriate for the run-time machine is `MonadSEI`, since the state, and error handling, are run time objects. Hence, the algebras we need don't share the same carrier.

Looking at the definition of `MonadRSEI` and `MonadSEI`, we can intuitively say that the relationship between them is very simple, letting r be the appropriate environment.

```
MonadSEI = (MonadRSEI r)
```

However, there is no reason to believe that transforming from one carrier to another will always be so simple. It may be the case that the structure of monad transformers is such that such a carrier transformation can easily be found, or derived straight from the definition of the monad transformers involved, but we have no proof of this. Our conclusion, then, is that the theory behind the calculational approach needs to be expanded to take this additional challenges into consideration. This will make the calculational approach more applicable to real-world languages.

Chapter 6

Conclusion

Perhaps the most obvious conclusion to be drawn from our work is that it is very hard to specify a language; denotational specifications are more successful as language descriptions than as implementation guides. For example, any commercial Scheme compiler, when it builds a closure, will only store that part of the environment that corresponds to the free variables in the lambda expression of interest. But looking over R4RS, we find no mention of that fact, simply because it is solely an implementation issue. Sadly, but not surprisingly, the truth seems to be that any kind of semantics directed compiler generation will not enable a person with an inadequate understanding of both compilers and language specifications to produce a good compiler. These techniques will only allow a poor compiler writer to efficiently generate a poor compiler.

It is worth briefly comparing the two approachs; which one does the job better? Our experience with partial evaluation demonstrated to us that this technique can be considered a feasible method of automatically generating a compiler for a source language, given an interpretive specification. However, the compiler didn't just simply "fall out" of the specification. It was necessary to experiment with the interpreter, and its binding time properties, before converging on a residual program in which all of the usual compile-time actions had been executed.

The obvious criticism of this technique is that the various transformations were too "ad-hoc", and that any "tuning" done to one interpreter would have no relevance to a different one. It was originally our hope that further research might identify a small set of transformations that, for any language, would guarantee good specialization behavior. Our analogy was CPS, a program transformation technique that, when first used in a compiler, may have looked like a "trick" of some kind, but now is recognized as an extrememly useful tool for any compiler writer. However, we cannot say with any confidence that we have identified any such set of transformations with "across-the-board" utility. Our summary of partial evaluation, then is that it has, compared to the calculational approach, both a pluses and minuses. The big plus, in our eyes, is that the ability to tweak makes the process feel more interactive, and therefore "easier". Also, the interpreters that we wrote didn't have to satisfy any particular mathematical properties, giving us more freedom while crafting them. There were some big minuses, however. The ability to tweak the interpreter, looking for "good" properties tended to encourage aimless experimentation ("Hey, I think I'll try residualizing this function; nothing else seems to do the trick!"). The fact that the partial evaluator is, essentially, a black box, means that it can often be difficult to understand and predict its behavior. Also, the very fact that the interpreter did not have to have any mathematical structure meant that there was no clear definition of what the compiled code should look like. How does one know when the partial evaluator has done the best it can? We attempted to overcome this problem by carrying out a detailed analysis of the binding time properties of the various functions that make up R4RS (described in detail in Chapter 3). It might have been faster to write the code to do the first compiler pass by hand.

The virtues and vices of the calculational approach are virtually the opposite of those of the partial evaluation approach. The calculational approach required more discipline when it came to writing the interpreter; using monad transformers for the first time was conceptually more difficult than just trying to translate denotational to its "computational cousin", Scheme. The comparison might even have come out worse for the calculational approach if we hadn't done the partial evaluation work first; when it came time to write an R4RS-based interpreter for our calculational experiments, we had already become old friends with R4RS.

The fact that we weren't relying on any "black boxes", and the requirement that the interpreter be mathematically structured, lent us more confidence in what we were doing, and where we were going, at each phase of the process. It has been remarked upon by many computer scientists that modularization, the ability to break a large, difficult problem (or programming project) into multiple small, understandable problems is one of the most useful tools in the problem solver's arsenal. It was our experience that conceptualizing an interpreter in terms of functors, algebras, etc., was a powerful form of modularization.

There are some very interesting areas for future work in this field. One fascinationg opportunity would be to investigate the merits of combining the two approaches. What would be gained by writing an interpreter in the calculational style, along with the use of transformers, and feeding that interpreter to a partial evaluator? If nothing else, the residual program might help suggest possible transformations between carriers and/or signatures. Multiple passes with the partial evaluator might translate into multiple refinements of the original interpreter. After all, going straight from source code to compiled code in one step is probably going to result in some very complex functors, and "eureka" steps between them. It seems like a better idea to do this in a stepwise fashion, letting the partial evaluator suggest "stepping stones" from source language to target language.

As mentioned in the previous chapter, the addition of monad transformers to the calculation approach originally outlined by Meijer calls for the development of new theory to support simultaneous transformations from the source language carrier and signature to the run-time version.

Another idea for further work would be expanding the techniques to include a very important aspect of a modern programming language that we have ignored thus far – the programming environment. Most modern programming languages have features such as error handlers, debuggers, garbage collectors, etc. Many of these languages are fundamentally interpreted, not compiled. Instead of starting with an interpreter, and generating a compiler, a more practical approach would be to start with an interpreter for the source language, and have the system generate a complete interpretive run-time system – an interpreter plus programming environment. This approach more closely reflects the realities of modern language design and development.

After having worked on the problem of automatic compiler generation for some time, we started to wonder about the existence of a "conservation of difficulty" law that operates in the universe of compiler construnction. Regardless of technique, it seemed that all of them relied on some kind of eureka step. For the partial evaluation approach, this meant the user had to specify the division between the static and dynamic inputs, as well as decisions to unfold or specialize functions. In the case of the calculational approach, the run-time machine/compiler factorization of an interpreter was dependent upon a manually-supplied function. The moral of the story seems to be that if you provide the 1 percent inspiration, the compiler generator will provide the 99 percent perspiration! Our hope is that by structuring the problem properly, we can make that 1 percent contribution attainable to any (inspired) compiler writer.

Bibliography

[1] J. Backus. Fortran. In J. Sammet, editor, *1st ACM Conference on the History of Programming Languages*, pages 25–29, 1977.

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

[3] A. Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[4] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–95, 1991.

[5] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *FPCA'93, Sixth International Conference on Functional Programming Languages and Computer Architecture*, pages 308–17, 1993.

[6] William Clinger. The Scheme 311 Compiler; An Exercise in Denotational Semantics. In *Confer. Rec. 11th ACM Symposium, Principles of Programming Languages*, pages 356–364, 1984.

[7] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language scheme. *Lisp Pointers*, Vol. 4 No. 3:1–55, 1991.

[8] C. Consel, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Yale University, 1992. Research Report 909, Department of Computer Science.

[9] C. Consel. Report on Schism'92. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, 1992.

[10] C. Consel. A tour of Schism: A partial evaluation system for higher order applicative languages. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, 1992.

[11] C. Consel. polyvariant binding time analysis for applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[12] C. Consel and O. Danvy. Static and dynamic semantics processing. In *ACM Symposium on Principles of Programming Languages*, pages 14–23, 1991.

[13] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, 1993.

[14] C. Consel and S.C. Khoo. Semantics-directed generation of a prolog compiler. In *Third International Symposium on Programming Language Implementation and Logic Programming*, volume 528, Lecture Notes in Computer Science, pages 135–146. Springer-Verlag, 1991.

[15] Pascal Fradet and Daniel Le Métayer. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Languages and Systems*, Vol. 13 No. 1:21–51, 1991.

[16] Daniel P. Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[17] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[18] S. Harnett and M. Montenyohl. Towards efficient compilation of a dynamic object-oriented language. In C. Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, 1992.

[19] P. Hudak and N. Jones, editors. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, 1991.

[20] H. F. Jacobsen. Speeding up the backpropagation algorithm by partial evaluation. DIKU Student Project 90-10-13, DIKU, University of Copenhagen, 1990.

[21] N. Jones and D. Schmidt. Compiler generation from denotational semantics. In *Semantics-Directed Compiler Generation*, volume 94, Lecture Notes in Computer Science, pages 70–93. Springer-Verlag, 1980.

[22] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[23] N.D. Jones, P. Sestoft, and H. Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[24] J. Jorgensen. Generating a compiler for a lazy functional language by partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 258–268, 1992.

[25] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *ACM Symposium on Principles of Programming Languages*, pages 281–92, 1989.

[26] S.C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 211–22, 1991.

[27] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM Conference on Principles of Programming Languages, 1995*, pages 211–224, 1995.

[28] Erik Meijer. Bananas in space: Extending fold and unfold to exponential types. In *ACM Symposium on Functional Programming and Computer Architecture*, pages 137–149, 1995.

[29] Erik Meijer. More advice of proving a compiler correct: Improve a correct compiler. Dept. of Computer Science, Utrecht University, 1995.

[30] Eugenio Moggi. An abstract view of programming languages. Technical report ecs-lfcs-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[31] P. Mosses. SIS — semantics implementation system, reference manual and user guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.

[32] Peter D. Mosses. Unified algebras and action semantics. In *STACS'89*, pages 17–35, 1989.

[33] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[34] B. Rytz and M. Gengler. A polyvariant binding time analysis. In C. Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28, 1992.

[35] S. Safra and E. Shapiro. Meta interpreters for real. In *Information Processing 86, Dublin, Ireland*, pages 271–8. North-Holland, 1986.

[36] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* Wm. C. Brown Publishers, 1988.

[37] P. Sestoft. Automatic call unfolding in a partial evaluator. In *Partial evaluation and Mixed Computation.* North-Holland, 1988.

[38] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[39] Philip Wadler. The essence of functional programming (invited talk). In *ACM Conference on Principles of Programming Languages, 1992*, pages 1–14, 1992.

[40] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computationo*, Vol. 7, No. 1:39–55, 1994.

[41] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Trans. Programming Languages Systems*, Vol. 4, No. 3:496–517, 1982.

[42] M. Wand. Semantics-directed machine architecture. In *Confer. Rec. 9th ACM Symposium, Principles of Programming Languages*, pages 234–241, 1982.

[43] M. Wand. Loops in combinator-based compilers. *Information and Control*, Vol. 57, Issues 2 and 3:148–164, 1983.

[44] M. Wand. A semantic prototyping system. In *Proc. ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 213–221, 1984.

[45] M. Wand. From interpreter to compiler: A representational derivation. In N. Jones, editor, *Workshop on Programs as Data Objects*, pages 306–324. Springer-Verlag, 1986.

[46] Mitchell Wand. The Register-Closure Abstract Machine: a Machine Model for CPS Compiling. Technical Report NU-CCS-89-24, Northeastern University, College of Computer Science, 1989.

Biographical Note

I was born in Burlington Vt., on April 21, 1961. I earned a B.S.M.E at Lehigh University in 1983, where I was a member of Pi Tau Sigma (national honorary mechanical engineering fraternity). From 1987 to 1990 I pursued graduate work in pure mathematics, while employed as a Lecturer in mathematics at California Polytechnic University, Pomona. In 1991 I received a Battelle fellowship to pursue graduate work in computer science at OGI.