Flexible and Practical Flow Analysis for Higher-Order Programming Languages

by

J. Michael Ashley

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

May 1996

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

R. Kent Dybvig, Ph.D.

Daniel P. Friedman

Peter Shirley

Jon Barwise

April 12, 1996

Flexible and Practical Flow Analysis for Higher-Order Programming Languages

Copyright 1996
by
J. Michael Ashley

Acknowledgements

In retrospect I will consider my graduate career to be one of the most lively and exciting times of my life. It could be no less considering the contributions and support of so many outstanding colleagues.

Dan Friedman and Kent Dybvig both supported me as advisors. Dan taught me to strive for clarity and simplicity without regard for realistic implementation. Kent taught me when and how to take realistic implementation into consideration. Their advice was invaluable, and they did their best to prepare me for independent research.

Colleagues at the institutions I visited exposed me to different perspectives and broadened my background. Joe Fasel, Kathleen Kelly, and Bob Hiromoto supported my work at Los Alamos and encouraged my interest in the parallel execution of functional programs. Gregor Kiczales, John Lamping, and Mike Dixon at Xerox PARC guided me through the problems of embedded computation and open implementation. My work on open compilers in particular motivated my interest in correct compilers and program translators. Charles Consel played a particularly important role in my training by forcing me to focus on one problem and see it through to publication. Working with him at Oregon Graduate Institute marked my transition between a student learning and a researcher contributing.

Professional colleagues helped with my training, but others kept me sane during the inevitable stress of graduate work. My parents supported me without question, and the Indiana University Computer Science Department staff helped with the technical and administrative problems I had to solve. Oscar Waddell showed me that kitchen recipes are not algorithms but heuristics. Susan Fox was a willing confidant in a particular romantic pursuit. Rick Bales, Brad Ramsey, Shan Bell, and David Moffett were great weight lifting partners. Finally, Michelle Porter and I got married late in my graduate career, and as my friend and my wife she made me realize it was time to finish and move on.

Abstract

A flow analysis is a procedure for computing static information about programs. Such an analysis can infer information about a program even if the program does not terminate when executed. This is done by abstracting over the values computed and the primitives used to manipulate those values. The information inferred is necessarily approximate but can still be used to justify the correctness of program transformations like partial evaluation, type check elimination, and register allocation.

Work on flow analyses for higher-order languages has identified several important aspects of the flow analysis problem. To date, however, no single flow analysis has been used for the practical flow analysis of mostly-functional languages. To be practical, an analysis must be able to analyze the full language, be accurate enough to justify program transformations such as compiler optimizations, and be acceptably fast for even large programs.

In this dissertation we develop a new flow analysis framework that unifies and extends previous work to yield an analysis that is indeed practical. The analysis framework can express a wide variety of analyses that vary in accuracy, speed, and the information they collect. The framework is proved correct using a novel application of a proof technique developed for proving the correctness of compilers. The analysis is fully implemented in a production-quality Scheme compiler. The analysis is used by the compiler to justify optimizations such as loop recognition, procedure call optimization, and closure elimination. Benchmarks demonstrate that the analysis is able to analyze large programs quickly yet retain enough accuracy to justify optimizations.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

Organization . . . 4

2 Background 5

2.1 Flow analysis for first-order languages . . . 5

2.2 Scheme-like languages . . . 7

2.3 Abstract interpretation . . . 8

2.3.1 Foundations . . . 9

2.3.2 Expressiveness . . . 13

2.3.3 Implementation techniques . . . 15

2.4 Summary . . . 19

3 An operational semantics for Scheme 21

3.1 Core Scheme . . . 22

3.2 Normalizing programs . . . 23

3.3 Abstract machines . . . 26

3.4 A CESK machine for A-Normal form . . . 27

3.5 Conversion to continuation-passing style . . . 28

3.6 A CES machine for CPS A-normal form . . . 33

3.7 Summary . . . 35

4 A flow analysis framework 36

4.1 A collecting machine for CPS A-normal form . . . 36

4.2 An abstract machine for CPS A-normal form . . . 39

4.3 Regulating speed and accuracy . . . 44

4.4 Summary . . . 47

5 Correctness **48**
5.1 Revised storage layout relations . . . 49
5.1.1 Storage layout relations . . . 50
5.1.2 Revised storage layout relations . . . 52
5.2 Proving the abstract machine correct . . . 54
5.2.1 Consistency . . . 56
5.2.2 Formalizing the abstract machine's behavior . . . 58
5.2.3 The correctness theorem . . . 61
5.3 Summary . . . 62

6 Implementation **64**
6.1 Overview . . . 64
6.1.1 Accommodating free variables . . . 65
6.1.2 Polyvariance . . . 68
6.1.3 Projection . . . 68
6.2 Implementing the abstract store . . . 69
6.2.1 A simple implementation . . . 69
6.2.2 Computing the least upper bound lazily . . . 72
6.2.3 Taking advantage of immutable locations . . . 79
6.3 Summary . . . 81

7 Applying the analysis **83**
7.1 Procedure representation and calling convention . . . 83
7.2 Instantiating the analysis . . . 87
7.3 Experiments . . . 88
7.4 Results . . . 90
7.5 Summary . . . 96

8 Conclusions **97**
8.1 Related work . . . 98
8.2 Future work . . . 100

Bibliography **102**

List of Figures

2.1 A simple program and its control-flow graph. 6
2.2 An example flowchart program. 11

3.1 The language *CS* of core Scheme. 22
3.2 The language $A(CS)$ of A-normal form. 23
3.3 The procedure *copy* and its equivalent in A-normal form. 24
3.4 An algorithm to translate from *CS* to $A(CS)$. 25
3.5 Auxiliary procedures for A-normalization. 26
3.6 Domains and state transition relation for the intermediate machine. 29
3.7 The language $CPS(A(CS))$ of CPS A-normal form. 30
3.8 A translator from $A(CS)$ to $CPS(A(CS))$. 31
3.9 Auxiliary procedure for the CPS translator. 32
3.10 The procedure *copy* in CPS A-normal form. 32
3.11 Domains and transition relation for the standard machine. 34

4.1 Domains and transition relation for the collecting machine. 38
4.2 A portion of the cache computed by the collecting machine for the procedure *copy*. 38
4.3 Domains and transition relation for the abstract machine. 42
4.4 Auxiliary functions used by the abstract transition relation. 43
4.5 A portion of the cache computed by the abstract machine for the procedure *copy*. 44
4.6 Revised auxiliary function *apply* for the abstract machine. 46

5.1 Illustration of the stuttering relationship between a concrete and abstract machine when using storage layout relations. 52
5.2 Illustration of the more complex stuttering relationship between the collecting and abstract machine when using revised storage layout relations. 53

6.1 A procedure for handling escaping values. 67
6.2 Straightforward implementation of the least upper bound operator. 69
6.3 Revised algorithm which avoids unnecessary set unions. 71
6.4 Store representation for the second algorithm. 73
6.5 A small example program and its CPS A-normal form equivalent. 74

6.6 An example flowgraph. 75
6.7 Operations for store lookup using the second algorithm. 76
6.8 Auxiliaries for store lookup using the second algorithm. 77
6.9 Operations for procedure application using the second algorithm's store data type. 78
6.10 Hybrid algorithm which uses a global store for variables that are assigned once. 80

List of Tables

3.1 Notational conventions. 27

7.1 Benchmarks in addition to the Gabriels. 89
7.2 Compile times and run-time speedups for all benchmarks. Run time speedups are improvements over the unoptimized benchmarks. 91
7.3 Static measurements of the compiler's ability to optimize the representation of procedures and eliminate argument count checks. 92
7.4 Static measurements of the compiler's ability to optimize procedure call sequences on the caller's side. All counts are the number of operations inserted into the code. 93
7.5 Dynamic measurements of operations eliminated compared to the unoptimized benchmarks. Statistics are given as percentage of operations eliminated. 95

Chapter 1

Introduction

This dissertation supports the thesis that the flow analysis of higher-order, mostly-functional languages can be made flexible and practical for use in compilers and interactive programming environments.

A flow analysis computes control- and data-flow information about programs. The flow information about a program can be represented as a graph that describes the execution of the program. The flow graph of a program is generally a finite approximation to the exact flow graph, because the program may not terminate when executed or it may use higher-order procedures. Some analyses, *e.g.*, a type checker, compute an approximate value for each expression in the program. Other analyses establish contextual information about statements in a program, *e.g.*, the set of variables live from a given statement.

Flow analyses are essential for justifying many types of program transformations. They have been applied mostly in the context of compiler optimizations. In that context, flow information has been used to justify the correctness of optimizations like dead code elimination, useless variable elimination, register allocation, copy propagation, constant folding, automatic parallelization, and more. Flow analyses have also been used to justify source-to-source transformations like partial evaluation [6, 19], closure conversion [55], type check elimination [37], and touch elimination [25]. Transformations that require flow information are important, because they can transform a program written to solve a problem cleanly into a new program that, for example, is more efficient and fault tolerant. Ultimately, flow analysis-based transformations can boost software reliability and programmer productivity.

For a traditional language like Fortran, designing a flow analysis is a well-understood process [1]. The control-flow graph is constructed from the text of the program, and the

data-flow information is computed using the control-flow graph. This strategy works because traditional languages do not support or encourage the use of first-class procedures or control operators. Since procedures and branch points are named, a sufficiently precise control-flow graph can be constructed from the syntax of the program alone.

The situation is not so simple for higher-order languages. Higher-order languages support procedures and continuations as first-class citizens. They can be passed to other procedures, stored in data structures, and otherwise treated as ordinary values. This makes constructing a control-flow graph difficult, since data-flow information is needed to determine what procedures may be applied at a given call site. To compute the data-flow graph, however, traditional techniques require that the control-flow graph be available. This would appear to be a dilemma.

One possible solution is to compute the control-flow graph as precisely as possible by computing the data flow of procedures only. Afterwards, traditional data-flow analysis techniques can be used to compute data-flow information for other values in the program. This was the approach taken by Shivers in his thesis on the control-flow analysis of higher-order languages [49]. A solution which builds more precise graphs is to combine the control- and data-flow analyses into one flow analysis. This was the approach taken by Harrison [32].

Both Shivers and Harrison showed that the flow analysis of higher-order languages was feasible, and this was an important achievement. Work since then has identified several important aspects of the flow analysis problem for higher-order languages.

- the accurate treatment of data structures, including mutable data structures,
- the use of type tests to constrain abstract values,
- the use of polyvariance to increase accuracy,
- the use of projection (widening) to accelerate convergence to a solution.

To date, however, no single flow analysis has been used for the practical flow analysis of mostly-functional languages. To be practical, an analysis must be able to analyze the full language, be accurate enough to justify program transformations (including compiler optimizations), and be acceptably fast for even large programs. In this dissertation we develop a new flow analysis framework that unifies the above aspects and extends previous work to yield an analysis that is indeed practical.

The analysis framework can express a wide variety of analyses that vary in accuracy, speed, and the information they collect. The framework is parameterized over a polyvariance operator, a projection operator, and an abstraction operator. The abstraction operator is used to specify the type of information, *e.g.*, type or range information, collected by the analysis. Different polyvariance operators can be used to regulate accuracy, and different projection operators can be used to regulate speed.

The analysis is fully implemented in the *Chez Scheme* [13] compiler, a production compiler for the Scheme language. To our knowledge, our analysis is the first to be implemented in a production-quality environment. Previous analyses have a worst-case complexity of $O(n^3)$ or greater with respect to the size of the input program. Our base analysis has similar complexity bounds. While average-case running times may be better than $O(n^3)$, it is important to be able to guarantee better worst-case bounds for a general-purpose analysis that may be deployed in a variety of environments. For example, since *Chez Scheme* uses an incremental compiler it is important to guarantee fast compile times. To this end we demonstrate a projection operator that can improve the running time of the analysis to be acceptable for interactive use. In particular, the worst case complexity is reduced to $O(n^2)$ in general and $O(n)$ for programs that cause no side effects. While most work on projection operators has been theoretical, our projection operator demonstrates that practical operators can be devised and deployed in practice.

The framework is proved correct using a novel application of a proof technique known as *storage layout relations* [30]. It is a technique for reasoning about the correctness of one operational semantics with respect to another operational semantics. It was developed for proving the correctness of operational semantics that address more details of an implementation than their counterparts. In this context, the analysis framework is cast in terms of an abstract operational semantics and proved correct with respect to a standard operational semantics. The proof requires small modifications to the storage layout relations technique, but the spirit of the approach remains the same.

The analysis is used to justify optimizations in the compiler. It collects information to justify the optimization of loops, procedure calls, and closure representations. These optimizations have been addressed by Rozas [47], but this work improves on this previous effort. A weak, intraprocedural instantiation of our framework that operates in one pass can still do a good job of optimizing loops, procedure calls, and closure representations.

Organization

The rest of this dissertation develops the flow analysis framework and elaborates on these contributions. Chapter 2 provides sufficient background material to understand the technical details of the dissertation. Chapter 3 develops a standard operational semantics for Scheme that serves as the basis of the flow analysis framework. In Chapter 4 the framework is developed, and it is proven correct in Chapter 5. Chapter 6 describes the implementation of the flow analysis framework in a production Scheme compiler. Chapter 7 describes how the framework is used to justify compiler optimizations such as procedure call optimizations and type check elimination. Chapter 8 describes related work and conclusions.

Chapter 2

Background

In this chapter the traditional approach for the analysis of first-order language is described followed by a history of abstract interpretation-based analyses. The history focuses on developments of expressive frameworks and efficient implementations in the context of functional and mostly-functional languages. Knowledge of the traditional approach to dataflow analysis and the history of abstract interpretation-based analyses provide the context in which this dissertation is situated.

2.1 Flow analysis for first-order languages

A first-order language is a language in which all procedures are named and cannot be passed to or from other procedures. Intraprocedural flow analysis for first-order languages is done in two steps. First a control-flow graph is constructed and then a data-flow analysis is done in terms of the control-flow graph. The control-flow graph is built from the syntax of the program. The nodes are basic blocks, and arcs represent possible control-flow paths from block to block. An example control-flow graph appears in Figure 2.1. The data-flow analysis is performed by generating and solving a system of equations that relate information between statements in the program. A typical equation has the form

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

which says that the information at the end of a statement S is the combination of the information generated from S and the information available when control enters S, minus

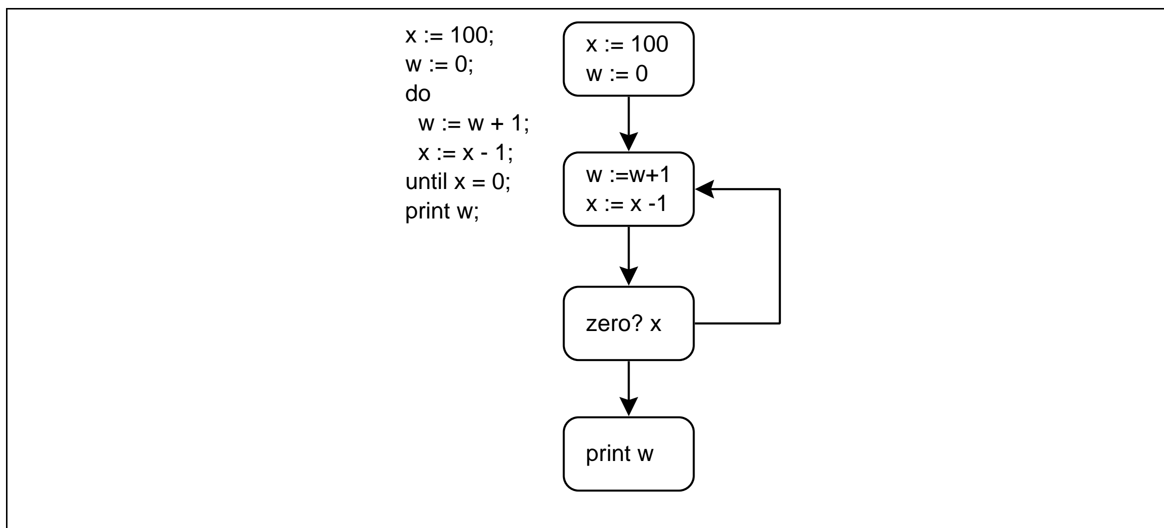


Figure 2.1: A simple program and its control-flow graph.

any information invalidated by the execution of S. The information collected depends on the application of the analysis.

Another aspect of the data-flow analysis is whether it is a *forward* or *backward* analysis. A forward analysis propagates information forward through the control-flow graph. A backward analysis propagates information the other way.

Whether the analysis is forward or backward dictates how *in* and *out* are defined. For a forward analysis, $in[S_i]$ is the union of $out[S_j]$ for all j such that there is an arc from S_j to S_i in the control-flow graph. For a backward analysis, $out[S_i]$ is defined similarly in terms of $in[S_j]$ for all j such that there is an arc from S_i to S_j.

This approach to data-flow analysis can be generalized to interprocedural analyses [1]. A separate control-flow graph is built for each procedure. Some of the data-flow equations, however, are defined in terms of equations from other procedures. For example, if S_0 is the first statement of a procedure F, $in[S_0]$ is defined in terms of the equations at the calls to F. Likewise, at a call site S_i which calls F, $out[S_i]$ will be defined in terms of $out[S_n]$, where S_n is the last statement of F.

This framework depends on being able to construct a control-flow graph before initiating data-flow analysis. This is an acceptable restriction when analyzing C or Fortran, since in Fortran the procedure being called is known at every call site, and in C, the procedure is known at nearly every call site, in practice. As we will see in the next section, this is *not* an acceptable restriction when analyzing higher-order languages.

2.2 Scheme-like languages

Scheme [17, 23] is a lexically-scoped dialect of Lisp. The language is small compared to many traditional languages and other Lisp dialects, but despite its small size it is expressive. It supports, in particular, first-class procedures and continuations. Implementations are also required to be properly tail-recursive. This means that if a procedure call is the last thing to be done in the evaluation of a procedure body, the current activation record is discarded in place of the callee's activation record. The consequence of this is that loops can be expressed as recursive procedures; special looping constructs are not necessary and can be excluded from the language.

First-class procedures in Scheme are more general than function pointers in C. In C, procedures are created once at either compile- or load-time, and they do not need to maintain a lexical environment to reference free variable bindings. In Scheme, on the other hand, many procedures may be created dynamically from one procedure definition. Each procedure must therefore maintain the lexical environment that existed at the time the procedure was created in order to reference free variable bindings.

Besides a more general notion of procedure, Scheme encourages the programmer to use procedures differently from C. In C, procedures often consist of a large amount of code with a few procedure calls that each do a lot of work. In Scheme, procedures are often much smaller and the frequency of calls is often much greater.

First-class continuations in Scheme enable powerful control constructs to be defined. A first-class continuation is obtained by reifying the current continuation using the operator **call/cc**. The reified continuation represents the "rest of the computation" from the point in which it was captured. The reified continuation appears as a procedure and behaves like one, except when it is applied. When applied, the current continuation is aborted and the saved continuation is used in its place. The effect is that control jumps to the point at which the invoked continuation was captured. The primitive **call/cc** operator is sufficient for implementing a wide variety of control operators, *e.g.*, engines [33] and threads. C does not have an operator comparable to the expressive power of first-class continuations.

Traditional flow analysis techniques cannot support first-class procedures and continuations in general. Traditional techniques depend on being able to construct a control-flow graph from the syntax of the program. Although C supports function pointers, they are used infrequently in most C programs. As a result flow analyses for C can afford to treat

them with safe but inaccurate approximations. This cannot be done for Scheme, since only small control-flow fragments can be constructed directly from program syntax, and they cannot be related unless there is knowledge of which procedures are called from which call sites.

As a result, conventional flow analysis techniques cannot be used for the practical flow analysis of mostly-functional languages. We need a stronger technique. The one used for the flow analysis in this dissertation is abstract interpretation.

2.3 Abstract interpretation

Suppose you are driving from Bloomington, Indiana to New Orleans, Louisiana. You are going to Mardi Gras, and you have a friend in Tupelo, Mississippi that is going to come with you. You need to determine when you will arrive in Tupelo so your friend will have an idea of when to expect you.

You have several ways to gauge how long it will take to drive from Bloomington to Tupelo. One way is to get in your car and drive there. That will give you a good, indeed exact, idea of how long it takes to make the drive. Another way is to use a road map. By computing mileages and estimating average speed, you can determine the trip time within some margin of error.

Both methods are feasible, but the latter is certainly the better choice. The reason is that while it gives a less precise driving time, it gives a *sufficiently precise* driving time. Your friend does not need to know that you will be there in eight hours and forty-three minutes. He needs to know you'll be there in about nine hours, as opposed to six or fourteen. The principle at operation in this example is *approximation*. We do not need to know every detail of the trip to estimate the trip time. We just need the mileage and some idea of your average speed.

The same principle can be applied to mathematical problems. One can determine whether a binary number is even or odd by looking at the least significant digit. The other digits can be ignored. Similarly, the sign of a number is all that is needed to know if its square root is an imaginary number.

A more substantial example is the "rule of signs" for arithmetic operators. The question we would like to answer is: "Given two numbers, what is the sign of their product (or sum)." To answer this question, we do not need to know the exact values of the two numbers.

We just need to know if the numbers are 0, positive, or negative. Here is the table for multiplication.

$*$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

We might try to build a similar table for addition.

$+$	$-$	0	$+$
$-$	$-$	$-$	$\top$
0	$-$	0	$+$
$+$	$\top$	$+$	$+$

The problem is that taking the sum of a positive and negative number produces a number with an indeterminate sign. The number might be zero, positive, or negative! We cannot choose a sign randomly, because it will be the wrong sign for the sum of some numbers. We must approximate the sign, which is done using the symbol $\top$ (read "top") in this example.

The idea of abstract interpretation is to extend the idea of abstracting over processes to the execution of computer programs. Instead of executing a program directly, we can compute a property of the program via an abstract interpretation of the program. A type checker for Pascal is an example of such an interpretation. It does not actually run a program but instead does an abstract execution to compute the type of each expression.

In the next subsection we give a more detailed introduction to abstract interpretation by reviewing seminal work. After the introduction, an overview of the field is given from two perspectives. We first review how abstract interpretation has been applied to the static analysis of programming languages. We then cover work that has been done on the implementation of abstract interpreters.

2.3.1 Foundations

The foundations of abstract interpretation were developed by Cousot and Cousot [20]. In their work they considered a simple flowchart language. The language was given an operational semantics defined in terms of a standard semantic domain. Their idea was that

static properties of a program P could be determined by interpreting P using a nonstandard semantic domain instead.

Their approach is straightforward. The operational semantics is a state machine that maps a flowchart node and an execution state to another flowchart node and execution state. The execution state is a function mapping variables to values. The values are elements of the standard semantic domain.

The first step to computing static properties is to define a *static (*or *collecting) semantics* for the language. The static semantics determines a context vector for a given program P. The context vector associates a set of execution states with each arc (called a *program point*) of P. In particular, for a given program point q, the context vector associates with q the set of those states that arise at q during the execution of P.

The static semantics is then abstracted over the standard semantic domain to yield an *abstract interpretation framework*. The framework is instantiated by supplying an abstract semantic domain and interpretations for the basic instructions in terms of that domain.

When instantiated, the framework yields an abstract interpreter that determines a static property of programs input to it. The abstract interpreter does this by computing an abstract context vector for an input program P. For each program point q of P, the abstract context vector associates with q a safe approximation of the states that occur at q during the standard execution of P.

Since the abstract interpreter is manipulating elements of a domain, some elementary concepts from domain theory are useful to introduce. A *partial order* (often called a "domain") $(P, \sqsubseteq)$ is a set P on which there is a binary relation $\sqsubseteq$ that is reflexive, transitive, and asymmetric. The least upper bound of two points a and b is the least element c such that $a \sqsubseteq c$ and $b \sqsubseteq c$. The least upper bound of a set A is the least element b such that for all $a \in A$, $a \sqsubseteq b$. A *complete partial order* is a partial order $(P, \sqsubseteq)$ for which every directed subset A of P has a least upper bound. Given a partial order $(P, \sqsubseteq)$, the *lifted partial order* $P_\perp$ is obtained by introducing a new element $\perp$ and extending $\sqsubseteq$ such that for all $a \in P$, $\perp \sqsubseteq a$. A function f between two partial orders X and Y is *monotonic* if and only if for all $a, b \in X$, if $a \sqsubseteq b$ then $f(a) \sqsubseteq f(b)$. A function f between two complete partial orders X and Y is *continuous* if and only if it is monotonic and for any subset A of X, $\bigsqcup_{a \in A} f(a) = f(\bigsqcup_{a \in A} a)$, *i.e.*, the behavior of f is consistent on least upper bounds. Further details on these definitions can be found in any number of texts. Winskel [56] provides an introduction in the context of programming language semantics.

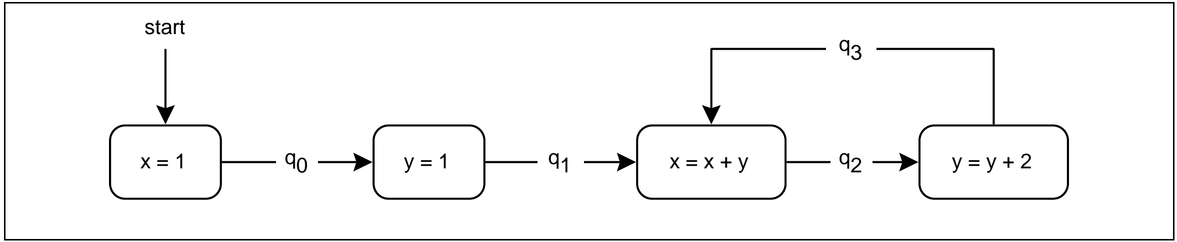


Figure 2.2: An example flowchart program.

As an example of an abstract interpreter, a type inferencer can be obtained as follows. Suppose the standard semantic domain consists of integers and booleans. The abstract domain might be the lifted domain $A = \{integer, boolean\}_\perp$ with the partial order $\perp \sqsubseteq integer$ and $\perp \sqsubseteq boolean$, and the interpretations for basic instructions would be defined in terms of A. The sum instruction, for instance, normally maps two integers to an integer. The abstract sum instruction would instead map two values of type *integer* to type *integer*.

Consider the program in Figure 2.2. The static semantics would compute the following context vector CV.

$$
\begin{aligned}
q_0 &= \{\{x \leftarrow 1\}\} \\
q_1 &= \{\{x \leftarrow 1, y \leftarrow 1\}\} \\
q_2 &= \{\{x \leftarrow 2, y \leftarrow 1\}, \\
&\quad \{x \leftarrow 5, y \leftarrow 3\}, \\
&\quad \{x \leftarrow 10, y \leftarrow 5\}, \\
&\quad \ldots\} \\
q_3 &= \{\{x \leftarrow 2, y \leftarrow 3\}, \\
&\quad \{x \leftarrow 5, y \leftarrow 5\}, \\
&\quad \{x \leftarrow 10, y \leftarrow 7\}, \\
&\quad \ldots\}
\end{aligned}
$$

Each state is represented as a set of variable bindings. An infinite number of states are associated with arcs q_2 and q_3.

The type inferencer, on the other hand, would compute the following abstract context vector $\widehat{CV}$. $\widehat{CV}$ computes just one abstract state for each program point.

$$
\begin{aligned}
q_0 &= \{x \leftarrow integer\} \\
q_1 &= \{x \leftarrow integer, y \leftarrow integer\}
\end{aligned}
$$

$$q_2 = \{x \leftarrow integer, y \leftarrow integer\}$$
$$q_3 = \{x \leftarrow integer, y \leftarrow integer\}$$

$\widehat{CV}$ is indeed a safe approximation of CV, since for any arc q_i, $\widehat{CV}[q_i]$ describes all the possible states in the set $CV[q_i]$.

Given a program P, its abstract interpretation can be viewed as a system of equations, one for each program point of P. The equations are defined in terms of other equations in the system and interpretations on basic instructions. The result of the abstract interpretation is a fixed point of the equation set. The least fixed point is preferred since that gives the most precise results.

To guarantee that a fixed point exists, the abstract interpretation framework must impose two restrictions on the framework's parameterization. First, the abstract domain must form a complete partial order. Second, the interpretations of the basic instructions must be continuous under the same ordering.

Assuming the these restrictions, the least fixed point of an equation set can be determined by computing the sequence of iterates over the set. The iteration is initialized by setting each equation to the least element of the abstract domain, *i.e.*, $\perp$. On each step of the iteration, the right-hand side of each equation is evaluated. A reference to another equation is resolved using the value of the equation from the previous iteration. The iteration terminates when the value of every equation in the set remains unchanged from one iteration to the next.

The solution is computable if the sequence of iterates is finite. Furthermore, the sequence is guaranteed to be finite if the abstract domain has finite height. Alternatively, the domain may be of infinite height, but the interpretation of basic instructions may be defined in such a way as to guarantee a finite sequence. We will return to this point in Section 2.3.3.

The Cousots's work laid the theoretical foundations of abstract interpretation. They pioneered the approach and realized the potential of a unified framework for program analysis. Research since then has focused on applying abstract interpretation to more expressive languages, improving implementation techniques, and increasing accuracy.

2.3.2 Expressiveness

Abstract interpretation has been applied to a number of different language paradigms, including logic, functional, and mostly-functional languages. In this section we review developments in the abstract interpretation of functional and mostly-functional languages, *e.g.*, Scheme. We do not review developments in the logic programming language community, since most of the results coincide with work on first-order functional languages.

Functional languages

Soon after the Cousots introduced the idea of abstract interpretation, Mycroft [43] applied the technique to perform strictness analysis for a first-order, call-by-need functional language. A strictness analysis determines which arguments of a procedure are needed in the evaluation of the procedure's body. In his characterization, program points corresponded to procedure entry points. For each procedure definition in the program, the abstract interpretation computed the arguments that would definitely be needed in the procedure at run time.

In contrast to the operational framework used by the Cousots, Mycroft's work is cast in a denotational setting. In the denotational framework, the meaning of a program is determined by a function mapping syntax to values. Mycroft's abstract semantics instead maps syntax to values on a flat two-point domain (strict and nonstrict). Since each procedure corresponds to an equation, the abstract semantics produces a system of equations from the program which can then be solved.

At about the same time, Nielson [44] introduced a general abstract interpretation framework based on denotational semantics. The goal was to be able to flow analyze programs in any language that could be given a denotational semantics.

Mycroft and Nielson seem to have established a trend that continued for a decade: abstract interpreters were put in denotational, as opposed to operational, frameworks. The reason for this is not clear, except perhaps that during the same time period standard semantics for functional languages were often given denotationally, and it therefore is sensible to express abstract interpreters for these languages denotationally in order to facilitate correctness proofs.

While early applications were for first-order functional language, Burn, Hankin, and Abramsky [12] extended abstract interpretation to accommodate higher-order functions.

Their work was based on Mycroft's. Higher-order functions in the object language were modeled as functions in the semantic domain. That is, an *intensional* representation of functions was used as opposed to an *extensional* representation, *e.g.*, closures. This choice had enormous consequences on implementations based on this approach, as we will see in the next section.

Wadler [54] described how abstract interpretation over flat domains could be extended to data structures. The idea is to enlarge the domain to accommodate data structures but abstract over them cleverly so that the infinite set of possible data structures is represented using a few domain elements. Wadler's work fits in cleanly with Burn, Hankin, and Abramsky's work on handling higher-order procedures.

Scheme

In the late 1980's, abstract interpretation was applied to Scheme by Harrison [32] and Shivers [49, 51] working independently. Both analyses were expressed denotationally, and both were able to handle assignment and first-class continuations. Their approaches, and motivations, were quite different, however.

Shivers used abstract interpretation to define a control flow analysis for Scheme to be used in an optimizing compiler. The goal was to track the flow of closures through the program and determine what closures arrived at what call sites. Subsequently, a data flow graph could be constructed and compiler optimizations performed.

One way in which Scheme differs from purely functional languages is that it has assignment operators and first-class continuations. Shiver's analysis handled first-class continuations by converting the input program to continuation-passing style (CPS). Assignments were eliminated by performing assignment conversion [24]: assigned variables were bound to heap-allocated cells, and references and assignments were translated accordingly.

Data structures were accommodated by the analysis, but not with any accuracy. In fact, a single location was used to model the constituent locations of all heap-allocated data structures. The effect is that all values placed into data structures would escape, and the analysis would have to assume they could be applied to any value. Similarly, the value retrieved from a data structure could be any value that had been stored in a data structure. Shivers adopted this approach since he wanted a flow analysis for Scheme that tracked only closures.

Harrison's flow analysis was more ambitious in theory. He was concerned with the automatic parallelization of Scheme programs. Side-effects and first-class continuations are a major impediment to achieving that goal, and Harrison hoped to use abstract interpretation as a framework for designing a flow analysis for Scheme. The flow analysis would yield enough information to identify regions of code that could be executed in parallel and estimate object lifetimes in order to improve memory allocation.

Harrison's abstract interpretation was more precise than Shivers's. All values, including closures and continuations, were traced, even if they passed through data structures. Harrison did not convert the input program to CPS but instead used a representation that is similar to three-address code and is sufficient to handle first-class continuations.

2.3.3 Implementation techniques

Computing an abstract interpretation requires finding the fixed point of a set of equations. As already mentioned, the naïve algorithm iteratively computes the iterates by brute force. This algorithm is expensive and not practical, however. Given an abstract domain of n elements and an equation dependent on m variables, there are n^m possible values for the equation that must be explored.

Fortunately, the naïve algorithm is not the only option for computing fixed points. In this section we review a number of implementation techniques devised to find a fixed point more rapidly. The first, the *frontiers algorithm*, is designed to quickly find a global fixed point. The second, *chaotic iteration*, is designed to compute a local fixed point instead of a global fixed point. The final two techniques, *widening* and limiting *polyvariance*, are designed to accelerate convergence to a fixed point, not necessarily the least, when the abstract domain is large or infinite.

The frontiers algorithm

Clack and Peyton Jones [16] introduced the frontiers algorithm for optimizing the abstract interpretation of first-order functional languages. They assumed a two-point abstract domain $\{\bot, \top\}$, since their intended application was strictness analysis. Their goal was to find a compact representation for each equation that was not exponential in the number of arguments to the equation.

Their approach can be explained as follows. Since the abstract domain is a partial order, the possible argument vectors of an equation form a partial order as well. Also, since each equation is monotonic, once the equation returns $\top$ for one argument vector, the equation returns $\top$ for greater argument vectors as well. By exploiting this observation, a "frontier set" can define the cutoff between $\bot$ and $\top$ among the possible argument vectors to the equation. Because the value of the equation on all possible argument vectors can be determined from the frontier set, the frontier set represents the function and is a compact representation at that.

Using frontiers allows the fixed point algorithm to be optimized. Since the frontier determines the function, the fixed point algorithm must find just the frontier, which means that not every possible argument vector to the equation needs to be explored. Clack and Peyton Jones did not outline any specific heuristics, but the potential for optimization was clear.

The restriction that the abstract domain be two points is rather severe. Fortunately, Martin and Hankin [42] solved this problem and also generalized the frontiers algorithm to higher-order languages. Abstract domains of more than two points are handled by representing each function on the larger domain with a family of functions on the two point domain.

Chaotic iteration

The frontiers algorithm assumes that every equation in the equation set is called with all values on the abstract domain. It computes a global fixed point of the equation set. This is overly general in practice. In fact, the equations need to be applied only on some points, depending on how they are used by other equations in the equation set. In other words, only a local fixed point on the points of interest is needed. This observation is the heart of an optimization technique known as *chaotic iteration* [21]. This technique has been used in practice to get efficient abstract interpretations for Prolog [14, 45] and functional languages [3, 38].

The general idea of chaotic iteration is that given a system of continuous, monotone equations, each equation can be applied in any order provided that every equation is eventually applied an infinite number of times. The algorithm to do this is called a *chaotic iteration algorithm*, and a particular choice for the order in which the equations are applied

is called a *chaotic iteration strategy* [10]. To find a local fixed point, the initial values for one equation are provided, and this drives the chaotic iteration strategy to apply other equations until the local fixed point is computed.

Using chaotic iteration requires that it be possible to compare the arguments to which an equation is applied and the results that it returns. This precludes modeling procedures in the object language as functions in the semantic domain, since it is not possible to compare functions. Therefore, procedures are modeled as closures. Rosendahl [46] described how to do this in the framework of chaotic iteration, but the basic idea is older, going back to work on minimal function graphs [39, 40] and Young's pending analysis [59].

The use of chaotic iteration opens the door to a number of possible iteration strategies. Many strategies are based on dependency analysis to determine which equations depend on which others. The strategies fall into two broad categories. The first includes *static* dependency algorithms, which use static properties of the equation set to determine the dependency relationships [45]. These algorithms work well for first-order languages. The other category consists of *dynamic* dependency algorithms, which collect information on-the-fly to determine the dependent equations [3]. The most sophisticated algorithms combine techniques from both categories [14, 15, 53].

Widening

Cousot and Cousot's original formulation of abstract interpretation permitted the use of abstract domains with infinite height. The problem is that algorithms to find fixed points do not terminate when the domain has infinite height. To address this problem, Cousot and Cousot introduced the notion of widening operators [20]. Given a system of equations, one or more widening operators can be "sprinkled" among the equations. The operators force the fixed point of the decorated equation set beyond the fixed point of the undecorated equation set. As a side effect, the sequence of iterates computed by the fixed point algorithm is rendered finite.

Widening operators are necessary when the abstract domain is infinite, but they are also useful when the abstract domain is finite but large. A fixed point algorithm may require many iterations to reach the least fixed point, but by using a widening operator, a fixed point can be found in less time. The consequence, of course, is that the best fixed point may not be found and this leads to a loss of accuracy. As a simple example, suppose

the abstract domain consists of types, including the types $R = \{\mathit{fixnum}, \mathit{bignum}, \mathit{float}\}$ to represent various representations for numbers. A widening operator may choose to map any abstract value V which contains a subset of R to the abstract value $V \cup R$. This operator may speed up the analysis, but the result is a loss of accuracy when distinguishing the representations of numbers.

Widening operators have not been used extensively because it is hard to determine good operators and to determine where the operators should be placed among the equations. A good operator is one that does not overshoot the least fixed point by much. Unfortunately, choosing a good operator requires intimate knowledge of the abstract domain. Similarly, knowing where to place an operator is highly dependent on the specific equation set being decorated. For these reasons, incorporating widening operators as a reliable, built-in component of fixed point algorithms is difficult.

Consequently, only a few uses of widening operators have been reported in the literature. Bourdoncle [9, 10] has investigated their use, but his investigation is theoretical, and he does not report an implementation using his techniques. Mauborgne uses widening operators to accelerate a strictness analysis for a small higher-order functional language. Higher-order functions are represented using boolean expressions, and widening operators are used to manage the size of the expressions. Yi and Harrison [58] use *projection operators*, which are a special case of widening operators that project values to the top of the abstract domain. The placement of the projection operators is dictated by the user of the analysis.

Polyvariance

Chaotic iteration is a more efficient means of computing a fixed point because it computes the behavior of the equations on just enough points to determine a local fixed point. Although chaotic iteration computes the smallest number of points needed to determine the local fixed point, it is possible to further decrease the number of points at the expense of accuracy. Suppose a monotonic equation Ψ is defined on two points a and b, and those two points are needed to compute a local fixed point of Ψ via chaotic iteration. Suppose a point c such that $a \sqsubseteq c$, $b \sqsubseteq c$, and $\Psi(c)$ is defined. Since Ψ is monotonic, $\Psi(c)$ represents a safe, although perhaps less precise approximation to the behavior of Ψ on *both* points a and b. Exploiting this observation allows a fixed point algorithm to compute a local fixed point

using a smaller number of points than the standard chaotic iteration.

A fixed point algorithm that computes the behavior of a function on more than one point is called *polyvariant*. When the fixed point algorithm computes the behavior of an equation using just one point to approximate all others, the algorithm is called *monovariant*. Monovariant algorithms are still iterative, because in order to preserve some sense of accuracy, the algorithm searches for the least point in the abstract domain that approximates all the points to which the equation is applied. For example, if a standard chaotic iteration would compute the behavior of Ψ on points a, b, and c, a monovariant algorithm would search for the least point d such that $a \sqsubseteq d$, $b \sqsubseteq d$, and $c \sqsubseteq d$. Monovariant algorithms are generally more efficient than their polyvariant counterparts, however, because the equations need to be reevaluated on fewer points.

Interestingly, the literature considers polyvariance to be an extension of monovariance. The truth is in fact just the opposite. A "maximally" polyvariant analysis is the most natural, since that is what falls out of the equations with no effort. A monovariant analysis is really just a particular restriction on a polyvariant analysis.

Both monovariant and polyvariant analyses appear in practice, but as expected, the monovariant analyses are more efficient. Whether or not the loss in accuracy is acceptable depends on the application. Some examples of monovariant analyses include Shivers's 0CFA analysis [49, 51], and Bondorf and Jorgensen's binding-time analysis [7]. Examples of polyvariant analyses include Consel's polyvariant binding-time analysis [3, 18] and Steensgaard's [52] polyvariant closure analysis. A noteworthy category of polyvariant analyses includes those for polymorphic functional languages, for which the nature of the polyvariance coincides with the polymorphism in the input program. Examples of these analyses include a strictness analyzer for Haskell [38] and Henglein and Mossin's binding-time analysis [35].

2.4 Summary

This chapter has presented an informal introduction to abstract interpretation and has reviewed the history of the field. The introduction was cast in the framework of the Cousots, who laid the theoretical foundations of abstract interpretation and anticipated most of the theoretical developments in the application of abstract interpretation.

Since its introduction, abstract interpretation has been applied predominantly to the analysis of functional and logic programming languages. In the functional community, abstract interpretation was recast in a denotational framework and applied first to the analysis of first-order languages. It was subsequently extended to languages with higher-order procedures and data structures. In the logic programming community, it has been applied both to first-order and higher-order logic programming languages.

A primary concern is the efficiency of implemented abstract interpreters. The simple algorithm for computing an abstract interpretation is extremely expensive and is not practical. The problem is that each equation is evaluated on every point in the abstract domain. A more efficient algorithm must therefore find a way to reduce the number of points on which each equation must be examined.

Three techniques are common. One approach to obtaining a more efficient interpretation is the frontiers algorithm. The algorithm reduces the number of points on which the equation needs to be analyzed. Another approach is to use chaotic iteration. Chaotic iteration is used to investigate the behavior of equations on just enough points to compute a local fixed point. Controlling the polyvariance of the analysis is yet a third way to obtain better efficiency. By exploiting the monotonicity of the equations, some points in the abstract domain can be used as approximations for other points, reducing still further the number of points on which the equation needs to be examined.

Chapter 3

An operational semantics for Scheme

In this chapter we give an operational semantics for Scheme. The semantics corresponds to the standard semantics in Cousot and Cousot's framework and will be the basis of our flow analysis framework as well.

We begin by defining a core subset of Scheme. The core language retains enough features to address the important semantic aspects of the language, but it is simple enough so that we are not mired in details when giving it a semantics.

Despite the simplicity of the core language, an operational semantics for the language is relatively complex. To simplify the semantics, core Scheme is translated in two steps to a form whose semantics is simpler and thus better suited as the basis of our flow analysis. In the first step core Scheme is translated into *A-normal form* [27]. A-normal form is a representation that normalizes continuations and names intermediate expressions not in tail position. An operational semantics is given for this language. Converting to A-normal form simplifies the semantics, but it is still more complicated than necessary. Thus in a second step A-normal form expressions are converted to continuation-passing style (CPS) [2]. An operational semantics for CPS A-normal expressions style is defined, and this semantics is the standard semantics.

$$
\begin{array}{rcl}
M & = & c \mid v \mid (\textbf{if}\ (\textbf{pair?}\ M_1)\ M_2\ M_3) \mid (\textbf{call/cc}\ M) \mid (\textbf{lambda}\ (v_1\ \ldots\ v_n)\ M) \mid \\
 & & (M_0\ M_1\ \ldots\ M_n) \mid (\textbf{cons}\ M_1\ M_2) \mid (\textbf{car}\ M) \mid (\textbf{begin}\ (\textbf{set-car!}\ M_1\ M_2)\ M_3) \\
c & \in & \textit{Constants} \\
v & \in & \textit{Variables}
\end{array}
$$

Figure 3.1: The language *CS* of core Scheme.

3.1 Core Scheme

The language *CS* of core Scheme is given in Figure 3.1. One predicate is included, and its use is restricted to test contexts. Assignment is discarded, since assignment can be eliminated through assignment conversion [24]. Since the use of **set-car!** has an unspecified value, its use is restricted to a for-effect only context.

A semantics that can give meaning to core Scheme can be easily extended to give a semantics for full Scheme or other higher-order languages. A semantics that supports **call/cc** must make a continuation available for reification. Thus the semantics can be extended to accommodate other control features such as multiple return values [4]. The choice of primitives admits but abstracts the creation of aggregate data structures and mutation. A semantics that can give meaning to these features must therefore maintain an explicit store. The language is a simple enough language with which to reason but rich enough to demand a complete semantics.

The characterization of execution state in the semantics impacts the elegance and capabilities of our flow analysis framework. Recall Cousot and Cousot's standard operational semantics. The program state changes as execution proceeds from node to node in the program flowchart, and each arc in the flowchart is a program point. The collecting semantics records the set of states that arise at each program point during execution, and the abstract semantics computes an approximation of the set of states at each program point.

Our standard semantics also follows this approach. Unfortunately, any semantics we could define for core Scheme would be complicated. Continuations are the source of the complication. In an operational semantics, the continuation is often represented as a sequence of "frames." Each frame has some code and perhaps some other values with which to continue the computation. For example, a continuation frame for an application holds the arguments to the procedure call, and when the continuation is invoked with a procedure value, the semantics uses the saved arguments to perform the procedure call. Landin's

$$
\begin{array}{lcl}
A & = & (\textbf{let}\ ((v\ M))\ A) \mid (\textbf{set-car!}\ S_1\ S_2\ A) \mid M \mid S \\
M & = & (\textbf{cons}\ v_1\ v_2\ S_1\ S_2) \mid (\textbf{car}\ S) \mid (\textbf{pair?}\ S\ A_1\ A_2) \mid (\textbf{call/cc}\ S) \mid (S_0\ S_1\ \ldots\ S_n) \\
S & = & c \mid v \mid (\textbf{lambda}\ (v_1\ \ldots\ v_n)\ A)
\end{array}
$$

Figure 3.2: The language $A(CS)$ of A-normal form.

SECD machine [41] is an example of an operational semantics that uses this strategy for representing control.

Continuations represented in this way are complicated for two reasons. First, there are several different types of continuation frames, which complicates the domain of continuations. Second, fetching values is complicated, since the semantics must be able to reference values not only through an environment but through a continuation as well.

In the rest of this chapter, we focus on obtaining an operational semantics that removes these complications. Once the complications are eliminated, the resulting operational semantics is simple and ideal for use as the standard semantics.

3.2 Normalizing programs

An alternative to a complicated operational semantics is to normalize terms to some simpler language and give a semantics to the simpler language instead. Presumably the semantics for the simpler language is easier to specify, understand, and manipulate. This is the strategy we apply to our problem. Core Scheme is normalized by translating expressions to A-normal form.

The language $A(CS)$ of A-normal form expressions is given in Figure 3.2. Expressions in A-normal form have two important properties: complex intermediate expressions are named, and the code is linearized. Furthermore, redundant keywords are eliminated from the syntax of assignment and conditional expressions, and two fresh, template variables are added to **cons** expressions. These changes simplify the operational semantics that will be given to this language. While the template variables are not strictly necessary for the standard semantics, they will be required for the abstract semantics to ensure that it is computable. An example showing the A-normal form equivalent of a small Scheme program is given in Figure 3.3.

Expressions in core Scheme can be rewritten into A-normal form in time and space proportional to the size of the input expression. A Scheme program to do the translation is

```
((lambda (copy) (copy (cons 1 (cons 2 '())) copy))
 (lambda (ls copy)
   (if (pair? ls)
       (cons (car ls) (copy (cdr ls) copy))
       '())))

((lambda (copy)
   (let ((v0 (cons v1 v2 2 '())))
     (let ((v3 (cons v4 v5 1 v0)))
       (copy v3 copy))))
 (lambda (ls copy)
   (pair? ls
          (let ((v6 (car ls)))
            (let ((v7 (cdr ls)))
              (let ((v8 (copy v7 copy)))
                (cons v9 v10 v6 v8))))
          '())))
```

Figure 3.3: The procedure *copy* and its equivalent in A-normal form.

given in Figures 3.4 and 3.5. It is a straightforward adaptation of the algorithm to given by Flanagan *et. al.* [27]. The program assumes that the input expression has been parsed into variant records, and **variant-case** [28] is used to destructure the records. *normalize-term* is called to initiate the translation.

By linearizing the code and naming complex intermediate expressions, the structure of continuations is simplified considerably. By naming intermediate expressions, the continuation can reference them by name through the environment instead of referencing them through elements of a continuation structure. By linearizing the code, the continuation does not need to maintain a tag denoting the evaluation context. The code to execute next is given directly as another A-normal form expression. Thus, every continuation must record the same three pieces of information:

1. to what variable will the received value be bound,
2. what A-normal form expression is to be executed next, and
3. in what environment will the variable be bound and the expression be executed.

```
(define normalize-term
  (lambda (x)
    (normalize x (lambda (x) x))))

(define normalize
  (lambda (x k)
    (variant-case x
      (lit () (k x))
      (varref () (k x))
      (proc (formals body)
        (k (make-proc formals (normalize-term body))))
      (callcc (rand)
        (normalize-name rand (lambda (rand) (k (make-callcc rand)))))
      (if (test then else)
        (normalize-name test
          (lambda (exp)
            (k (make-if exp (normalize-term then) (normalize-term else))))))
      (app (rator rands)
        (normalize-name* (cons rator rands)
          (lambda (ls)
            (k (make-app (car ls) (cdr ls))))))
      (car (exp)
        (normalize-name exp (lambda (exp) (k (make-car exp)))))
      (setcar (exp1 exp2 rest)
        (normalize-name exp1
          (lambda (exp1)
            (normalize-name exp2
              (lambda (exp)
                (make-setcar exp1 exp2 (normalize-term rest tail? k)))))))
      (cons (exp1 exp2)
        (normalize-name exp1
          (lambda (exp1)
            (normalize-name exp2
              (lambda (exp2)
                (k (make-cons (new-var) (new-var) exp1 exp2))))))))))
```

Figure 3.4: An algorithm to translate from *CS* to *A*(*CS*).

```
(define normalize-name
  (lambda (x k)
    (normalize x
      (lambda (x)
        (if (or (lit? x) (varref? x) (proc? x))
            (k x)
            (let ((var (new-var)))
              (make-bind var x (k (make-varref var))))))))))

(define normalize-name*
  (lambda (ls k)
    (if (null? ls)
        (k '())
        (normalize-name (car ls)
          (lambda (first)
            (normalize-name* (cdr rands) (lambda (rest) (k (cons first rest)))))))))
```

Figure 3.5: Auxiliary procedures for A-normalization.

3.3 Abstract machines

One form of an operational semantics is an abstract state machine. When loaded with a program, the machine goes through some number of transitions and terminates with the result of the program's evaluation. Many possible definitions for abstract machines are possible. The following definition, modified from the one given by Guttman *et. al.* [30], is convenient.

Definition 3.3.1 (State Machine) A *state machine* is a tuple $\langle states, inits, halts, acts\rangle$ such that

1. $inits \subseteq states$, and $halts \subseteq states$;
2. $acts \subseteq states \times states$; and
3. $s \in halts$ implies that for all $s' \in states$, $\langle s, s'\rangle \notin acts$.

Table 3.1 summarizes the mathematical notation used in this definition and elsewhere in the dissertation.

A state machine is a set of states with some of them being initial states and halt states. A state transition relation *acts* maps states to states. Halt states have the property that there is no transition out of a halt state.

$\emptyset$	the empty set
$A \times B$	the cross product of sets A and B
A^*	the set of sequences of length 0 and greater made from A
$\mathcal{P}(A)$	the powerset of A
$\langle \ldots \rangle$	sequence (tuple) formation
$x \frown y$	the concatenation of sequences x and y
$\vec{v}$	a sequence of unspecified length
$A \rightarrow B$	the set of functions from A to B
$A \stackrel{\text{fin}}{\rightarrow} B$	the set of finite functions from A to B
$dom(f)$	the domain of a finite function f
$[x \leftarrow v]f$	the functional extension of the finite function f
$[x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]f$	shorthand for $[x_n \leftarrow v_n] \ldots [x_1 \leftarrow v_1]f$
$- \rightarrow -, -$	McCarthy conditional

Table 3.1: Notational conventions.

A number of terms are used when describing the behavior of an abstract machine. If $\langle s, s' \rangle \in acts$, we will say that s *can proceed to* s', or $s \rightarrow s'$. A state s is called a *halt state* if $s \in halts$. A state s is called an *initial state* if $s \in inits$. If T is a finite or infinite sequence of states, then it is a *computation* (or *trace*) if and only if $T(0) \in inits$ and for every i such that $T(i+1)$ is well-defined, $\langle T(i), T(i+1) \rangle \in acts$. A state is *accessible* if $T(i) = s$ for some computation T. A computation is maximal if it is infinite or if its last state is not in the domain of $acts$. A finite, maximal computation is *successful* if its last state is in $halts$. Any other finite, maximal computation is *erroneous*. When T is a successful computation that terminates in a halt state s, we will write $ans(T)$ to mean s. Otherwise $ans(T)$ is undefined. We will say that M can compute s_1 from s_0 if there exists a computation T such that $T(0) = s_0$ and $ans(T) = s_1$.

3.4 A CESK machine for A-Normal form

We are now able to define an operational semantics for A-normal form. The machine takes the form of a CESK (code, environment, store, continuation) machine. The machine is defined as follows.

Definition 3.4.1 (Intermediate Machine) Let $A(CS)$ be the language of A-normal form expressions and *Environments*, *Stores*, and *Continuations* the domains defined in Figure 3.6. Then M^I is the state machine

$$\langle (A(CS) \times Environments \times Stores \times Continuations), inits^I, halts^I, acts^I \rangle$$

where

1. inits^I is the set $\{\langle A, \emptyset, \emptyset, \langle\rangle\rangle \mid A \in A(CS)\}$,
2. halts^I is the set such that $\langle v, \rho, \sigma, \langle\rangle\rangle \in \mathit{halts}^I$ if and only if $v \in \mathit{dom}(\rho)$ and $\rho(v) \in \mathit{dom}(\sigma)$, and
3. acts^I is the transition relation $\longrightarrow$ described in Figure 3.6.

The behavior of the abstract machine is captured by the transition rules. They are not unusual. Expressions are evaluated in tail position with respect to some continuation κ. The evaluation of a simple expression simply sends the value of the expression to the continuation. For a conditional, one arm or the other is selected based on the value of the test, and the details of the selection are abstracted with the function *truish*?, whose definition is omitted. There are two cases for application, depending on whether a procedure or continuation is being applied. The function *new* takes a sequence of variables and a store. The sequence is used as a template to generate new locations that do not occur in the domain of the store. For a **let** expression, the right-hand side is evaluated in tail position with respect to a new continuation that will accept the value of the right-hand side, bind it to the variable being introduced, and continue execution with the body.

3.5 Conversion to continuation-passing style

The complications outlined in Section 3.1 are mostly solved by converting to A-normal form. The domain of continuations is simple since every frame has the same form. Furthermore, all references go through the environment; continuation frames do not contain any values referenced directly by code.

While this is an improvement, the CESK machine is still complicated because the continuation is manipulated as an extra piece of state. The continuation is a distinct datatype with its own operations, and the continuation is located in a “register” separate from all other values. The extra register holding the continuation can be eliminated by manipulating the continuation at the source level. The continuation can be brought to the source level by converting expressions to *continuation passing style*. The conversion process is relatively simple, since intermediate values are already named and expressions have a linear shape to them.

$$\begin{array}{rcl}
\rho \in \mathit{Environments} & = & \mathit{Variables} \xrightarrow{\text{fin}} \mathit{Locations} \\
\sigma \in \mathit{Stores} & = & \mathit{Locations} \xrightarrow{\text{fin}} \mathit{Objects} \\
\epsilon \in \mathit{Objects} & = & \mathit{Constants} \cup (\mathit{Locations} \times \mathit{Locations}) \cup \mathit{Continuations} \cup \\
& & (\mathit{Variables}^* \times A(\mathit{CS}) \times \mathit{Environments}) \\
\kappa \in \mathit{Continuations} & = & (\mathit{Variables} \times A(\mathit{CS}) \times \mathit{Environments})^* \\
l & \in & \mathit{Locations}
\end{array}$$

$$\langle S, \rho, \sigma, \kappa \rangle \longrightarrow \mathit{apply}(\kappa, [\![S]\!]\rho\sigma, \sigma)$$

$$\langle (\textbf{call/cc}\ S), \rho, \sigma, \kappa \rangle \longrightarrow \langle A, [v \leftarrow l]\rho', [l \leftarrow \kappa]\sigma, \kappa \rangle$$

$$\begin{array}{rcl}
\mathit{where}\ \langle \langle v \rangle, A, \rho' \rangle & = & [\![S]\!]\rho\sigma \\
\langle l \rangle & = & \mathit{new}(\langle v \rangle, \sigma)
\end{array}$$

$$\langle (\textbf{cons}\ v_1\ v_2\ S_1\ S_2), \rho, \sigma, \kappa \rangle \longrightarrow \mathit{apply}(\kappa, \langle l_1, l_2 \rangle, [l_1, l_2 \leftarrow [\![S_1]\!]\rho\sigma, [\![S_2]\!]\rho\sigma]\sigma)$$

$$\mathit{where}\ \langle l_1, l_2 \rangle = \mathit{new}(\langle v_1, v_2 \rangle, \sigma)$$

$$\langle (\textbf{car}\ S), \rho, \sigma, \kappa \rangle \longrightarrow \mathit{apply}(\kappa, \sigma(l_0), \sigma)$$

$$\mathit{where}\ \langle l_0, l_1 \rangle = [\![S]\!]\rho\sigma$$

$$\langle (\textbf{set-car!}\ S_1\ S_2\ A), \rho, \sigma, \kappa \rangle \longrightarrow \langle A, \rho, [l_0 \leftarrow [\![S_2]\!]\rho\sigma]\sigma, \kappa \rangle$$

$$\mathit{where}\ \langle l_0, l_1 \rangle = \sigma([\![S_1]\!]\rho\sigma)$$

$$\langle (\textbf{pair?}\ S\ A_1\ A_2), \rho, \sigma, \kappa \rangle \longrightarrow \langle (\mathit{truish?}([\![S]\!]\rho\sigma) \rightarrow A_1, A_2), \rho, \sigma, \kappa \rangle$$

$$\langle (S_0\ S_1), \rho, \sigma, \kappa \rangle \longrightarrow \mathit{apply}([\![S_0]\!]\rho\sigma, [\![S_1]\!]\rho\sigma, \sigma)$$

$$\mathit{where}\ \langle v, A, \rho \rangle : \kappa = [\![S_0]\!]\rho\sigma$$

$$\langle (S_0\ S_1\ \ldots\ S_n), \rho, \sigma, \kappa \rangle \longrightarrow \langle A, \rho'', \sigma', \kappa \rangle$$

$$\begin{array}{rcl}
\mathit{where}\ \langle \langle v_1, \ldots, v_n \rangle, A, \rho' \rangle & = & [\![S_0]\!]\rho\sigma \\
\langle l_1, \ldots, l_n \rangle & = & \mathit{new}(\langle v_1, \ldots, v_n \rangle, \sigma) \\
\rho'' & = & [v_1, \ldots, v_n \leftarrow l_1, \ldots, l_n]\rho' \\
\sigma' & = & [l_1, \ldots, l_n \leftarrow [\![S_1]\!]\rho\sigma, \ldots, [\![S_n]\!]\rho\sigma]\sigma
\end{array}$$

$$\langle (\textbf{let}\ ((v\ M))\ A), \rho, \sigma, \kappa \rangle \longrightarrow \langle M, \rho, \sigma, \langle v, A, \rho \rangle : \kappa \rangle$$

$$[\![c]\!]\rho\sigma = c$$

$$[\![v]\!]\rho\sigma = \sigma(\rho(v))$$

$$[\![(\textbf{lambda}\ (v_1\ \ldots\ v_n)\ A)]\!]\rho\sigma = \langle \langle v_1, \ldots, v_n \rangle, A, \rho \rangle$$

$$\mathit{apply}(\langle v, A, \rho \rangle : \kappa, \epsilon, \sigma) = \langle A, [v \leftarrow l]\rho, [l \leftarrow \epsilon]\sigma, \kappa \rangle$$

$$\mathit{where}\ \langle l \rangle = \mathit{new}(\langle v \rangle, \sigma)$$

Figure 3.6: Domains and state transition relation for the intermediate machine.

$$
\begin{array}{lcl}
A & = & (\textbf{let}\ ((v\ (\textbf{cons}\ v_1\ v_2\ S_1\ S_2)))\ A) \mid (\textbf{let}\ ((v\ (\textbf{car}\ S)))\ A) \mid \\
 & & (S_0\ S_1\ \ldots\ S_n) \mid (\textbf{pair?}\ v\ S_1\ S_2\ A_1\ A_2) \mid (\textbf{set-car!}\ S_1\ S_2\ A) \mid (\textbf{halt}\ v) \\
S & = & c \mid v \mid (\textbf{lambda}\ \eta\ (v_1\ \ldots\ v_n)\ A) \\
\eta & \in & \mathit{Tags}
\end{array}
$$

Figure 3.7: The language $CPS(A(CS))$ of CPS A-normal form.

The language $CPS(A(CS))$ of CPS A-normal form programs is given in Figure 3.7. Continuations are represented as procedures, so continuation creation, invocation, and reification can all be expressed as operations on procedures. **call/cc** is eliminated since it can be expressed using **lambda** and application. To avoid code duplication, the continuation of a conditional is bound outside the scope of the conditional. Also, a new form (**halt** v) is added for the initial continuation.

Lambda expressions are also tagged in the conversion process. In Cousot and Cousot's framework, the arcs of a program's flowchart constituted the set of program points, and each arc was labeled. Lambda expressions are labeled here for the same reason, as each lambda expression corresponds to a program point. From the perspective of conventional flow analysis, the body of each lambda expression is a basic block, and a lambda expression's tag labels the basic block that the body represents. The standard semantics defined in this chapter has no use for the tags, but they will be necessary when defining the flow analysis framework.

A translator from A-normal form to CPS A-normal form is given in Figures 3.8 and 3.9. The function *cps* takes a program and converts it to continuation-passing style. It provides an initial continuation and compiles the expression in the context of the variable to which the continuation is bound. The translator operates in one pass over the input expression. It is also assumed to be hygienic [5] in the sense that variables and tags that are introduced do not clash with preexisting variables and tags.

The procedure *cps* compiles A-normal form expressions to CPS A-normal form. **call/cc** is eliminated and replaced with an application. The application references the reified continuation, which is a procedure that accepts a continuation and argument. When the continuation is applied, it discards the received continuation and applies the saved continuation to the received argument. Nontail expressions that must build a continuation use a **let** binding to introduce the continuation. The nontail expression, now in tail position, is

```
(define cps
  (lambda (x)
    (cps-term x
      (let ((formal (new-var)))
        (make-proc (new-tag) (list formal) (make-halt formal))))))

(define cps-term
  (lambda (x vk)
    (variant-case x
      (lit () (make-app vk x))
      (varref () (make-app vk x))
      (proc (formals body)
        (make-app vk
          (let ((vk (new-var)))
            (make-cpsproc (new-tag) (cons vk formals)
              (cps-term body (make-varref vk))))))
      (bind (var rhs body)
        (handle-bind var rhs (cps-term body vk)))
      (app (rator rands)
        (make-app rator (cons vk rands)))
      (if (test then else)
        (let ((vkp (new-var)))
          (let ((vkp-ref (make-varref vkp)))
            (make-cpsif vkp vk test (cps-term then vkp-ref)
              (cps-term else vkp-ref)))))
      (callcc (rand)
        (make-app rand
          (list vk (let ((vb (new-var)))
                      (make-proc (new-tag) (list (new-var) vb)
                        (make-app vk (list (make-varref vb))))))))
      (cons (var1 var2 exp1 exp2)
        (let ((vb (new-var)))
          (make-bind vb x (make-app vk (list (make-varref vb))))))
      (car (exp)
        (let ((vb (new-var)))
          (make-bind vb x (make-app vk (list (make-varref vb))))))
      (setcar (exp1 exp2 rest)
        (make-setcar exp1 exp2 (cps-term rest vk))))))
```

Figure 3.8: A translator from $A(CS)$ to $CPS(A(CS))$.

```
(define handle-bind
  (lambda (var rhs body)
    (variant-case rhs
      (car () (make-bind var rhs body))
      (cons () (make-bind var rhs body))
      (else (cps-term rhs (make-proc (new-tag) (list var) body))))))
```

Figure 3.9: Auxiliary procedure for the CPS translator.

```
((lambda η0 (k0 copy)
   (let ((v0 (cons v2 v3 2 '())))
     (let ((v1 (cons v4 v5 1 v0)))
       (copy k0 v1 copy))))
 (lambda η1 (v6) (halt v6))
 (lambda η2 (k1 ls copy)
   (pair? k2 k1 ls
     (let ((v7 (cdr ls)))
       (copy (lambda η3 (v8)
                (let ((v9 (car ls)))
                  (let ((v10 (cons v11 v12 v9 v8)))
                    (k2 v10))))
             v7
             copy))
     (k2 '()))))
```

Figure 3.10: The procedure *copy* in CPS A-normal form.

then compiled in the context of the new continuation. The CPS A-normal version of the procedure *copy* is given in Figure 3.10.

The CPS translation represents continuations as procedures. This is typical but not essential. It is possible, and sometimes desirable, to preserve the continuation as a distinct datatype. For example, a compiler that converts to continuation-passing style may represent continuations differently in order to better allocate them at run time. For our purposes, however, unifying continuations and procedures simplifies the semantics without any loss of essential information. Furthermore, after the analysis the direct-style version of the program may be recovered if desired [22].

3.6 A CES machine for CPS A-normal form

By making the continuation explicit at the source level, there is no need for the abstract machine to maintain a continuation. Assuming a function $compile : CS \rightarrow CPS(A(CS))$, we can define a CES machine for CPS A-normal form programs.

Definition 3.6.1 (Standard Machine) Let $CPS(A(CS))$ be the language of CPS A-normal form and *Environments* and *Stores* the domains defined in Figure 3.11. Then M^S is the state machine

$$\langle (CPS(A(CS)) \times Environments \times Stores), inits^E, halts^E, acts^E \rangle$$

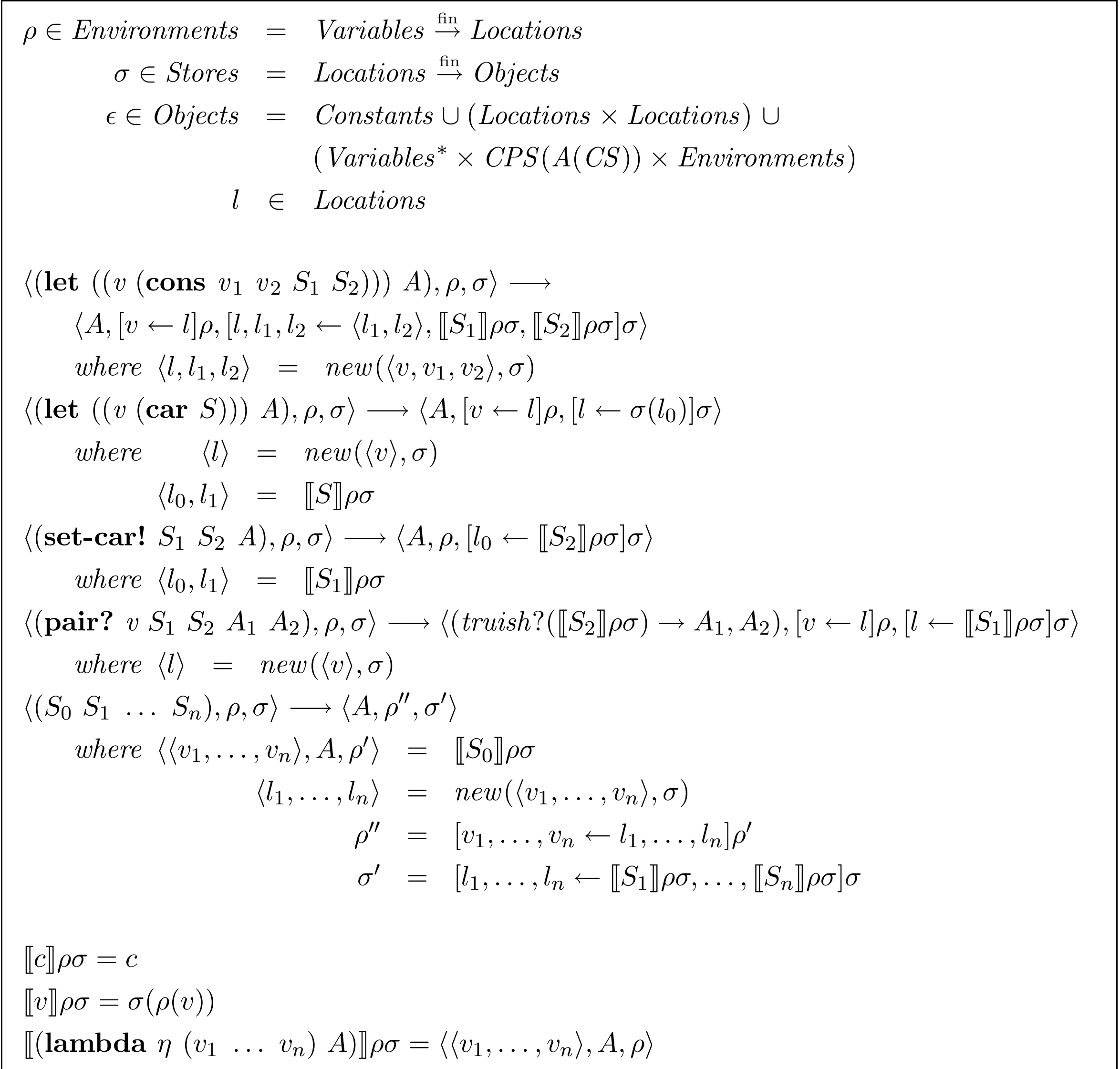
where

1. $inits^E$ is the set $\{\langle A, \emptyset, \emptyset \rangle \mid \exists M \in CS. A = compile(M)\}$;
2. $halts^E$ is the set such that $\langle (\textbf{halt}\ v), \rho, \sigma \rangle \in halts^E$ iff $v \in dom(\rho)$, and for all $v \in dom(\rho)$, $\rho(v) \in dom(\sigma)$; and
3. $acts^E$ is the transition relation described in Figure 3.11.

Because the continuation is explicit, the definitions of initial and halt states are somewhat different from the CESK machine defined in Section 3.4. An initial state is one in which a compiled program is awaiting evaluation, and a halt state is one in which $(\textbf{halt}\ v)$ is the expression to be executed.

The transition rules are similar to the rules for the CESK machine but much simpler because the continuation is eliminated. Since the program is in CPS, there is no notion of procedure return and both the application rule for continuations and the function *apply* are eliminated. The rule for simple expressions is eliminated, and **cons** and **car** can appear only on the right-hand sides of **let** expressions, so the evaluation rules for these primitives perform the action and bind the result to the introduced variable. The evaluation of a conditional not only evaluates the test and selects an arm, but it also evaluates S_1 to obtain the continuation of the conditional and binds that continuation to v.

This standard machine is the standard semantics for our flow analysis framework. It is simpler than the CESK machine and much simpler than any operational semantics we could have given to core Scheme directly. In the standard machine, the domain of values is composed of just constants, pairs, and procedures, and there is one transition rule for



$$
\begin{aligned}
\rho \in \mathit{Environments} &= \mathit{Variables} \xrightarrow{\text{fin}} \mathit{Locations} \\
\sigma \in \mathit{Stores} &= \mathit{Locations} \xrightarrow{\text{fin}} \mathit{Objects} \\
\epsilon \in \mathit{Objects} &= \mathit{Constants} \cup (\mathit{Locations} \times \mathit{Locations}) \cup \\
&\quad (\mathit{Variables}^* \times \mathit{CPS}(A(\mathit{CS})) \times \mathit{Environments}) \\
l &\in \mathit{Locations}
\end{aligned}
$$

$$
\begin{aligned}
&\langle (\mathbf{let}\ ((v\ (\mathbf{cons}\ v_1\ v_2\ S_1\ S_2)))\ A), \rho, \sigma \rangle \longrightarrow \\
&\quad \langle A, [v \leftarrow l]\rho, [l, l_1, l_2 \leftarrow \langle l_1, l_2 \rangle, [\![S_1]\!]\rho\sigma, [\![S_2]\!]\rho\sigma]\sigma \rangle \\
&\quad \mathit{where}\ \langle l, l_1, l_2 \rangle = \mathit{new}(\langle v, v_1, v_2 \rangle, \sigma) \\
&\langle (\mathbf{let}\ ((v\ (\mathbf{car}\ S)))\ A), \rho, \sigma \rangle \longrightarrow \langle A, [v \leftarrow l]\rho, [l \leftarrow \sigma(l_0)]\sigma \rangle \\
&\quad \mathit{where}\ \langle l \rangle = \mathit{new}(\langle v \rangle, \sigma) \\
&\qquad\qquad \langle l_0, l_1 \rangle = [\![S]\!]\rho\sigma \\
&\langle (\mathbf{set\text{-}car!}\ S_1\ S_2\ A), \rho, \sigma \rangle \longrightarrow \langle A, \rho, [l_0 \leftarrow [\![S_2]\!]\rho\sigma]\sigma \rangle \\
&\quad \mathit{where}\ \langle l_0, l_1 \rangle = [\![S_1]\!]\rho\sigma \\
&\langle (\mathbf{pair?}\ v\ S_1\ S_2\ A_1\ A_2), \rho, \sigma \rangle \longrightarrow \langle (\mathit{truish?}([\![S_2]\!]\rho\sigma) \rightarrow A_1, A_2), [v \leftarrow l]\rho, [l \leftarrow [\![S_1]\!]\rho\sigma]\sigma \rangle \\
&\quad \mathit{where}\ \langle l \rangle = \mathit{new}(\langle v \rangle, \sigma) \\
&\langle (S_0\ S_1\ \ldots\ S_n), \rho, \sigma \rangle \longrightarrow \langle A, \rho'', \sigma' \rangle \\
&\quad \mathit{where}\ \langle \langle v_1, \ldots, v_n \rangle, A, \rho' \rangle = [\![S_0]\!]\rho\sigma \\
&\qquad\qquad \langle l_1, \ldots, l_n \rangle = \mathit{new}(\langle v_1, \ldots, v_n \rangle, \sigma) \\
&\qquad\qquad \rho'' = [v_1, \ldots, v_n \leftarrow l_1, \ldots, l_n]\rho' \\
&\qquad\qquad \sigma' = [l_1, \ldots, l_n \leftarrow [\![S_1]\!]\rho\sigma, \ldots, [\![S_n]\!]\rho\sigma]\sigma
\end{aligned}
$$

$$
\begin{aligned}
&[\![c]\!]\rho\sigma = c \\
&[\![v]\!]\rho\sigma = \sigma(\rho(v)) \\
&[\![(\mathbf{lambda}\ \eta\ (v_1\ \ldots\ v_n)\ A)]\!]\rho\sigma = \langle \langle v_1, \ldots, v_n \rangle, A, \rho \rangle
\end{aligned}
$$

Figure 3.11: Domains and transition relation for the standard machine.

each syntactic form in the language. This machine realizes our goal of finding a simple operational semantics for core Scheme.

3.7 Summary

In this chapter a language and standard semantics for that language were defined as the basis of our flow analysis framework. First the language of core Scheme was defined. Core Scheme includes **call/cc** and several representative primitives. An operational semantics for core Scheme is too complex for our purposes, however. To solve this problem core Scheme was translated into A-normal form. A-normal form is a canonical form in which every intermediate expression is named and the order of evaluation is determined. Expressions in A-normal form have a regular continuation structure that simplifies the handling of continuations in an abstract machine for the language. The final step was to further reduce A-normal form expressions to continuation-passing style A-normal form. This step makes the continuation explicit at the source level. An abstract machine for CPS expressions does not need to manage a continuation explicitly, and therefore, CPS A-normal form and its corresponding operational semantics are the basis of our flow analysis framework.

Chapter 4

A flow analysis framework

In this chapter we develop a flow analysis framework for Scheme programs in CPS A-normal form. The initial development proceeds in two steps following Cousot and Cousot's methodology. In the first step, a collecting machine is defined in terms of the standard machine. A collecting machine is an instrumented variant of the standard machine that builds a history of a program's execution. In our case, the collecting machine builds a cache which describes the execution states that arise at each program point as the program is executed. In the second step, the collecting machine is abstracted to obtain an abstract machine. Given a program, the abstract machine approximates the collecting machine by computing a cache that conservatively describes the cache determined by the collecting machine. This abstract machine is the basis of our analysis framework. After the initial framework is developed, it is refined to extend the range of analyses it can express. The refined framework is able to express a wide range of analyses that vary in accuracy and performance.

4.1 A collecting machine for CPS A-normal form

Following Cousot and Cousot's methodology, a collecting machine for $CPS(A(CS))$ should behave similarly to the standard machine. It differs, however, by accumulating a cache that expresses the set of execution states that arise at each procedure entry point during the execution of a program. Since continuations are represented as procedures in $CPS(A(CS))$, the cache also expresses information about execution state on procedure return. One possible collecting machine may be defined as follows.

Definition 4.1.1 (Collecting Machine) Let $\mathit{CPS}(A(\mathit{CS}))$ be the language of CPS A-normal form expressions and *Stores* and *Caches* the domains defined in Figure 4.1. Then M^C is the state machine

$$\langle(\mathit{CPS}(A(\mathit{CS})) \times \mathit{Environments} \times \mathit{Stores} \times \mathit{Caches}), \mathit{inits}^C, \mathit{halts}^C, \mathit{acts}^C\rangle$$

where

1. inits^C is the set $\{\langle A, \emptyset, \emptyset, \emptyset\rangle \mid \exists M \in \mathit{CS}.A = \mathit{compile}(M)\}$;
2. halts^C is the set such that for all $\rho \in \mathit{Environments}$, $\sigma \in \mathit{Stores}$, and $\gamma \in \mathit{Caches}$, $\langle(\mathbf{halt}\ v), \rho, \sigma, \gamma\rangle \in \mathit{halts}^C$ if and only if $v \in \mathit{dom}(\rho)$ and $v \in \mathit{dom}(\sigma)$; and
3. acts^C is the transition relation described in Figure 4.1.

The machine is very similar to the standard machine. The domains differ only to reflect that a lambda expression's tag is added to a closure yielded by the expression's evaluation. In addition, a domain of caches is added. A cache is a set of tag, environment, and state triples, each of which describes an execution state on entry to a procedure. The current cache is added as a fourth element to the collecting machine's state. As for the transition rules, the only significant change is to the application rule. In addition to setting up for the procedure call, the evaluation of an application adds the current machine state to the cache.

As given, the collecting machine in fact implements a flow analysis. After the machine terminates successfully, a post processor can use the cache to determine control- and data-flow graphs for the program. For example, a portion of the cache computed by the collecting machine for the program in Figure 3.10 is given in Figure 4.2. It describes the program points that correspond to the continuations of calls to *copy*. The environments and stores described in the figure are specified using a set of bindings, so the set $\{a \mapsto b, c \mapsto d\}$ would specify the finite function $[a, b \leftarrow c, d]\emptyset$. The first triple shows the continuation being applied to the empty list. The second shows the continuation being applied to the list (2). The last triple is the initial continuation being applied to the final answer (1 2).

Although the collecting machine computes exact flow information, it unfortunately does not terminate for all programs, since nonterminating programs may naturally lead to an infinite number of execution states. From the collecting machine, however, it is possible to derive an abstract machine that can implement a computable but approximate flow analysis.

$$
\begin{aligned}
\rho \in \mathit{Environments} &= \mathit{Variables} \xrightarrow{\text{fin}} \mathit{Locations} \\
\sigma \in \mathit{Stores} &= \mathit{Locations} \xrightarrow{\text{fin}} \mathit{Objects} \\
\mathit{Objects} &= \mathit{Constants} \cup (\mathit{Locations} \times \mathit{Locations}) \cup \\
&\quad (\mathit{Tags} \times \mathit{Variables}^* \times \mathit{CPS}(A(\mathit{CS})) \times \mathit{Environments}) \\
\gamma \subseteq \mathit{Caches} &= \mathit{Tags} \times \mathit{Environments} \times \mathit{Stores}
\end{aligned}
$$

$$\langle(\textbf{let}\ ((v\ (\textbf{cons}\ v_1\ v_2\ S_1\ S_2)))\ A), \rho, \sigma, \gamma\rangle \longrightarrow \langle A, [v \leftarrow l]\rho, \sigma', \gamma\rangle$$
$$
\begin{aligned}
\textit{where}\ \langle l, l_1, l_2\rangle &= \mathit{new}(\langle v, v_1, v_2\rangle, \sigma) \\
\sigma' &= [l, l_1, l_2 \leftarrow \langle l_1, l_2\rangle, [\![S_1]\!]\rho\sigma, [\![S_2]\!]\rho\sigma]\sigma
\end{aligned}
$$

$$\langle(\textbf{let}\ ((v\ (\textbf{car}\ S)))\ A), \rho, \sigma, \gamma\rangle \longrightarrow \langle A, [v \leftarrow l]\rho, [l \leftarrow \sigma(l_0)]\sigma, \gamma\rangle$$
$$
\begin{aligned}
\textit{where}\ \langle l\rangle &= \mathit{new}(\langle v\rangle, \sigma) \\
\langle l_0, l_1\rangle &= [\![S]\!]\rho\sigma
\end{aligned}
$$

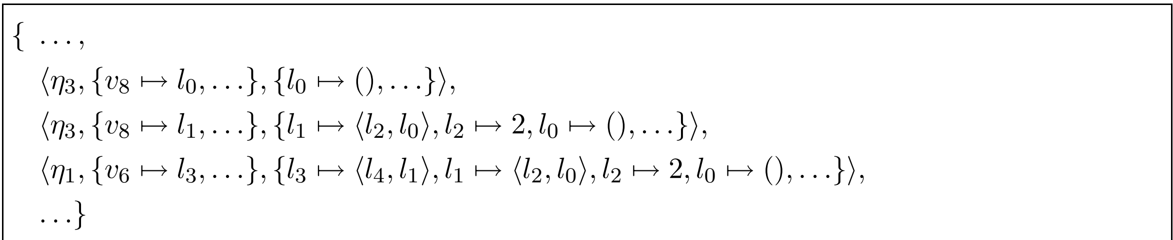
$$\langle(\textbf{set-car!}\ S_1\ S_2\ A), \rho, \sigma, \gamma\rangle \longrightarrow \langle A, \rho, [l_0 \leftarrow [\![S_2]\!]\rho\sigma]\sigma, \gamma\rangle$$
$$\textit{where}\ \langle l_0, l_1\rangle = [\![S_1]\!]\rho\sigma$$

$$\langle(\textbf{pair?}\ v\ S_1\ S_2\ A_1\ A_2), \rho, \sigma, \gamma\rangle \longrightarrow \langle(\mathit{truish?}([\![S_2]\!]\rho\sigma) \rightarrow A_1, A_2), [v \leftarrow l]\rho, \sigma', \gamma\rangle$$
$$
\begin{aligned}
\textit{where}\ \langle l\rangle &= \mathit{new}(\langle v\rangle, \sigma) \\
\sigma' &= [l \leftarrow [\![S_1]\!]\rho\sigma]\sigma
\end{aligned}
$$

$$\langle(S_0\ S_1\ \ldots\ S_n), \rho, \sigma, \gamma\rangle \longrightarrow \langle A, \rho'', \sigma', \gamma \cup \{\langle \eta, \rho'', \sigma'\rangle\}\rangle$$
$$
\begin{aligned}
\textit{where}\ \langle \eta, \langle v_1, \ldots, v_n\rangle, A, \rho'\rangle &= [\![S_0]\!]\rho\sigma \\
\langle l_1, \ldots, l_n\rangle &= \mathit{new}(\langle v_1, \ldots, v_n\rangle, \sigma) \\
\rho'' &= [v_1, \ldots, v_n \leftarrow l_1, \ldots, l_n]\rho' \\
\sigma' &= [l_1, \ldots, l_n \leftarrow [\![S_1]\!]\rho\sigma, \ldots, [\![S_n]\!]\rho\sigma]\sigma
\end{aligned}
$$

$$[\![c]\!]\rho\sigma = c$$
$$[\![v]\!]\rho\sigma = \sigma(\rho(v))$$
$$[\![(\textbf{lambda}\ \eta\ (v_1\ \ldots\ v_n)\ A)]\!]\rho\sigma = \langle \eta, \langle v_1, \ldots, v_n\rangle, A, \rho\rangle$$

Figure 4.1: Domains and transition relation for the collecting machine.



$$
\begin{aligned}
\{\ &\ldots, \\
&\langle \eta_3, \{v_8 \mapsto l_0, \ldots\}, \{l_0 \mapsto (), \ldots\}\rangle, \\
&\langle \eta_3, \{v_8 \mapsto l_1, \ldots\}, \{l_1 \mapsto \langle l_2, l_0\rangle, l_2 \mapsto 2, l_0 \mapsto (), \ldots\}\rangle, \\
&\langle \eta_1, \{v_6 \mapsto l_3, \ldots\}, \{l_3 \mapsto \langle l_4, l_1\rangle, l_1 \mapsto \langle l_2, l_0\rangle, l_2 \mapsto 2, l_0 \mapsto (), \ldots\}\rangle, \\
&\ldots\}
\end{aligned}
$$

Figure 4.2: A portion of the cache computed by the collecting machine for the procedure *copy*.

4.2 An abstract machine for CPS A-normal form

To obtain a computable abstraction of the collecting machine, the collecting machine's cache must be rendered finite. The collecting machine may produce an infinite cache because an infinite number of environment/store pairs are produced by the machine. An infinite number of pairs may be produced if a variable is bound to an infinite number of locations. For example, consider the following program.

```
(letrec ((count
           (lambda (x)
             (if (positive? x)
                 (count (− 0 x))
                 (count (+ (− 0 x) 1))))))
  (count 1))
```

This program does not terminate, and a collecting machine that has been extended with the necessary primitives would gather an infinite set of environment/store pairs at the program point representing entry to the procedure *count*:

$$
\begin{array}{ll}
\langle \rho_0, \sigma_0 \rangle & \text{where } \rho_0(x) = l_0 \text{ and } \sigma_0(l_0) = 1 \\
\langle \rho_1, \sigma_1 \rangle & \text{where } \rho_1(x) = l_1 \text{ and } \sigma_1(l_1) = -1 \\
\vdots & \\
\langle \rho_{2n}, \sigma_{2n} \rangle & \text{where } \rho_{2n}(x) = l_{2n} \text{ and } \sigma_{2n}(l_{2n}) = n \\
\langle \rho_{2n+1}, \sigma_{2n+1} \rangle & \text{where } \rho_{2n+1}(x) = l_{2n+1} \text{ and } \sigma_{2n+1}(l_{2n+1}) = -n \\
\vdots &
\end{array}
$$

This cache is infinite, because an infinte number of locations are allocated for some program variables, *e.g.*, the variable x. A first step at rendering the cache finite is to reduce the set of locations *Locations* to a finite set. Conceptually, this is done by dividing the set into a finite number of partitions and using one location to represent each partition.

Using our previous example, suppose that the locations $l_0, l_2, l_4, \ldots$ compose one partition represented by the location l'_0 and the locations $l_1, l_3, l_5, \ldots$ are represented by the location l'_1. If we also assume a partitioning for other locations in the store, a machine operating under these assumptions would compute a finite set of environment/store pairs at the program point representing procedure entry:

$$
\begin{array}{ll}
\langle \rho_0, \sigma_0 \rangle & \text{where } \rho_0(x) = l'_0 \text{ and } \sigma_0(l'_0) = \{1, 2, 3, \ldots\} \\
\langle \rho_1, \sigma_1 \rangle & \text{where } \rho_1(x) = l'_1 \text{ and } \sigma_1(l'_1) = \{-1, -2, -3, \ldots\}
\end{array}
$$

There are two environment/store pairs because the set of locations to which x is bound has been divided into two partitions. The store maps a location l' to a set of values because l' is approximating all the locations in a partition. This set corresponds to the values that the collecting machine would assign to the locations in the partition during its execution. Therefore in this example l'_0 is mapped to the set of positive integers, and l'_1 is mapped to the set of negative integers.

This cache is still not finite, however, since the set of positive integers and the set of negative integers are both infinte. To make the cache finite, we must find finite approximations to the values computed by the collecting machine. Since there are now only a finite number of locations, aggregates such as pairs and closures which are constructed from locations will be finite, so all that remains is to select finite representations of the base data types, *e.g.*, the integers.

Continuing with our example, we may simply assume that a positive integer is represented using $+$ and a negative integer is represented using $-$. The cache computed then becomes

$$\begin{array}{ll} \langle \rho_0, \sigma_0 \rangle & \text{where } \rho_0(x) = l'_0 \text{ and } \sigma_0(l'_0) = \{+\} \\ \langle \rho_1, \sigma_1 \rangle & \text{where } \rho_1(x) = l'_1 \text{ and } \sigma_1(l'_1) = \{-\} \end{array}$$

This is now a finite approximation of the infinite cache computed by the collecting machine.

It is still necessary to map locations to sets of values in order to represent all the locations in a partition. For example, suppose the partitioning scheme for the example was to group all the locations $l_0, \ldots, l_n, \ldots$ into one partition. In this case the finite cache computed would be

$$\langle \rho_0, \sigma_0 \rangle \quad \text{where } \rho_0(x) = l'_0 \text{ and } \sigma_0(l'_0) = \{+, -\}$$

It is not required that the store maps locations to sets of values, but this choice offers a great deal of flexibility since the goal is to develop a general flow analysis framework that can model many complex data types.

The abstraction methodology developed is reflected in the domains given in Figure 4.3. The domain of abstract locations $\widehat{Locations}$ is finite, and an abstract store maps an abstract location to an abstract value. An abstract value is a set of objects, and each object is either an abstract literal from the finite set $\widehat{Constants}$, a pair, or a closure. The domain of possible stores for a given program forms a finite, complete partial order $(\widehat{Stores}, \sqsubseteq)$. Given

two abstract stores $\hat{\sigma}_0$ and $\hat{\sigma}_1$, the ordering relation $\sqsubseteq$ and least upper bound operator $\sqcup$ on the order can be defined as follows.

$$\begin{array}{rcl} \hat{\sigma}_0 \sqsubseteq \hat{\sigma}_1 & \Leftrightarrow & dom(\hat{\sigma}_0) \subseteq dom(\hat{\sigma}_1) \wedge \forall v \in dom(\hat{\sigma}_0).\hat{\sigma}_0(v) \subseteq \hat{\sigma}_1(v) \\ (\hat{\sigma}_0 \sqcup \hat{\sigma}_1)(v) & = & \begin{cases} \hat{\sigma}_0(v) \cup \hat{\sigma}_1(v) & \text{if } v \in dom(\hat{\sigma}_0) \text{ and } v \in dom(\hat{\sigma}_1) \\ \hat{\sigma}_0(v) & \text{if } v \in dom(\hat{\sigma}_0) \text{ and } v \notin dom(\hat{\sigma}_1) \\ \hat{\sigma}_1(v) & \text{if } v \in dom(\hat{\sigma}_1) \text{ and } v \notin dom(\hat{\sigma}_0) \\ \mathit{undefined} & \text{otherwise} \end{cases} \end{array}$$

The abstract machine may be defined as follows.

Definition 4.2.1 (Abstract Machine) Let $CPS(A(CS))$ be the language of CPS A-normal form expressions and $\widehat{Environments}$, $\widehat{Stores}$, $\widehat{Caches}$, and $Pending$ the domains defined in Figure 4.3. Then M^A is the state machine

$$\langle (CPS(A(CS)) \times \widehat{Environments} \times \widehat{Stores} \times \widehat{Caches} \times Pending), inits^A, halts^A, acts^A \rangle$$

where

1. $inits^A$ is the set $\{\langle A, \emptyset, \emptyset, \emptyset, \langle\rangle \rangle \mid \exists M \in CS.A = compile(M)\}$;
2. $halts^A$ is the set such that for all $A \in CPS(A(CS))$, $\hat{\rho} \in \widehat{Environments}$, $\hat{\sigma} \in \widehat{Stores}$, and $\hat{\gamma} \in \widehat{Caches}$, $\langle A, \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \langle\rangle \rangle \in halts^A$ if and only if $\hat{\gamma} \neq \emptyset$; and
3. $acts^A$ is the transition relation described in Figures 4.3 and 4.4.

Given a compiled source program A, the initial state of the machine is $\langle A, \emptyset, \emptyset, \emptyset \rangle$. The machine halts when the function *pop* is applied to an empty pending set Σ. As the machine executes, it builds a progressively more approximate cache until it is a safe approximation of the cache computed by the collecting machine.

The execution rules of the abstract machine are similar to those of the collecting machine, but the abstract machine must take into account that a value is now a set of objects instead of a single object. Simple expressions are evaluated by injecting them into a singleton set using the abstraction function α to map constants in the domain *Constants* to the domain $\widehat{Constants}$. By customizing α the analysis can be instantiated to abstract over different properties of the basic datatypes. For example, if the analysis were being instantiated for soft typing, α may map all of the integers in a program to *integer*. A **cons** expression

$$\hat{\rho} \in \widehat{Environments} = Variables \xrightarrow{\text{fin}} \widehat{Locations}$$
$$\hat{\sigma} \in \widehat{Stores} = \widehat{Locations} \xrightarrow{\text{fin}} \mathcal{P}(\widehat{Objects})$$
$$\hat{\epsilon} \subseteq \widehat{Objects} = Constants \cup (\widehat{Locations} \times \widehat{Locations}) \cup Closures$$
$$c \in Closures = (Tags \times Variables^{*} \times CPS(A(CS)) \times \widehat{Environments})$$
$$\hat{\gamma} \in Caches = (Tags \times \widehat{Environments}) \xrightarrow{\text{fin}} \widehat{Stores}$$
$$\Sigma \subseteq Pending = CPS(A(CS)) \times \widehat{Environments} \times \widehat{Stores}$$

$$\langle(\textbf{let}\ ((v\ (\textbf{cons}\ v_1\ v_2\ S_1\ S_2)))\ A), \hat{\rho}, \hat{\sigma}', \hat{\gamma}, \Sigma\rangle \longrightarrow \langle A, [v \leftarrow l]\hat{\rho}, \hat{\sigma}', \hat{\gamma}, \Sigma\rangle$$
$$\textit{where}\ \langle l, l_1, l_2\rangle = \widehat{new}(\langle v, v_1, v_2\rangle, \hat{\sigma})$$
$$\hat{\sigma}' = [l, l_1, l_2 \leftarrow \{\langle l_1, l_2\rangle\}, [\![S_1]\!]\hat{\rho}\hat{\sigma}, [\![S_2]\!]\hat{\rho}\hat{\sigma}]\hat{\sigma}$$
$$\langle(\textbf{let}\ ((v\ (\textbf{car}\ S)))\ A), \hat{\rho}, \hat{\sigma}, \hat{\gamma}\rangle \longrightarrow \langle A, [v \leftarrow l]\hat{\rho}, [l \leftarrow \hat{\sigma}(l_0) \cup \ldots \cup \hat{\sigma}(l_n)]\hat{\sigma}, \hat{\gamma}, \Sigma\rangle$$
$$\textit{where}\ \langle l\rangle = \widehat{new}(\langle v\rangle, \sigma)$$
$$\{l_0, \ldots, l_n\} = \{l \mid \langle l, l'\rangle \in [\![S]\!]\hat{\rho}\hat{\sigma}\}$$
$$\langle(\textbf{set-car!}\ S_1\ S_2\ A), \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \Sigma\rangle \longrightarrow$$
$$\langle A, \hat{\rho}, [l_1, \ldots, l_n \leftarrow \hat{\sigma}(l_1) \cup [\![S_2]\!]\hat{\rho}\hat{\sigma}, \ldots, \hat{\sigma}(l_n) \cup [\![S_2]\!]\hat{\rho}\hat{\sigma}]\hat{\sigma}, \hat{\gamma}, \Sigma\rangle$$
$$\textit{where}\ \{l_1, \ldots, l_n\} = \{l \mid \langle l, l'\rangle \in [\![S_1]\!]\hat{\rho}\hat{\sigma}\}$$
$$\langle(\textbf{pair?}\ v\ S_1\ S_2\ A_1\ A_2), \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \Sigma\rangle \longrightarrow$$
$$pop(\hat{\gamma}, true([\![S_2]\!]\hat{\rho}\hat{\sigma}, A_1, \hat{\rho}', \hat{\sigma}', false([\![S_2]\!]\hat{\rho}\hat{\sigma}, A_2, \hat{\rho}', \hat{\sigma}', \Sigma)))$$
$$\textit{where}\ \langle l\rangle = \widehat{new}(\langle v\rangle, \hat{\sigma})$$
$$\hat{\rho}' = [v \leftarrow l]\hat{\rho}$$
$$\hat{\sigma}' = [l \leftarrow [\![S_1]\!]\hat{\rho}\hat{\sigma}]$$
$$\langle(S_0\ S_1\ \ldots\ S_n), \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \Sigma\rangle \longrightarrow pop(\hat{\gamma}', \Sigma')$$
$$\textit{where}\ \{c_1, \ldots, c_n\} = \{\langle \eta, \vec{v}, A, \hat{\rho}\rangle \mid \langle \eta, \vec{v}, A, \hat{\rho}\rangle \in [\![S_0]\!]\hat{\sigma}\}$$
$$\vec{\epsilon} = \langle [\![S_1]\!]\hat{\rho}\hat{\sigma}, \ldots, [\![S_n]\!]\hat{\rho}\hat{\sigma}\rangle$$
$$\langle \hat{\gamma}', \Sigma'\rangle = apply(c_1, \vec{\epsilon}, \hat{\sigma}, \ldots apply(c_n, \vec{\epsilon}, \hat{\sigma}, \langle \hat{\gamma}, \Sigma\rangle))$$
$$\langle(\textbf{halt}\ S), \hat{\sigma}, \hat{\gamma}, \Sigma\rangle \longrightarrow pop(\hat{\gamma}, \Sigma)$$

$$[\![c]\!]\hat{\rho}\hat{\sigma} = \{\alpha(c)\}$$
$$[\![v]\!]\hat{\rho}\hat{\sigma} = \hat{\sigma}(\hat{\rho}(v))$$
$$[\![(\textbf{lambda}\ \eta\ (v_1\ \ldots\ v_n)\ A)]\!]\hat{\rho}\hat{\sigma} = \{\langle \eta, \langle v_1, \ldots, v_n\rangle, A, \hat{\rho}\rangle\}$$

Figure 4.3: Domains and transition relation for the abstract machine.

$$
\begin{array}{l}
\mathit{apply}(\langle \eta, \langle v_1, \ldots, v_n \rangle, A, \hat{\rho} \rangle, \langle \hat{\epsilon}_1, \ldots, \hat{\epsilon}_n \rangle, \hat{\sigma}, \langle \hat{\gamma}, \Sigma \rangle) = \\
\quad \langle \hat{\gamma}', \hat{\gamma}(\eta, \hat{\rho}') = \hat{\gamma}'(\eta, \hat{\rho}') \rightarrow \Sigma, \Sigma \cup \{\langle A, \hat{\rho}', \hat{\sigma}' \rangle\} \rangle \\
\quad \begin{array}{rcl}
\mathit{where}\ \langle l_1, \ldots, l_n \rangle & = & \widehat{\mathit{new}}(\langle v_1, \ldots, v_n \rangle, \hat{\sigma}) \\
\hat{\rho}' & = & [v_1, \ldots, v_n \leftarrow l_1, \ldots, l_n]\hat{\rho} \\
\hat{\sigma}' & = & [l_1, \ldots, l_n \leftarrow \hat{\epsilon}_1, \ldots, \hat{\epsilon}_n]\hat{\sigma} \\
\hat{\gamma}' & = & [\langle \eta, \hat{\rho}' \rangle \leftarrow (\langle \eta, \hat{\rho}' \rangle \in \mathit{dom}(\hat{\gamma}) \rightarrow \hat{\gamma}(\eta, \hat{\rho}') \sqcup \hat{\sigma}', \hat{\sigma}')]\hat{\gamma}
\end{array} \\
\mathit{pop}(\hat{\gamma}, \{\langle A, \hat{\rho}, \hat{\sigma} \rangle\} \cup \Sigma) = \langle A, \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \Sigma - \{\langle A, \hat{\rho}, \hat{\sigma} \rangle\} \rangle \\
\mathit{true}(\hat{\epsilon}, A, \hat{\rho}, \hat{\sigma}, \Sigma) = \hat{\epsilon} \cap (\mathit{Locations} \times \mathit{Locations}) = \emptyset \rightarrow \Sigma, \Sigma \cup \{\langle A, \hat{\rho}, \hat{\sigma} \rangle\} \\
\mathit{false}(\hat{\epsilon}, A, \hat{\rho}, \hat{\sigma}, \Sigma) = \hat{\epsilon} - (\mathit{Locations} \times \mathit{Locations}) = \emptyset \rightarrow \Sigma, \Sigma \cup \{\langle A, \hat{\rho}, \hat{\sigma} \rangle\}
\end{array}
$$

Figure 4.4: Auxiliary functions used by the abstract transition relation.

evaluates to a singleton set consisting of one pair. Both the **car** and **set-car!** rules must anticipate their first argument evaluating to more than one pair.

For a conditional, the test will evaluate to a set of objects, some of which may be pairs and some of which may not be. The function *true* adds the true arm to the pending set if the test value contains a pair. Likewise, *false* does the same for the else arm if the test value contains something other than a pair.

The application rule must anticipate the operator evaluating to a set of multiple closures. The auxiliary function *apply* is used to apply each closure to its arguments. The environment and store are extended to bind the arguments, and the cache is updated appropriately. If the cache entry changes, the context of the closure's application has changed, and the closure's code needs to be evaluated in the updated execution context. Since the machine can process only one expression at a time, the machine maintains a pending set Σ of expression, environment, store triples awaiting evaluation. *apply* adds the changed triple to Σ.

The function $\widehat{\mathit{new}}$ is similar to *new* in that it is a deterministic function that returns a sequence of locations that can be used to create new bindings. By relaxing the assumption that $\widehat{\mathit{Locations}}$ be finite and setting $\widehat{\mathit{new}} = \mathit{new}$, every value manipulated by the machine is a singleton set, and the abstract machine essentially emulates the collecting machine. On the other hand, by restricting $\widehat{\mathit{Locations}}$ to be a finite set, the abstract machine is transformed into an abstract operational semantics for $\mathit{CPS}(A(\mathit{CS}))$ that builds a cache that safely approximates the cache computed by the collecting machine. For example, a monovariant analysis can be obtained by setting $\widehat{\mathit{Locations}}$ to be the variables in the text

$$\{\ \ldots,$$
$$\langle \eta_3, \{v_8 \mapsto v_8, \ldots\}, \{v_8 \mapsto \{(), \langle v_{11}, v_{12}\rangle\}, v_{11} \mapsto \{1, 2\}, v_{12} \mapsto \{(), \langle v_{11}, v_{12}\rangle\}, \ldots\}\rangle,$$
$$\langle \eta_1, \{v_6 \mapsto v_6, \ldots\}, \{v_6 \mapsto \{(), \langle v_{11}, v_{12}\rangle\}, v_{11} \mapsto \{1, 2\}, v_{12} \mapsto \{(), \langle v_{11}, v_{12}\rangle\}, \ldots\}\rangle,$$
$$\ldots\}$$

Figure 4.5: A portion of the cache computed by the abstract machine for the procedure *copy*.

of the program and setting $\widehat{new}$ to be the identity function on its first argument.

Assuming such a function for $\widehat{new}$ and the identity function for α, the abstract machine would yield the abstract store described in Figure 4.5. The figure is similar to Figure 4.2 in that both describe the values to which the continuations of *copy* are applied. The two caches are different, however, in that the abstract cache conservatively estimates the collecting machine's cache. According to the abstract cache, the continuations are applied to one or more of the infinite set of values represented by the following context free grammar.

$$\begin{array}{lcl} v_{12} & \rightarrow & \langle v_{11}, v_{12}\rangle \mid () \\ v_{11} & \rightarrow & 1 \mid 2 \end{array}$$

The first few lists described by this grammar are enumerated as follows.

```
(),
(1), (2),
(1 1), (1 2), (2 1), (2 2),
(1 1 1), (1 1 2), (1 2 1), ..., (2 1 1), ...
```

This infinite set of lists includes the list computed by the collecting machine: (1 2), so the abstract machine's cache indeed approximates the collecting machine's cache.

4.3 Regulating speed and accuracy

The accuracy of the abstract machine can be regulated to some extent by using different functions in place of the allocation function $\widehat{new}$. The identity function mentioned above is relatively inaccurate. Its effect on the abstract machine is to use exactly one location to approximate all the locations that may be bound to a variable. A more accurate analysis can be obtained by using a more precise allocation function. The extra precision comes from partitioning the values to which a variable may be bound among more than one location. For example, a partial evaluator may use two partitions: one for static values, *i.e.*, values known at specialization time, and one for dynamic values, *i.e.*, values unavailable until run time.

Consequently, two locations would be associated with each variable in the program. One location would be bound to static abstract values, and the other would be bound to dynamic abstract values. This strategy has been used in the polyvariant partial evaluator Schism [18]. As another example, a 1CFA [50] analysis can be obtained by using an allocation function that allocates different locations depending on the call site that results in the environment extension.

Varying $\widehat{new}$ has only a limited effect on the accuracy of the analysis, though. In particular, the decision to partition values must be made at the time the values are bound. Consider the following program fragment.

```
(lambda (x)
  (if (eq? (car x) 'lit)
      ...
      (if (eq? (car x) 'varref)
          ...
          ...)))
```

In this example, a polyvariant binding-time analyzer similar to the one described above would want to allocate two locations for x. One location would contain all the pairs whose car is static, and the other location would contain all other values. The problem, however, is that in order to effectively achieve this partitioning, *all* the pairs in the program must be partitioned in this way at the time they are allocated. This is expensive, especially since not all the pairs will necessarily flow to the program fragment being considered. A much better alternative would be to partition just those pairs that become bound to x during analysis.

Another limitation of the abstract machine is that it can be expensive to run. The source of the cost is the number of times that the body of each lambda expression, *i.e.*, basic block, must be analyzed. Given that n is the cardinality of the set $\widehat{Objects}$ and m is the number of variables free in a basic block, the block may have to be analyzed up to nm times before stabilizing. As a result, the worst-case complexity of a naïve implementation of the machine is $O(n^4)$ where n indicates the size of the input program. Although the running time of the analysis may not be bad in practice, it is important to guarantee a better worst-case bound for a general-purpose analysis.

Figure 4.6 gives a modification to the abstract machine that addresses both of these issues. The domain of caches is modified, and two new functions are introduced. The

$$\hat{\gamma} \in \mathit{Caches} = (\mathit{Tags} \times \widehat{\mathit{Environments}} \times \mathit{Indices}) \xrightarrow{\text{fin}} \widehat{\mathit{Stores}}$$

$$\begin{aligned}
&\mathit{apply}(\langle \eta, \langle v_1, \ldots, v_n \rangle, A, \hat{\rho} \rangle, \langle \hat{\epsilon}_1, \ldots, \hat{\epsilon}_n \rangle, \hat{\sigma}, \langle \hat{\gamma}, \Sigma \rangle) = \\
&\quad \langle \hat{\gamma}', \hat{\gamma}(\eta, \hat{\rho}', \pi(\hat{\sigma}')) = \hat{\gamma}'(\eta, \hat{\rho}', \pi(\hat{\sigma}')) \to \Sigma, \Sigma \cup \{\langle A, \hat{\rho}', \hat{\sigma}' \rangle\} \rangle \\
&\quad \mathit{where}\ \langle l_1, \ldots, l_n \rangle = \widehat{\mathit{new}}(\langle v_1, \ldots, v_n \rangle, \hat{\sigma}) \\
&\qquad \hat{\rho}' = [v_1, \ldots, v_n \leftarrow l_1, \ldots, l_n]\hat{\rho} \\
&\qquad \hat{\sigma}' = \Theta([l_1, \ldots, l_n \leftarrow \hat{\epsilon}_1, \ldots, \hat{\epsilon}_n]\hat{\sigma}) \\
&\qquad \hat{\gamma}' = [\langle \eta, \hat{\rho}', \pi(\hat{\sigma}') \rangle \leftarrow (\langle \eta, \hat{\rho}', \pi(\hat{\sigma}') \rangle \in \mathit{dom}(\hat{\gamma}) \to \hat{\gamma}(\eta, \hat{\rho}', \pi(\hat{\sigma}')) \sqcup \hat{\sigma}', \hat{\sigma}')]\hat{\gamma}
\end{aligned}$$

Figure 4.6: Revised auxiliary function *apply* for the abstract machine.

function π is a polyvariance function mapping $\widehat{\mathit{Stores}}$ to the new, finite domain *Indices*. It is used to further regulate the accuracy of the analysis. The function Θ is a projection operator mapping $\widehat{\mathit{Stores}}$ to $\widehat{\mathit{Stores}}$. It is used to regulate the rate at which the machine converges to a solution.

The change to the cache and the use of π together increase the accuracy of the analysis. π is used to divide the domain of abstract stores into a finite number of partitions. The cache domain is changed to identify each program point with a partition. The cache maps a program point/partition combination to an abstract store that approximates all of the abstract stores in the partition that arrive at the program point. The basic abstract machine can be emulated by setting π to be a constant function, but similar to $\widehat{\mathit{new}}$, more accurate analyses can be obtained by substituting a more discriminating function.

The projection operator Θ has the property that for all $\hat{\sigma} \in \widehat{\mathit{Stores}}$, $\hat{\sigma} \sqsubseteq \Theta(\hat{\sigma})$. By projecting an abstract store to a more general abstract store, the convergence to a stable cache is accelerated. Using a projection operator may cause the analysis to generalize beyond the most accurate solution, *i.e.*, the least fixed point, but in exchange the analysis may be faster.

Projection operators have two extremes. At one end the projection operator can be just the identify function, and the accuracy of the analysis is not sacrificed at all. This is the behavior of the basic abstract machine. At the other end, Θ can be defined such that for all $\hat{\sigma}$, $\Theta(\hat{\sigma}) = \top$, *i.e.*, the most approximate value. This yields a very fast analysis that collects little useful information. An example of a more useful projection operator is given in Chapter 6. That operator, for example, can bring the cost of the analysis down to $O(n)$ when there are no assignments and $O(n^2)$ when there are assignments, although since

assignments are rare in mostly-functional programs, the average running time is closer to $O(n)$.

4.4 Summary

In this chapter the flow analysis framework for CPS A-normal form Scheme was developed. First a collecting machine was defined based on the standard machine defined in Chapter 3. The collecting machine maintains a cache that describes the execution state that arises at each program point. When the machine terminates, the cache can be used to construct exact control- and data-flow graphs describing the execution of the program. Naturally, the collecting machine does not terminate on programs that would not terminate in the standard machine. The next step was to abstract the collecting machine to yield an abstract machine that specifies the basic flow analysis framework. The abstract machine also computes a cache, but for a given program, the cache is a conservative approximation of the cache computed by the collecting machine. Furthermore, as long as the codomain of the function $\widehat{new}$ is finite, the abstract machine is guaranteed to terminate in a finite number of steps. As a final step, the abstract semantics was refined to be parameterized over the accuracy of the computed cache and the rate at which the machine converges to a stable cache. This can, for example, bring the worst-case complexity of the analysis from $O(n^4)$ down to $O(n^2)$ and down to $O(n)$ in practice.

Chapter 5

Correctness

In this chapter we prove the correctness of the abstract machine developed in Chapter 4. The proof proceeds by proving that the abstract machine M^A is sound with respect to the collecting machine M^C. Strictly speaking, M^S should be proved sound with respect to M^C, but the proof is omitted since M^C is just an instrumented variant of M^S. Furthermore, the translation from core Scheme to CPS A-normal form should be proved correct, but the correctness of the core Scheme to A-normal form translation is given elsewhere [27], and conversion to continuation-passing style is a well-understood transformation.

We cannot use traditional proof techniques to demonstrate correctness, because the semantics of our language is expressed operationally. Historically, flow analyses have been characterized either denotationally [12, 43, 59] or logically [48]. In a denotational framework, a standard semantics and an abstract semantics for the object language are defined. A pair of functions, an abstraction function and concretization function, are also defined that relate values in the standard and abstract value domains. A correctness proof uses the functions to show that the standard denotation of an expression is mapped to the abstract denotation of the expression. In a logical framework, a dynamic semantics and static semantics are defined as logical systems. The execution of a program in the dynamic semantics consists of a (possibly infinite) proof. Likewise, an "abstract" execution of the program in the static semantics consists of a proof. A soundness proof is used to show that the static semantics is correct with respect to the dynamic semantics. The proof is an induction on the size of proofs in the static and dynamic semantics and shows that given an expression, the value derived for the term in the static semantics is a safe approximation of the value derived in the dynamic semantics.

These proof techniques are not applicable when proving correctness results about operational semantics characterized using machines. We instead use a proof technique based on *storage layout relations* that we call *revised storage layout relations.* The general idea is to show that the execution of one machine in some sense *respects* the execution of another machine. After introducing the basic proof technique, we use it to prove that the abstract machine is sound with respect to the collecting machine.

5.1 Revised storage layout relations

Revised storage layout relations is motivated by the storage layout relations technique devised for use in the VLISP project. In this section we first describe the basic storage layout relations technique and then describe our revision.

The goal of the VLISP project was to prove correct the Scheme 48 bytecode compiler and virtual machine [30, 31]. The correctness of the system was defined relative to a modified version of the denotational semantics for Scheme [17]. The correctness proof proceeded first by showing that the compiler generated correct bytecodes for an idealized virtual machine. The idealized virtual machine was designed to correspond closely to the denotational semantics. The close correspondence facilitated the proof that the idealized machine was faithful to the denotational semantics.

On the other hand, the idealized virtual machine did not correspond to a real virtual machine implementation. A real implementation must be concerned with object representations, *e.g.*, how numbers are represented in memory, garbage collection, and a variety of other details that are not handled by the denotational semantics for Scheme. In essence, the idealized machine is too abstract.

Therefore, the second part of the verification involved successively refining the idealized virtual machine to obtain a realistic virtual machine. On each step, a *concrete* machine was defined that mimicked the behavior of the *abstract* machine, except that the concrete machine made some details of the implementation, *e.g.*, data representation, explicit. Ultimately, the final virtual machine manipulated words of memory, supported garbage collection, etc. At each refinment step, the correctness of the refinement had to be shown. Storage layout relations was the proof technique used to do so.

5.1.1 Storage layout relations

For the VLISP work, the following notion of refinement was used to relate a concrete machine M^C and an abstract machine M^A. Let $\equiv$ be a relation between $halts^C$ and $halts^A$ such that $s^C \equiv s^A$ if and only if the states represent termination with the same answer.

Definition 5.1.1 (Refinment) Consider a binary relation $\sim_0 \subseteq states^C \times states^A$. M^C refines M^A via $\sim_0$ if and only if:

1. Every abstract initial state corresponds to some concrete initial state:

$$\forall s^A \in inits^A \,.\, \exists s^C \in inits^C \,.\, s^S \sim_0 s^A$$

2. Whenever $s^C \sim_0 s^A$ and M^C can compute s_0^C from s^C, then M^A can compute s_0^A from s^A and $s_0^C \equiv s_0^A$.

3. Whenever $s^C \sim_0 s^A$ and M^A can compute s_0^A from s^A, then M^C can compute s_0^C from s^C and $s_0^C \equiv s_0^A$.

The definition says that a concrete machine refines an abstract machine if three properties hold between the two machines. First, given a start state of the abstract machine we can find a related start state in the concrete machine. The other two properties constrain the relationship between the two machines as they execute. In particular, if the abstract machine makes a transition from one state to another there is a corresponding transition in the concrete machine. Similarly, for every transition in the concrete machine there must be a corresponding transition in the abstract machine.

Given this notion of refinement, we can proceed to define a particular kind of relation $\sim$, a storage layout relation, which is guaranteed to imply that M^C refines M^A. The definition relies on a similarity relation $\approx$ between traces of M^C and M^A, which is defined in terms of $\sim$. $\approx$ just imposes a particular relationship between states of the two machines as they execute.

Definition 5.1.2 (Sequential extension $\approx$) Consider a relation $\sim \subseteq states^C \times states^A$. Then $\approx$, the sequential extension of $\sim$, is the relation between traces of M^C and traces of M^A such that $T^C \approx T^A$ if and only if there exists a one-one, onto, and monotonic function R between the indices and $\forall \langle i, j \rangle \in R \,.\, T^C(i) \sim T^A(j)$.

In the following we use the notion of quasi-equality, written $x == y$, meaning that if x or y is defined, then $x \equiv y$. In particular, for states s and s', $s == s'$ implies that either both s and s' are halt states or else neither of them is a halt state.

Definition 5.1.3 (Storage layout relation) A relation $\sim \subseteq \mathit{states}^C \times \mathit{states}^A$ is a *storage layout relation* for M^C and M^A if and only if:

1. Every abstract initial state corresponds to some concrete initial state:

$$\forall s^A \in \mathit{inits}^A \,.\, \exists s^C \in \mathit{inits}^C \,.\, s^C \sim s^A$$

2. For every concrete trace, and abstract state s^A corresponding to its first member, some corresponding abstract trace starts from s^A:

$$\forall T^C, s^A \in \mathit{inits}^A \,.\, T^C(0) \sim s^A \Rightarrow \exists T^A \,.\, T^A(0) = s^A \wedge T^C \approx T^A$$

3. For every abstract trace, and concrete state s^C corresponding to its first member, some corresponding concrete trace starts from s^C:

$$\forall T^A, s^C \in \mathit{inits}^C \,.\, s^C \sim T^A(0) \Rightarrow \exists T^C \,.\, T^C(0) = s^C \wedge T^C \approx T^A$$

4. Of two corresponding states, if both are halt states, then they deliver the same answer, and otherwise neither is a halt state:

$$\forall s^C, s^A \,.\, s^C \sim s^A \Rightarrow \mathit{ans}^C(s^C) == \mathit{ans}^A(s^A)$$

The definition says that given a trace in one machine, a corresponding trace can be found in the other machine. Furthermore, when both machines halt, they halt in equivalent states. By the definition of $\approx$, there is some leeway in how a concrete and abstract computation may be related. In particular, the abstract computation can "stutter" while the concrete machine proceeds through two or more state transitions. This is illustrated by the diagram in Figure 5.1. The need for such a capability is obvious. For example, a more abstract machine may implement **cons** as an atomic operation, while a concrete machine may need several steps to allocate storage for the pair, fill each element, and tag it.

The main result of the storage layout relations proof technique is the following theorem.

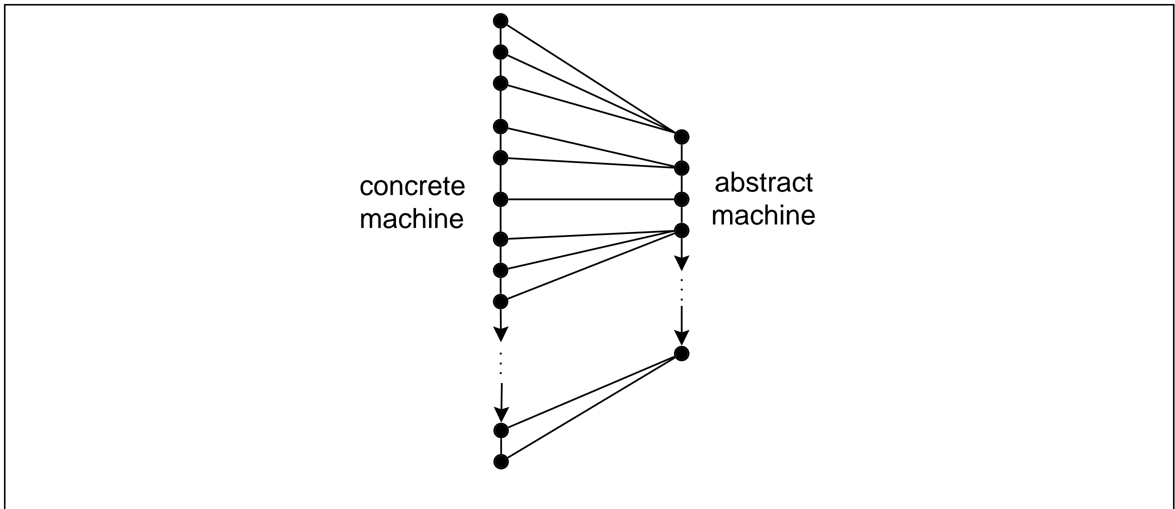


Figure 5.1: Illustration of the stuttering relationship between a concrete and abstract machine when using storage layout relations.

Theorem 5.1.1 (Storage layout relations guarantee refinement) Consider a relation $\sim \subseteq \mathit{states}^C \times \mathit{states}^A$. If $\sim$ is a storage layout relation, then M^C refines M^A via $\sim$.

Proof sketch: By case analysis and two inductions [30].

A consequence of this theorem is that the inductions on the lengths of computations is factored out of applications of the proof technique. To use the technique, one must merely demonstrate a storage layout relation between the states of the concrete and abstract machines. The result then follows from the above theorem.

5.1.2 Revised storage layout relations

Storage layout relations have two problems. The first is with the notion of refinement, which specifies only refinement over terminating computations. That is, the relationship between nonterminating computations of M^C and M^A do not play a role in determining whether M^C refines M^A. This is not acceptable for our purposes. The reason is that even though a computation in the collecting machine may not terminate, the abstract machine must terminate. Furthermore, it must still terminate with a safe approximation of the nonterminating collecting computation. The second problem is that the correspondence between computations is inappropriate in the context of our application. The correspondence is problematic for two reasons. First, our definition of correctness is in terms of soundness

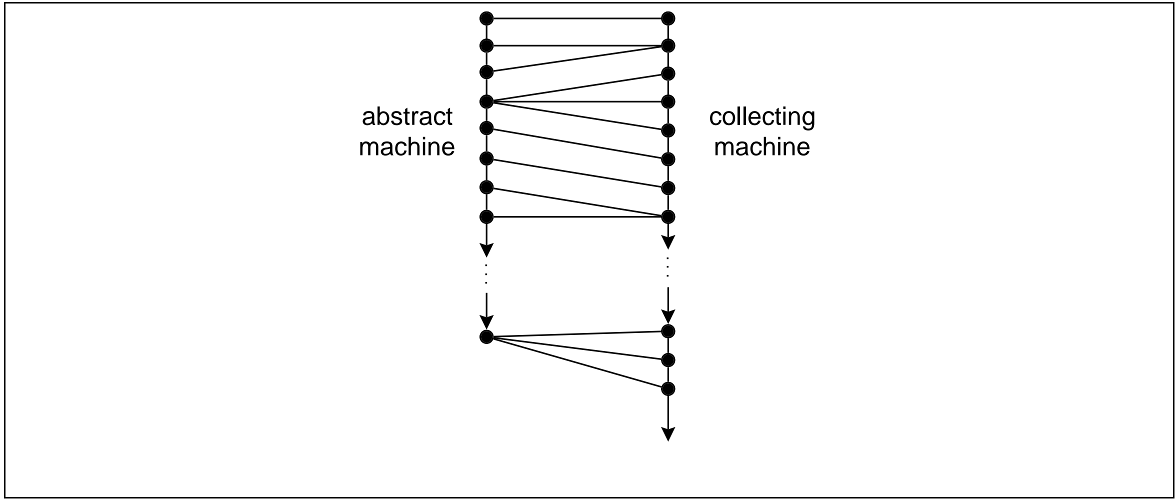


Figure 5.2: Illustration of the more complex stuttering relationship between the collecting and abstract machine when using revised storage layout relations.

only. It does not matter if for every computation in the abstract machine, we can find a corresponding computation in the collecting machine. We are concerned only with the other direction. Second, to prove the correctness of the abstract machine with respect to the collecting machine we will need to stutter both the collecting and abstract machines. The consequences of these changes are illustrated in Figure 5.2, and to realize them the constraint in the definition of $\approx$ that R be a function must be relaxed.

We now proceed to give our notion of revised storage layout relations, beginning with our revised definition of $\approx$. The definition uses the notion of a *monotonic relation*. A monotonic relation R between ordered domains X and Y satisfies the condition $\forall \langle x, y \rangle, \langle x', y' \rangle \in R\,.\,(x = x') \vee (x \sqsubseteq x' \Rightarrow y \sqsubseteq y')$.

Definition 5.1.4 (Revised sequential extension) Consider a relation $\sim \subseteq \mathit{states}^C \times \mathit{states}^A$. Then $\approx$, the sequential extension of $\sim$, is the relation between traces of M^C and traces of M^A such that $T^C \approx T^A$ if and only if there exists a one-one, onto, and monotonic relation R between the indices and $\forall \langle i, j \rangle \in R\,.\,T^C(i) \sim T^A(j)$.

Our notion of refinment is simplified considerably.

Definition 5.1.5 (Revised refinement) Consider a relation $\sim_0 \subseteq \mathit{states}^C \times \mathit{states}^A$. M^C *refines* M^A *via* $\sim$ if and only if for all traces T^A there exists a trace T^C such that $T^C \approx T^A$.

The definition is simplified by moving the burden of what it means to be correct to the definition of the state correspondence relation $\sim$. It also implies that proving one machine refines another will involve an induction on the length of computations in the abstract machine. As another way of looking at it, the storage layout relations techniques is structured carefully to factor application-independent details of proving that one machine refines another. The price of this factoring is that one cannot reason about nonterminating computations. Since we must be able to reason about nonterminating computations, we cannot factor the independent details and are left with simpler definitions.

5.2 Proving the abstract machine correct

Most of the work in the correctness proof involves establishing properties of the abstract machine that are needed to demonstrate the correspondence between abstract and collecting machine states. Because a number of properties need to be established and proved, the proof is divided into four subsections. Each subsection establishes properties needed for the next subsection until, in the last subsection, we are in a position to demonstrate that the abstract machine respects the collecting machine.

We begin by modifying the definitions of the collecting and abstract machines. The definition of these machines depends on modified transition relations, and their definitions will depend on the following restricted grammar of CPS A-normal form terms. The grammar contains all the terms except an application as the root of a term.

$$\begin{aligned} B \;=\; & (\textbf{let}\ ((v\ (\textbf{cons}\ v_1\ v_2\ S_1\ S_2)))\ A) \mid (\textbf{let}\ ((v\ (\textbf{car}\ S)))\ A) \mid \\ & (\textbf{pair?}\ v\ S_1\ S_2\ A_1\ A_2) \mid (\textbf{set-car!}\ S_1\ S_2\ A) \mid (\textbf{halt}\ v) \end{aligned}$$

The collecting machine is revised first followed by the abstract machine.

Definition 5.2.1 (Collecting Machine) Let $\mathit{CPS}(A(\mathit{CS}))$ be the language of CPS A-normal form expressions and *Stores* and *Caches* the domains defined in Figure 4.1. Then M^C is the state machine

$$\langle(\mathit{CPS}(A(\mathit{CS})) \times \mathit{Environments} \times \mathit{Stores} \times \mathit{Caches}), \mathit{inits}^C, \mathit{halts}^C, \mathit{acts}^C\rangle$$

where

1. inits^C is the set $\{\langle A, \emptyset, \emptyset, \emptyset\rangle \mid \exists M \in \mathit{CS}. A = \mathit{compile}(M)\}$;

2. $halts^C$ is the set such that for all $\rho \in Environments$, $\sigma \in Stores$, and $\gamma \in Caches$, $\langle(\mathbf{halt}\ v), \rho, \sigma, \gamma\rangle \in halts^C$ if and only if $v \in dom(\rho)$ and $v \in dom(\sigma)$; and

3. $acts^C$ is the transition relation defined as follows. Let $\rightarrow_0$ be the transition relation described in Figure 4.1. Then $\langle\langle A, \rho, \sigma, \gamma\rangle, \langle A', \rho', \sigma, \gamma'\rangle\rangle \in acts^C$ if and only if

$$\exists\langle B_0, \rho_0, \sigma_0, \gamma_0\rangle, \ldots, \langle B_n, \rho_n, \sigma_n, \gamma_n\rangle, \langle(S_0\ S_1\ \ldots\ S_n), \rho'', \sigma'', \gamma''\rangle\ .$$
$$\begin{aligned}\langle A, \rho, \sigma, \gamma\rangle &\rightarrow_0 \langle B_0, \rho_0, \sigma_0, \gamma_0\rangle \rightarrow_0 \ldots \rightarrow_0 \langle B_n, \rho_n, \sigma_n, \gamma_n\rangle \\ &\rightarrow_0 \langle(S_0\ S_1\ \ldots\ S_n), \rho'', \sigma'', \gamma''\rangle \rightarrow_0 \langle A', \rho', \sigma', \gamma'\rangle\end{aligned}$$

Definition 5.2.2 (Abstract Machine) Let $CPS(A(CS))$ be the language of CPS A-normal form expressions and $\widehat{Stores}$, $\widehat{Caches}$, and $Pending$ the domains defined in Figure 4.3. Then M^A is the state machine

$$\langle(CPS(A(CS)) \times \widehat{Environments} \times \widehat{Stores} \times \widehat{Caches} \times Pending), inits^A, halts^A, acts^A\rangle$$

where

1. $inits^A$ is the set $\{\langle A, \emptyset, \emptyset, \emptyset, \emptyset\rangle \mid \exists M \in CS . A = compile(M)\}$;

2. $halts^A$ is the set such that for all $A \in CPS(A(CS))$, $\hat{\rho} \in \widehat{Environments}$, $\hat{\sigma} \in \widehat{Stores}$, and $\hat{\gamma} \in \widehat{Caches}$, $\langle A, \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \emptyset\rangle \in halts^A$ if and only if $\hat{\gamma} \neq \emptyset$; and

3. $acts^A$ is the transition relation defined as follows. Let $\rightarrow_0$ be the transition relation described in Figures 4.3 and 4.4. Then $\langle\langle A, \hat{\sigma}, \hat{\gamma}, \Sigma\rangle, \langle A', \hat{\sigma}, \hat{\gamma}', \Sigma'\rangle\rangle \in acts^A$ if and only if

$$\exists\langle B_0, \hat{\sigma}_0, \hat{\gamma}_0, \Sigma_0\rangle, \ldots, \langle B_n, \hat{\sigma}_n, \hat{\gamma}_n, \Sigma_n\rangle, \langle(S_0\ S_1\ \ldots\ S_n), \hat{\sigma}'', \hat{\gamma}'', \Sigma''\rangle\ .$$
$$\begin{aligned}\langle A, \hat{\sigma}, \hat{\gamma}, \Sigma\rangle &\rightarrow_0 \langle B_0, \hat{\sigma}_0, \hat{\gamma}_0, \Sigma_0\rangle \rightarrow_0 \ldots \rightarrow_0 \langle B_n, \hat{\sigma}_n, \hat{\gamma}_n, \Sigma_n\rangle \\ &\rightarrow_0 \langle(S_0\ S_1\ \ldots\ S_n), \hat{\sigma}'', \hat{\gamma}'', \Sigma''\rangle \rightarrow_0 \langle A', \hat{\sigma}', \hat{\gamma}', \Sigma'\rangle\end{aligned}$$

The new definitions are similar to their original definitions. They differ only in how $acts^C$ and $acts^A$ are defined. In both cases, the definitions are still defined in terms of the relations defined in Figures 4.1 and 4.3 respectively. The difference is that given a machine state, the action relation maps the state to the next state corresponding to a closure entry point. The effect is that one action corresponds to the execution of one *thread*: a sequence of smaller steps that lead from program point to program point. The reason for larger steps

is that program points are a good place for synchronizing states between the abstract and collecting machines.

We first state and prove, albeit somewhat informally, that the abstract machine always terminates in a halt state.

Theorem 5.2.1 (Termination) All maximal computations of M^A are successful.

Proof: A computation is successful if it terminates in a halt state in a finite number of steps. A halt state is a state in which Σ is empty and $\hat{\gamma} \neq \emptyset$. Upon examining the machine, each transition $s_0^A \rightarrow s_1^A$ results in an element being removed from Σ. If only a finite number of elements can be added to Σ as the machine executes, then the machine will eventually terminate.
Consider how elements are added to Σ. On each step, an element is added only if the cache is updated. The cache is updated monotonically. By definition, the abstract domains are finite, so only a finite number of updates can be done. Hence, only a finite number of elements can be added to Σ. □

We now turn to establishing some core relationships between the abstract and collecting machines.

5.2.1 Consistency

Proving that a state in the abstract machine is related to a state in the collecting machine requires comparing the two states. Ultimately, this comparison will depend on relating abstract values and exact values. The following defines what it means for an abstract value to be related to an exact value.

Definition 5.2.3 (Object relation) Let $\simeq$ be the least relation on the set $(\mathcal{P}(\widehat{Objects}) \times \widehat{Environments} \times \widehat{Stores}) \times (Objects \times Environments \times Stores)$ and *abstract* be the least relation on the set $(Environments \times Stores \times \widehat{Environments} \times \widehat{Stores})$ that satisfy the following conditions.

1. $abstract(\rho, \sigma, \hat{\rho}, \hat{\sigma})$ if and only if $dom(\rho) = dom(\hat{\rho})$ and

$$\forall v \in dom(\rho).\langle \hat{\sigma}(\hat{\rho}(v)), \hat{\rho}, \hat{\sigma} \rangle \simeq \langle \sigma(\rho(v)), \rho, \sigma \rangle;$$

2. $\langle \hat{\epsilon}, \hat{\rho}, \hat{\sigma} \rangle \simeq \langle \epsilon, \rho, \sigma \rangle$ if and only if

(a) $\epsilon = c \Rightarrow \alpha(c) \in \hat{\epsilon}$;

(b) $\epsilon = \langle l_0, l_1 \rangle \Rightarrow \exists \langle l'_0, l'_1 \rangle \in \hat{\epsilon} \,.\, \langle \hat{\sigma}(l'_0), \hat{\rho}, \hat{\sigma} \rangle \simeq \langle \sigma(l_0), \rho, \sigma \rangle \wedge \langle \hat{\sigma}(l'_1), \hat{\rho}, \hat{\sigma} \rangle \simeq \langle \sigma(l_1), \rho, \sigma \rangle$; and

(c) $\epsilon = \langle \eta, \vec{v}, A, \rho' \rangle \Rightarrow \exists \langle \eta, \vec{v}, \hat{\rho}', A \rangle \in \hat{\epsilon} \,.\, \mathit{abstract}(\rho', \sigma, \hat{\rho}', \hat{\sigma})$.

The abstract machine manipulates sets of abstract values, while the collecting machine manipulates exact values. Thus, an exact value is related to an abstract value if its abstraction is in the set the abstract value denotes. For pairs, it must also hold that their constituent values are related. For closures, an abstract closure corresponds to an exact closure if and only if they share the same tag, formals, and body, and their environments are related by the relation *abstract*.

To relate abstract and collecting caches we define a relation that defines what it means for an abstract state to be consistent with a collecting state. Regardless of where in the execution the two machines are, the states are consistent if their caches are consistent.

Definition 5.2.4 (Consistency) Let $\equiv \subseteq \mathit{states}^A \times \mathit{states}^C$ be a relation such that

$$\langle A, \hat{\sigma}, \hat{\rho}, \hat{\gamma}, \Sigma \rangle \equiv \langle A', \rho, \sigma, \gamma \rangle$$

if and only if for all $\langle \eta, \rho, \sigma \rangle \in \gamma$ there exists $x \in X$ and $\hat{\rho} \in \mathit{Environments}$ such that

1. $\langle \eta, \hat{\rho}, x \rangle \in \mathit{dom}(\hat{\gamma})$; and
2. $\mathit{abstract}(\rho, \sigma, \hat{\rho}, \hat{\gamma}(\eta, \hat{\rho}, x))$.

An abstract cache is consistent with an exact cache if two conditions hold for every entry in the exact cache. First, there is a corresponding entry in the abstract cache. Second, the corresponding entry has the property that for every accessible value in the exact cache entry, there is a corresponding value in the abstract cache entry that satisfies the object relation $\simeq$.

This definition formalizes what it means for the abstract machine to be correct. The abstract machine is correct if for every state of its execution, the abstract cache at that state is consistent with the cache of the corresponding collecting state. The relation $\equiv$ will be an important component of the state correspondence relation $\sim$ between states^A and states^C.

Having defined $\equiv$, we may state and prove our first lemma, which says that actions in the abstract machine preserve consistency.

Lemma 5.2.1 If $s_0^A \equiv s^C$ and $s_0^A \to s_1^A$, then $s_1^A \equiv s^C$.

Proof sketch: By inspection of the transition rules for M^A. Upon transition from s_0^A to s_1^A, the abstract machine updates the cache monotonically. □

5.2.2 Formalizing the abstract machine's behavior

During execution, the abstract machine exhibits properties that need to be captured more formally. One property is that at a given abstract state, the abstract machine is executing a thread that may have a corresponding thread in the collecting machine. Another property is that for a given abstract state, the workset contains deferred threads that can correspond to collecting machine threads. The following relations formalize these observations.

Definition 5.2.5 Let *now* and *later* be relations on $states^A \times states^C$ such that given an abstract state $s^A = \langle A, \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \Sigma \rangle$ and a collecting state $s^C = \langle A', \rho, \sigma, \gamma \rangle$

$$\begin{aligned} now(s^A, s^C) &\Leftrightarrow A = A' \wedge abstract(\rho, \sigma, \hat{\rho}, \hat{\sigma}) \\ later(s^A, s^C) &\Leftrightarrow \exists \langle A'', \hat{\rho}', \hat{\sigma}' \rangle \in \Sigma \,.\, A = A'' \wedge abstract(\rho, \sigma, \hat{\rho}', \hat{\sigma}') \end{aligned}$$

If $now(s^A, s^C)$ then s^A represents the abstraction of the execution state s^C. *later* is an orthogonal definition that imposes a constraint on the workset. If $later(s^A, s^C)$, then a deferred thread that represents the abstract execution from s^C is in the workset.

Our informal knowledge about the abstract machine says that if a deferred thread is in the workset and the abstract machine executes a thread, then on the next step, the abstract machine will either execute the deferred thread or else it will still be in the workset.

Lemma 5.2.2 If $s^A \equiv s^C$ and $later(s^A, s^C)$, then for all $s_0^A \in states^A$, if $s^A \to s_0^A$, then either $now(s_0^A, s^C)$ or $later(s_0^A, s^C)$.

Proof: By inspection of the transition rules for the abstract machine. Let $s^C = \langle A', \rho, \sigma, \gamma \rangle$. In the course of the transition from s^A to s_0^A, *pop* is applied to a possibly modified set Σ'. By assumption, $\langle A'', \hat{\rho}, \hat{\sigma} \rangle \in \Sigma$ such that $A' = A''$, and $abstract(\rho, \sigma, \hat{\rho}, \hat{\sigma})$. If *pop* removes $\langle A'', \hat{\rho}, \hat{\sigma} \rangle$, then $now(s_0^A, s^C)$. Otherwise some other tuple will be removed and $\langle A'', \hat{\sigma} \rangle$ will remain in Σ', justifying $later(s_0^A, s^C)$. □

Using this lemma, we can further formalize our intuition that if a thread is deferred it will eventually be executed.

Lemma 5.2.3 If $s_0^A \equiv s^C$ and $later(s_0^A, s^C)$, then

$$\exists s_1^A, \ldots, s_n^A \in states^A \,.\, s^A \rightarrow s_0^A \rightarrow \ldots \rightarrow s_n^A \wedge now(s_n^A, s^C)$$

Proof: Assume the antecedent of the lemma. The task is to build a sequence of states $\langle s_0^A, \ldots, s_n^A \rangle$ that satisfy the consequent of the lemma. The sequence may be built in a finite number of steps. Assume that $s_i^A \equiv s^C$ and $later(s_i^A, s^C)$. Since $later(s_i^A, s^C)$, the machine must not be in a halt state, thus there is a state s_{i+1}^A, such that $s_i^A \rightarrow s_{i+1}^A$. By Lemma 5.2.1, $s_{i+1}^A \equiv s^C$. By Lemma 5.2.2, either $now(s_{i+1}^A, s^C)$ or $later(s_{i+1}^C,)$. In the latter, case the sequence must be extended further. In the former case, the sequence is terminated, and $n = i + 1$.
The sequence must be finite, since by Theorem 5.2.1 there are only a finite number of states in a maximal computation, and furthermore, the abstract machine must eventually reach a state s_n^A such that $now(s_n^A, s^C)$, since an element is removed from the pending set on each computation step. □

The next lemma establishes that the abstract machine preserves consistency in a lock-step execution with the collecting machine. It says that if two states s^A and s^C are consistent and $now(s^A, s^C)$, then after one step the resulting states will still be consistent.

Lemma 5.2.4 If $s^A \equiv s^C$, $now(s^A, s^C)$, and there exist s_0^A, s_0^C such that $s^A \rightarrow s_0^A$ and $s^C \rightarrow s_0^C$, then $s_0^A \equiv s_0^C$.

Proof: Assume the antecedent of the lemma. Proof proceeds by inspection of the transition rules. □

We can make another simple observation about the behavior of the abstract machine. When executing a thread, the abstract machine will determine one ore more threads that are the successor to the thread currently being executed. A thread may be discarded, however, if the abstract cache already correctly approximates the thread. Otherwise it will be added to the workset. Upon completing the execution of the thread, the abstract machine selects a thread from the workset. If a successor was added, either it will be chosen or it will not be chosen. The next lemma formalizes this behavior.

Lemma 5.2.5 If $s^A \equiv s^C$ and $now(s^A, s^C)$ and there exist s_0^A, s_0^C such that $s^A \rightarrow s_0^A$ and $s^C \rightarrow s_0^C$, then either exactly one of $now(s_0^A, s_0^C)$ or $later(s_0^A, s_0^C)$ holds or else neither holds.

Proof: Assume the antecedent of the lemma. Proof proceeds by inspection of the transition rules. □

The previous definitions and lemmas are concerned with what the abstract machine *will do* from a particular state. The next relation classifies states that satisfy a property describing what the abstract machine *has done* up to that state.

Definition 5.2.6 (Safety) Let $\mathit{safe} \subseteq \mathit{states}^A$ be a relation such that $\mathit{safe}(s^A)$ if and only if for all $s^C \in \mathit{states}^C$, if $s^A \equiv s^C$ and neither $\mathit{now}(s^A, s^C)$ nor $\mathit{later}(s^A, s^C)$, then for all $s^C_0 \in \mathit{states}^C$, if $s^C \rightarrow s^C_0$ then $s^A \equiv s^C_0$.

safe will be an extra condition on the state relation $\sim$ that guarantees that the abstract machine state is consistent with a one step execution of the concrete machine. The relation says that s^A is safe if for all states s^C with which it is consistent, then if an execution of the abstract machine from s^A will not execute the abstract thread corresponding to s^C, then s^A is already consistent with the action taken by s^C.

The next lemma establishes the fact that from a safe state, the the abstract machine always proceeds to another safe state.

Lemma 5.2.6 If $\mathit{safe}(s^A)$ and $s^A \rightarrow s^A_0$, then $\mathit{safe}(s^A_0)$.

Proof: Assume the antecedent of the lemma. To show $\mathit{safe}(s^A_0)$, assume an s^C such that $s^A_0 \equiv s^C$ and neither $\mathit{now}(s^A_0, s^C)$ nor $\mathit{later}(s^A_0, s^C)$ hold. We must show that for all s^C_0 such that $s^C \rightarrow s^C_0$, $s^A_0 \equiv s^C_0$. Assume an s^C_0 such that $s^C \rightarrow s^C_0$. There are two cases to consider.

Case: $s^A \equiv s^C$ and neither $\mathit{now}(s^A, s^C)$ nor $\mathit{later}(s^A, s^C)$. Then $s^A \equiv s^C_0$ by assumption, and by Lemma 5.2.1, $s^A_0 \equiv s^C_0$.

Case: $s^A \not\equiv s^C$ or $\mathit{now}(s^A, s^C)$ or $\mathit{later}(s^A, s^C)$. Proceed by cases on the form of s^A to determine the form of s^A_0.

Case: $s^A \not\equiv s^C$. This case is impossible, since if $s^A \not\equiv s^C$ and $s^A_0 \equiv s^C$, then executing s^A led to s^A_0 by adding information to the abstract cache of s^A through the function *apply*. But since $s^A_0 \equiv s^C$, the information added would concern the abstract state corresponding to s^C. Since $s^A \not\equiv s^C$, then either $\mathit{now}(s^A_0, s^C)$ or $\mathit{later}(s^A_0, s^C)$, which contradicts our assumption.

Case: $s^A \equiv s^C$ and $later(s^A, s^C)$. This case is not possible, since by Lemma 5.2.2, if $later(s^A, s^C)$, then either $now(s_0^A, s^C)$ or $later(s_0^A, s^C)$. But this contradicts our assumption that neither $now(s_0^A, s^C)$ nor $later(s_0^A, s^C)$ holds.

Case: $s^A \equiv s^C$ and $now(s^A, s^C)$. $s_0^A \equiv s_0^C$ holds by Lemma 5.2.4.

This concludes the proof. □

5.2.3 The correctness theorem

The definitions and relations in the previous subsections have imposed strong constraints on the behavior of the abstract machine. Given these relations, we are now able to define what it means for an abstract state to correspond to a collecting state.

Definition 5.2.7 (State relation) Let $\sim \subseteq states^A \times states^C$ be a relation such that $s^A \sim s^C$ if and only if $s^A \equiv s^C$, $safe(s^A)$, and either exactly one of $now(s^A, s^C)$ and $later(s^A, s^C)$ holds or neither holds.

With $\sim$ we can prove that M^A refines M^C. In order to facilitate the proof, however, we first define a somewhat stronger notion of correspondence. It is similar to $\sim$, but it precludes the possibility that for states s^A and s^C, $later(s^A, s^C)$.

Definition 5.2.8 Let $\asymp \subseteq \sim$ be a relation such that $s^A \asymp s^C$ if and only if $\neg later(s^A, s^C)$.

Theorem 5.2.2 M^A refines M^C via $\sim$.

Proof: We must find a one-one, onto, and monotonic relation R that satisfies the conditions given in Definition 5.1.4. We will construct R by induction on the length of collecting traces. The induction hypothesis is strengthened so that for all collecting traces of length i, $T^A \approx T^C$ and $last(T^A) \asymp last(T^C)$.

As the base case, let the length of T^C be 1. $T^C(0) \in inits^C$, so $T^C(0)$ has the form $\langle A, \emptyset, \emptyset, \emptyset \rangle$ Let T^A be a sequence of length 1 with $T^A(0)$ defined to be $\langle A, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. $T^A(0) \asymp T^C(0)$ holds on inspection. $T^A \approx T^C$ holds by letting $R = \{\langle 0, 0 \rangle\}$. Thus the base case holds.

For the induction step, let T^C be a trace of length i. By the induction hypothesis, there exists a T^A of length j such that $T^A \approx T^C$. Further assume there exists a $s_1^C \in states^C$ such that $T^C(i-1) \rightarrow s_1^C$, and let $T_0^C = T^C \frown \langle s_1^C \rangle$. The goal is to use T^A to construct a T_0^A such that $T_0^A \approx T_0^C$ and $last(T_0^A) \asymp last(T_0^C)$.

Let $s_0^A = \mathit{last}(T^A)$ and $s_0^C = \mathit{last}(T^C)$. By the induction hypothesis, $s_0^A \asymp s_0^C$. There are two cases to consider, depending on whether $\mathit{now}(s_0^A, s_0^C)$ or else neither $\mathit{now}(s_0^A, s_0^C)$ nor $\mathit{later}(s_0^A, s_0^C)$.

Case: $\mathit{now}(s_0^A, s_0^C)$. Then $s_0^A \rightarrow s_1^A$ and $s_0^C \rightarrow s_1^C$. By Lemma 5.2.4, $s_1^A \equiv s_1^C$. By Lemma 5.2.6, $\mathit{safe}(s_1^A)$. By Lemma 5.2.5, either exactly one of $\mathit{now}(s_1^A, s_1^C)$ or $\mathit{later}(s_1^A, s_1^C)$ holds or neither holds. We therefore conclude that $s_1^A \sim s_1^C$.

The construction of T_0^A depends on whether $\mathit{now}(s_1^A, s_1^C)$ or $\mathit{later}(s_1^A, s_1^C)$ holds or neither holds. There are two cases.

Case: $\mathit{now}(s_1^A, s_1^C)$ holds or neither holds. In this case, $s_1^A \asymp s_1^C$. Let $T_0^A = T^A \frown \langle s_1^A \rangle$ and $R' = R \cup \{\langle j, i \rangle\}$.

Case: If $\mathit{later}(s_1^A, s_1^C)$ holds, Lemma 5.2.3 supplies a sequence $\langle s_2^A, \ldots, s_n^A \rangle$ such that $s_1^A \rightarrow s_2^A \rightarrow \ldots \rightarrow s_n^A$ and $\mathit{now}(s_n^A, s_1^C)$. By Lemmas 5.2.1 and 5.2.6, $\forall 2 \leq i \leq n \,.\, s_i^A \sim s_1^C$. Since $\mathit{now}(s_n^A, s_1^C)$, $s_n^A \asymp s_1^C$. Let $T_0^A = T^A \frown \langle s_1^A, s_2^A, \ldots, s_n^A \rangle$ and let $R' = R \cup \{\langle j, i \rangle, \langle j+1, i \rangle, \ldots, \langle j+n-1, i \rangle\}$.

Case: Neither $\mathit{now}(s_0^A, s_0^C)$ nor $\mathit{later}(s_0^A, s_0^C)$. By the induction hypothesis and Definition 5.2.6, $s_0^A \equiv s_1^C$ and $\mathit{safe}(s_0^A)$. To establish that exactly one of $\mathit{now}(s_0^A, s_1^C)$ or $\mathit{later}(s_0^A, s_1^C)$ holds or neither holds, we must inspect s_1^C. There are two cases.

Case: $\mathit{now}(s_0^A, s_1^C)$ or neither $\mathit{now}(s_0^A, s_1^C)$ nor $\mathit{later}(s_0^A, s^C)$. Then $s_0^A \asymp s_1^C$. Let $T_0^A = T^A$ and let $R' = R \cup \{\langle j, i \rangle\}$. Then $T_0^A \approx T_0^C$ and $\mathit{last}(T_0^A) \asymp \mathit{last}(T_1^C)$.

Case: $\mathit{later}(s_0^A, s_1^C)$. By Lemma 5.2.3 there exists $s_1^A, \ldots, s_n^A$ such that $s_0^A \rightarrow s_1^A \rightarrow \ldots \rightarrow s_n^A$ and $\mathit{now}(s_n^A, s_1^C)$. By Lemmas 5.2.1 and 5.2.6, $\forall 0 \leq i \leq n \,.\, s_i^A \sim s_1^C$. Since $\mathit{now}(s_n^A, s_1^C)$, $s_n^A \asymp s_1^C$. Let $T_0^A = T^A \frown \langle s_1^A, \ldots, s_n^A \rangle$, and let $R' = R \cup \{\langle j, i \rangle, \ldots, \langle j+n-1, i \rangle\}$. Then $T_0^A \approx T_0^C$ via R', and $\mathit{last}(T_0^A) \asymp \mathit{last}(T_0^C)$.

This concludes the proof. □

5.3 Summary

In this chapter the abstract machine M^A which represents the flow analysis was proved correct with respect to the collecting machine M^C. The proof proceeded by showing that M^A respects M^C and used a technique based on *storage layout relations*. Storage layout relations is a technique used to demonstrate that a machine M respects another machine M' in the sense that at each state of the execution of M', there is a corresponding, equivalent state in the execution of M. The technique used here is similar to storage layout relations

in that a correspondence relation $\sim$ is established between states of the two machines, and a sequential extension of $\sim$ is defined that relates the states of computation sequences in the two machines. Storage layout relations differs from our technique in that it allows a clean separation between the induction argument on the length of computation sequences and the proof that the relation $\sim$ satisfies the properties necessary to justify the induction. On the other hand, our technique can prove that one machine respects another even on computations that do not terminate.

Chapter 6

Implementation

In this chapter the implementation of the analysis framework is described. The framework has been implemented in *Chez Scheme*, a production-quality Scheme implementation with an incremental, native-code compiler. Although the abstract semantics provides a direct path to implementation, three issues must be addressed. First, the semantics assumes that programs are closed, *i.e.*, there are no free variables. This is not a realistic assumption for practical use of the analysis. The second issue is the representation of the abstract store. A naïve representation of the store leads to a computationally expensive analysis. To be practical a more efficient representation is necessary. The final issue is the selection of a useful projection operator. Even though a good representation for the store can be found, the analysis is still too slow for interactive use. A tunable projection operator that allows the user to vary the speed/accuracy tradeoff of the analysis is necessary for the analysis to be practical. The first and third issues are addressed first in an overview of the analysis implementation. The second issue is considered separately.

6.1 Overview

The analysis framework is implemented in four passes within the compiler. The compiler's intermediate form is a direct-style representation of the input program. Consequently, the first pass is a preprocessor that converts the input program to CPS A-normal form. The second pass is the analysis. After analysis, a third pass decorates the abstract syntax tree with the information collected, and the fourth pass is a postprocessor that converts the

program back into the compiler's intermediate form. It is possible to combine the third and fourth passes, but this has not been done in the prototype implementation.

Since the normalized program must be converted back to direct style, the preprocessor must maintain enough information for the postprocessor to correctly perform its task. For example, CPS A-normal form explicitly names the continuation of conditional expressions to avoid duplication. Binding constructs are treated similarly. Although the continuation would not be duplicated, binding the continuation outside the binding form avoids having the context surrounding the binding form moved into the form's body when converting back to direct style. In addition, each expression that can cause a side effect, *i.e.*, a side-effecting primitive, is marked, and the conversion back to direct style **let**-binds the expressions in order to preserve the ordering constraint implied by the side effect.

Another aspect of the translation to CPS A-normal form is the treatment of primitives. In CPS A-normal form, a simple expression is either a variable reference, a literal, or a lambda expression. This simplifies the semantics considerably but leads to large normalized programs with many temporary variables. Furthermore, when converting back to direct style, order-of-evaluation constraints make it is difficult to move **let**-bound temporary expressions back into their original position. Therefore, the preprocessor considers primitive applications simple if they cannot cause a side effect. For example, the expression (**cons** (**car** (**cdr** x)) (**cdr** y)) would be considered simple, but the expression (**cons** (**set-car!** x y) 2) would not be.

Adding primitives to the set of simple expressions has three benefits. First, the number of temporary variables is reduced, which eliminates a small amount of overhead during analysis. Second, the postprocessor is able to translate the analyzed program back into a more compact direct-style program. Third, because there are fewer order-of-evaluation constraints on the postprocessed program, the compiler's register allocator has more freedom to select an ordering that minimizes register shuffling across procedure calls.

6.1.1 Accommodating free variables

As defined in Chapter 4, the analysis framework can process only closed programs. This is unacceptable for a practical implementation. Thus, references to free variables are handled by adding a unique object *unknown* to the set $\widehat{Constants}$ of abstract constants. References

to free variables, calls to foreign procedures, and calls to primitives whose return values are unknown are modified to return *unknown*.

The analysis must be modified to accommodate the consequences of adding *unknown*. In the context of the language of CPS A-normal form expressions, there are three places where special action must be taken if the *unknown* object is encountered. The first occurrence is when *unknown* is a constituent of the argument in a **car** expression. Since an unknown pair is being referenced, *unknown* must be added to the abstract value constituting the result of the evaluation. The second occurrence is when *unknown* is part of the first argument in a **set-car!** expression. In this case, the second argument must be considered an escaping value. The final occurrence is when *unknown* is part of the operator in an application. In this case the operands, including the continuation, also must be considered escaping.

An abstract value is considered escaping if it leaves the scope of the flow graph computed by the analysis. In the cases of **set-car!** and application, the analysis can neither track values stored in unknown pairs, nor can it track abstract values that are passed to unknown procedures. The analysis must safely accommodate abstract values that escape. For a pair that escapes, the analysis must assume that its constituents escape, and that the car is assigned an unknown value. For a closure that escapes, the analysis must assume that it is applied to unknown arguments.

Figure 6.1 gives a procedure for processing a list of abstract values that escape during one transition of the abstract machine. The procedure *escape!* is called initially with a list of escaping values and the abstract machine's current store and environment. It uses *escape-val* to process each escaping abstract value, collecting a list of closures that escape, a modified store, and a list of seen objects. The list of seen objects is necessary since the structures traversed by *escape!* may be cyclic. Once the values are processed, each closure that escapes is applied to unknown values in the modified store. Applying the closures in the context of the modified store is necessary for safety. For example, an unknown procedure may be called with a pair and a procedure that closes over the pair. The unknown procedure assigns a value to the pair and then calls the procedure it received, which presumably uses the pair. *escape!* returns the updated store.

```
(define escape!
  (lambda (absvals store)
    (let lp ((absvals absvals) (store store) (procs (make-empty-set)) (seen '()))
      (if (null? absvals)
          (begin
            (for-each
              (lambda (proc)
                (variant-case proc
                  (absclosure (formals)
                    (apply-closure proc (map (lambda (x) unknown) formals)
                      store))))
              (set->list procs))
            store)
          (call-with-values
            (lambda () (escape-val (car absvals) store procs seen))
            (lambda (store procs seen) (lp (cdr absvals) store procs seen)))))))

(define escape-val
  (lambda (absval store procs seen)
    (let lp ((ls (set->list absval)) (store store) (procs procs) (seen seen))
      (cond
        ((null? ls) (values store procs seen))
        ((memq (car ls) seen) (lp (cdr ls) store procs seen))
        (else (call-with-values
                    (lambda () (escape-obj (car ls) store procs (cons (car ls) seen)))
                    (lambda (store procs seen) (lp (cdr ls) store procs seen))))))))

(define escape-obj
  (lambda (obj store procs seen)
    (variant-case obj
      (absconstant () (values store procs seen))
      (absclosure () (values store (set-union (make-set obj) procs) seen))
      (abspair (v1 v2)
        (call-with-values
          (lambda () (escape-val (apply-store store v2) store procs seen))
          (lambda (store procs seen)
            (call-with-values
              (lambda () (escape-val (apply-store store v1) store procs seen))
              (lambda (store procs seen)
                (values (let ((x (apply-store store l1)))
                          (extend-store v1 (set-union unknown x) store))
                        procs
                        seen))))))))))
```

Figure 6.1: A procedure for handling escaping values.

6.1.2 Polyvariance

While the analysis framework supports polyvariant analyses, the implementation is fixed to be monovariant since the compiler is not currently able to use the information that would be gathered by a polyvariant analysis. This simplifies the implementation considerably. In particular, in a monovariant framework the environment can be assumed to be the identity function, and thus, the environment can be dropped.

Modifying the implementation to permit polyvariance is not difficult. Adding the environment back into the implementation is trivial, but other changes are necessary as well. Assuming monovariance, the implementation can associate one program point with each lambda expression in the program. In a polyvariant implementation this must be modified to permit multiple program points to be associated with each lambda expression, and when a procedure is applied, the correct program point must be selected based on the current environment and the index determined by the polyvariance operator. The details of this are dependent on the polyvariance operator and the criteria used to allocate locations, but the necessary structural changes are straightforward.

6.1.3 Projection

Besides polyvariance, the analysis framework also incorporates a very general projection operator to accelerate the analysis. The implementation incorporates a specific projection operator that can be tuned to vary the speed/accuracy tradeoff of the analysis. The operator is designed so that the programmer may easily correlate the effect of changing the operator's parameter with the results of the analysis.

The operator may be described as follows. With the exception of the locations associated with **let**- and **letrec**-bound variables, the analysis permits up to n updates to each binding l without interference. On subsequent updates, the abstract object *unknown* is added to the abstract value of l and the objects being added to the value of l are considered escaping.

This operator defines two bounds on the analysis' range of accuracy. When $n = 0$, the analysis performs an intraprocedural flow analysis that runs in one pass over the program. When $n = \infty$, the analysis is equivalent to a 0CFA analysis. From the programmer's perspective, the intuition behind n is that the flow analysis is allowed to make up to n calls to a procedure before the analysis quits and assumes a safe value for the procedure's arguments.

```
(define lub
  (lambda (s1 s2)
    (let ((locs (set-union (domain new-store) (domain old-store))))
      (extend-store locs
        (map (lambda (l) (set-union (apply-store s1 l) (apply-store s2 l))) locs)
        (make-empty-store)))))
```

Figure 6.2: Straightforward implementation of the least upper bound operator.

By tying the behavior of the projection operator to the program and the programmer's informal mental process of executing the program, the programmer can predict the behavior of the analysis under different values for n. This projection operator resultis in an analysis that is $O(n^2)$ in the presence of assignment, but since assignments are rare in mostly-functional programs, the average-case behavior is closer to $O(n)$.

6.2 Implementing the abstract store

Finding an efficient representation for the abstract store is difficult. Three strategies are given for implementing the abstract store. The first strategy shows how unnecessary work can be avoided when taking the least upper bound of two stores. The second strategy uses a "lazy" store and records dependencies to avoid even more unnecessary work. The final strategy employs an observation about the mutability of variables to improve on the second strategy. The final implementation strategy significantly improves the average running time of the analysis, but it does not lower the theoretical worst-case running time of the analysis. For general interactive use, the analysis must still be parameterized with a nontrivial projection operator such as the one described above to bring the cost of the analysis down to reasonable bounds.

6.2.1 A simple implementation

A straightforward implementation of the least upper bound operator is given in Figure 6.2. The procedure *domain* is used to determine the domain of each store, and *apply-store* is used to look up a location's binding in the store. If the location is unbound, the empty set is returned.

While this implementation is easy to understand, using it results in a large number of redundant basic block analyses. Consider the following example.

```
(let ((f (lambda (x f) (if (= x 0) 1.0 (* x (f (- x 1) f))))))
  (f 5 f)
  2)
```

Suppose that the abstraction operator α maps integer constants to $\{int\}$ and floating point numbers to $\{float\}$. Also suppose that during execution, the continuation of the call $(f\ 5\ f)$ is analyzed before the continuation of the call $(f\ (-\ x\ 1)\ f)$. Under these assumptions, the continuation of the call $(f\ 5\ f)$ will ultimately be analyzed twice: once when the argument to the continuation is bound to set $\{int\}$ and again when it is bound to $\{int, float\}$. This is unfortunate, though, because the value passed to the continuation is ignored, implying that the second analysis is redundant. The problem is that the least upper bound operator takes the set union of every location in the domains of the two stores regardless of which locations may or may not be referenced in the continuation of the point at which the least upper bound is taken.

Figure 6.3 gives a revised least upper bound procedure that addresses this problem. It takes an additional argument that is a list of live locations. This list is initially the locations named by the free variables of the lambda expression representing the program point at which the least upper bound is taken. The procedure *lub* then sweeps the store, tracing every accessible location and taking the set union at each location. Each new value is examined to find more reachable locations, which are then added to the live list. Using this least upper bound operator avoids redundant basic block analyses of the type illustrated above. In addition, the least upper bound operation is itself more efficient since usually not every location in the domains of the two stores has to be processed when taking the least upper bound. The combination of these two benefits significantly improves the average running time of the analysis.

Although the new least upper bound operator improves the performance of the analysis, the analysis is no longer faithful to the abstract semantics. Upon termination of the analysis, the abstract stores at each program point may not be accurate on locations that are not live from the continuation denoted by the program point. This is a problem only for consumers that depend on dead locations having correct abstract values. Most, if not all, transformations do not depend on the values of dead locations, however, so this is not a problem in practice.

```
(define lub
  (lambda (s1 s2 live)
    (let lp ((live live) (seen '()) (store (make-empty-store)))
      (if (null? live)
          store
          (let ((loc (car live)))
            (if (memq loc seen)
                (lp (cdr live) seen store)
                (let ((set (set-union (apply-store s1 loc) (apply-store s2 loc))))
                  (lp (append (foldl set '() reachable) (cdr live))
                      (cons loc seen)
                      (extend-store (list loc) (list set) store)))))))))

(define reachable
  (lambda (obj locs)
    (variant-case obj
      (absclosure () locs)
      (abspair (l1 l2) (cons l1 (cons l2 locs)))
      (absclosure (body env)
        (append (map (lambda (var) (apply-env env var)) (free-vars body))
                locs)))))
```

Figure 6.3: Revised algorithm which avoids unnecessary set unions.

6.2.2 Computing the least upper bound lazily

Despite the improvements gained from the modified least upper bound operator, the performance of the analysis is clearly still poor for large programs. The following example illustrates the problem.

```
(let ((g (lambda () ...)))
  (let ((h (lambda (x) (g) (car x))))
    (h (cons 1 2))
    (h (cons 3 4))))
```

In this example, the call (h (**cons** 1 2)) results in the call (g), which leads to a (terminating) computation of arbitrary size. Later is the call (h (**cons** 3 4)). This causes the binding of x to change. Using the least upper bound operator in Figure 6.3, the call (g) would have to to be reanalyzed since x is live in the continuation of the call. Depending on how g is defined, this may lead to an enormous amount of redundant analysis just to propagate the value of x through the control-flow graph to the point at which it is referenced.

Addressing this problem requires a more radical change to the representation of the store. Two changes are made. The first is the shift to a "lazy store", by which we mean that the act of computing a location's abstract value is deferred until the location is referenced. The second change is to record dependencies from the point at which a location is updated to the point at which the location is referenced. By making these changes, redundant analyses of the type described above can be avoided.

To motivate the new store representation, consider how a program point is analyzed. Some representation of the store is used to initiate the analysis. As the expression named by the basic block is analyzed, the store is updated functionally to yield a new store. The final store on completion of the expression's analysis contributes to the stores used to analyze the program points to which control flow leads.

This leads to the use of a global graph to represent the store at each program point of the analyzed program. Each program point is a node in the graph. At a given program point, the initial store may be thought of as a list I of stores that flow into the program point. The analysis of the program point proceeds by functional extension of the store represented by I. When the store is applied to a location l, the binding for l is either retrieved from the functional extension or else it is the set union of the values obtained by applying each element of I to l.

```
(define make-store cons)
(define store->base car)
(define store->ribs cdr)

(define extend-store
  (lambda (locs absvals store)
    (make-store (store->base store) (extend-rib locs absvals (store->ribs store)))))

(define-record base (in cache cache-dependencies trail-dependencies))
(define build-base (lambda () (make-base '() '() '() '())))

(define make-empty-ribs (lambda () '()))
(define extend-rib (lambda (locs absvals ribs) (cons (cons locs absvals) ribs)))

(define apply-ribs
  (lambda (ribs var)
    (if (null? ribs)
        #f
        (let ((i (list-index (caar ribs) var)))
          (if (= i -1)
              (apply-ribs (cdr ribs) var)
              (list-ref (cdar ribs) i))))))
```

Figure 6.4: Store representation for the second algorithm.

Figure 6.4 sketches part of the new store abstract data type. A store is a pair consisting of a base and a collection of ribs. The ribs express functional extensions to the store. The base is a data structure consisting of several fields. For the moment, the only relevant field is *in*, which contains a list of the stores that contribute to the initial store of a program point. A base data structure is associated with each program point and is used to construct the initial store needed to analyze the program point during the analysis.

Figure 6.5 gives a small program and its equivalent in CPS A-normal form. Figure 6.6 illustrates a store that may be constructed during the analysis using the new representation. In this particular graph, there is only one arc from η_5 to η_4, but another store graph could have two edges if program points were removed from the workset in a different order. A consequence of this representation of the store is that the graph is very similar to the control-flow graph computed by the analysis. In fact, for each program point η_i and η_j, if all the edges from η_i to η_j are replaced by just one edge, the resulting graph is precisely the control-flow graph.

```
(let ((f (lambda (x) (car x))))
  (let ((h (f (cons 1 2))))
    (f (cons 3 4))))

((lambda η0 (k0 f)
   (let ((v1 (cons l1 l2 1 2)))
     (f (lambda η0 (v2)
          ((lambda η0 (k1 h)
             (let ((v3 (cons l3 l4 3 4)))
               (f k1 v3)))
           k0
           v2))
        v1)))
 (lambda η0 (v4) (halt v4))
 (lambda η0 (k2 x) (let ((v3 (car x))) (k2 v3))))
```

Figure 6.5: A small example program and its CPS A-normal form equivalent.

Store references are more complicated in this representation, since the abstract value of a location is constructed lazily. Figures 6.7 and 6.8 give the code for store application. Given a store, its ribs are first checked to see if the location is locally bound to a value. If not, the cache is checked. The cache is a map from locations to abstract values, and if the location is in the cache, the cached value is returned. A cache miss indicates this is the initial reference to a location from the current program point. In this case, the procedure *slow-lookup* is used to construct the abstract value to which the location is bound, and the value returned is added to the cache.

The procedure *slow-lookup* must establish dependencies as it traverses the global graph. For each node traversed, a trail dependency is added. The dependency records the fact that the location *loc* is referenced at the program point *origin*. If the dependency has already been recorded at the node, then the node has already been crossed, and lookup is terminated with the empty set. If the dependency has not been previously added, the procedure *pseudo-apply-store* is called to obtain a value for *loc* from each of the stores flowing into the node.

The procedure *pseudo-apply-store* is similar to *apply-store*, but if a value for *loc* is found in the cache, a dependency on the cached value is recorded before returning the value. Otherwise *slow-lookup* is called to continue the search without modifying the cache.

Since the least upper bound operation on stores no longer exists directly, procedure

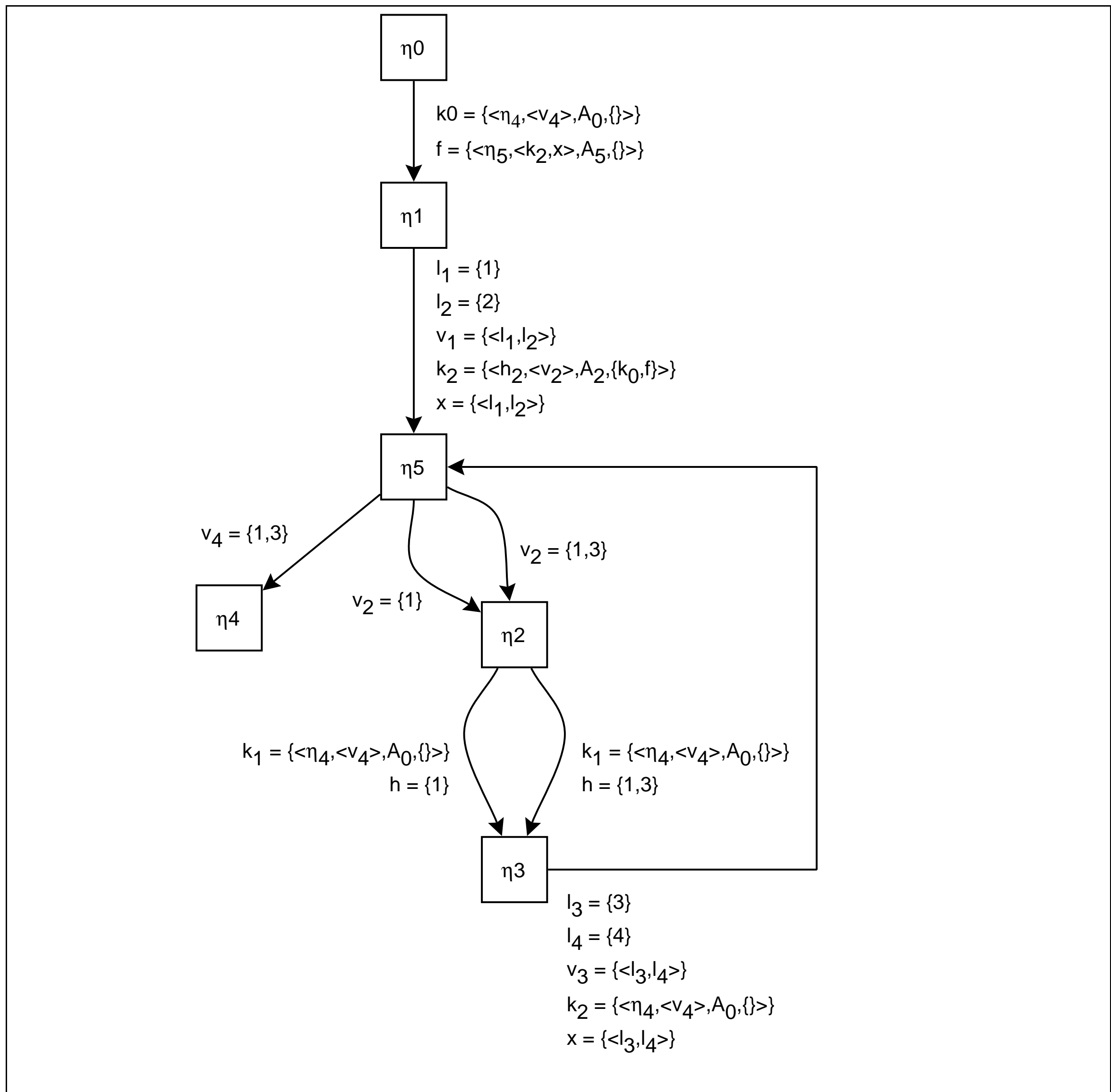


Figure 6.6: An example flowgraph.

```
(define apply-store
  (lambda (store loc)
    (or (apply-ribs (store->ribs store) loc)
        (let ((base (store->base store)))
          (let ((cache (base->cache base)))
            (or (lookup-cache cache loc)
                (set-base->cache! base
                  (extend-cache loc (slow-lookup base loc base) cache)))))))))

(define slow-lookup
  (lambda (base loc origin)
    (if (add-trail-dependency base loc origin)
        (foldl (base->in base) the-empty-set
          (lambda (store absval)
            (set-union (pseudo-apply-store store loc origin) absval)))
        the-empty-set)))

(define pseduo-apply-store
  (lambda (store loc origin)
    (or (apply-ribs (store->ribs store) loc)
        (let ((base (store->base store)))
          (cond
            ((lookup-cache (base->cache base) loc) =>
             (lambda (absval) (add-cache-dependency base loc origin) absval))
            (else (slow-lookup base loc origin)))))))
```

Figure 6.7: Operations for store lookup using the second algorithm.

```
(define add-trail-dependency
  (lambda (base loc origin)
    (add-dependency base loc origin base->trail-dependencies
      base->trail-dependencies!)))

(define add-cache-dependency
  (lambda (base loc origin)
    (add-dependency base loc origin base->cache-dependencies
      base->cache-dependencies!)))

(define add-dependency
  (lambda (base loc origin getter setter)
    (let ((alist (getter base)))
      (let ((pair (assq loc alist)))
        (if (null? pair)
            (begin (setter base (cons (list loc origin) alist)) #t)
            (and (not (memq origin (cdr pair)))
                 (set-cdr! pair (list-union (list origin) (cdr pair)))
                 #f))))))
```

Figure 6.8: Auxiliaries for store lookup using the second algorithm.

application also must be modified to work with the new store representation. In particular, the procedure *flow-in* given in Figure 6.9 is called to add the current store to the base data structure of the program point to which control is transferred. *flow-in* is called with an incoming store *store* and a base data structure *base* representing the program point to which control is flowing. If the *store* is the first store to be added to *base*, then the program point associated with *base* must be added to the workset. Otherwise, the store is added and the procedure *fake-apply-store!* is called to propagate the trail dependencies.

The procedure *fake-apply-store!* uses the trail dependencies established by store references to propagate abstract values. The trail dependencies record the fact that the store was applied to some locations at other points in the program. The procedure *pseudo-apply-store* is used to find the abstract value for each location, and the procedure *propagate-val!* is used to propagate the abstract value to each of the program points at which the store reference was originally performed.

The procedure *propagate-val!* uses the cache dependencies established by store references to propagate abstract values. Given a location and an abstract value for the location, *propagate-val!* checks to see if the abstract value contributes new information to the cache.

```
(define flow-in
  (lambda (store base)
    (if (null? (base->in base))
        (begin (set-base->in! base (list store))
               (add-to-workset! base))
        (begin (set-base->in! base (cons store (base->in base)))
               (fake-apply-store! store (base->trail-dependencies base))))))

(define fake-apply-store!
  (lambda (store dependencies)
    (unless (null? dependencies)
      (let ((loc (caar dependencies)))
        (let lp ((bases (cdar dependencies)))
          (if (null? bases)
              (fake-apply-store! store (cdr dependencies))
              (begin (propagate-val! (car bases) loc
                          (pseduo-apply-store store loc (car bases)))
                     (lp (cdr bases)))))))))

(define propagate-val!
  (lambda (base loc absval)
    (call-with-values
      (lambda ()
        (set-union? absval (lookup-cache (base->cache base))))
      (lambda (new changed?)
        (when changed?
          (add-to-workset! base)
          (set-base->cache! base
            (extend-cache (list loc) (list new) (base->cache base)))
          (let ((ls (assq loc (base->cache-dependencies base))))
            (when ls
              (for-each (lambda (b) (propagate-val! b loc new-absval))
                        (cdr ls)))))))))
```

Figure 6.9: Operations for procedure application using the second algorithm's store data type.

Supposing there is indeed new information to contribute, the entry is updated, the program point associated with the cache is added to the workset, and the new cache entry is propagated to other caches that may also depend on the value.

The procedures *fake-apply-store!* and *propagate-val!* use the dependency information established by store references to communicate abstract values directly from the program points at which values are bound to program points at which they are used. In fact, upon termination of the program this dependency information can be used to construct a data-flow graph between program points.

6.2.3 Taking advantage of immutable locations

The second algorithm works well, but it still has a high cost. It is expensive for two reasons. First, the initial reference to a location can be time consuming, since the store must be traversed to find all the points at which the location is updated with a value. Second, the trail must be propagated when a new store flows into a program point. These operations are expensive, since a location can be updated at any number of points in the control-flow graph, and since the least upper bound is built lazily, all of the accessible flow graph must be traversed to find potential updates.

While necessary in general, the vast majority of updates to a given location happen at exactly one point: the point at which a variable is bound or a data structure constructed. This is because while Scheme allows assignment, assignment is not encouraged. The key to our third and final algorithm is to optimize for locations that are assigned once while still permitting locations to be multiply assigned.

Assume for the moment that locations can be updated at only one point. Under this assumption, the value to which a location l is bound can be determined at the update point by taking the least upper bound of all the contributors to l's value. Furthermore, all references to l will refer to the binding established at the one program point at which l is bound. Applying these observations, we may conclude that a location's value and the points at which it is referenced can be directly associated with the location.

In our analysis, however, locations can be updated at multiple points, so our observations cannot be applied directly. They can, on the other hand, be incorporated into a hybrid algorithm which combines the second algorithm with an algorithm similar to the one just described. The hybrid algorithm is given in Figure 6.10.

```
(define-record loc (mutated? absval dependencies))

(define extend-store
  (lambda (locs absvals store)
    (for-each
      (lambda (loc absval)
        (unless (loc->mutated? loc)
          (call-with-values (lambda () (set-union absval (loc->absval loc)))
            (lambda (absval changed?)
              (when changed?
                (set-loc->absval! loc absval)
                (for-each add-to-workset! dependencies))))))
      locs
      absvals)
    (make-store (store->base store) (extend-rib locs absvals (store->ribs store)))))

(define extend-store/mutable
  (lambda (loc absval store)
    (set-loc->mutated?! loc #t)
    (for-each
      (lambda (base) (apply-store (make-store base (make-empty-ribs)) loc))
      (loc->dependencies loc))
    (make-store (store->base store)
                (extend-rib (list loc) (list absval) (store->ribs store)))))

(define apply-store
  (lambda (store loc)
    (or (and (not (loc->mutated? loc))
             (set-loc->dependencies! loc
               (list-union (list (store->base store))
                           (loc->dependencies loc)))
             (loc->absval loc))
        (apply-ribs (store->ribs store) loc)
        (let ((base (store->base store)))
          (let ((cache (base->cache base)))
            (or (lookup-cache cache loc)
                (set-base->cache! base
                  (extend-cache loc (slow-lookup base loc base) cache)))))))
```

Figure 6.10: Hybrid algorithm which uses a global store for variables that are assigned once.

The hybrid algorithm uses a different representation for locations. Instead of a symbol or some other atomic object, a record is used to represent a location. The record indicates whether or not the location has been mutated, *i.e.*, updated more than once, and if it is updated at only one point, it also records the value to which the location is bound and the program points where the location has been referenced.

The *apply-store* and *extend-store* operations are somewhat different from the second algorithm. If the location in question has been updated at more than one point, the algorithm behaves identically to the second algorithm. On the other hand, different operations can be used if the location is updated at just one point. Given a location l, *extend-store* executes by taking the set union of the old abstract value and the contributing abstract value to obtain a new abstract value for l. If the new value differs from the old value, the dependent program points are added to the workset. For *apply-store*, the program point corresponding to the basic block currently being executed is added to the locations list of dependencies, and the abstract value of l returned.

A new operation *extend-store/mutable* is introduced to extend the store when a mutation, *e.g.*, **set-car!**, occurs. It sets the flag on the assigned location and then simulates an *apply-store* from each point at which the location is referenced. This establishes the dependency information used by the second algorithm for future updates to the location's value.

While the *extend-store* operation involves somewhat more work than in the second algorithm, the more frequent operation *apply-store* has been made far more efficient. This final algorithm is the one currently used in the implementation and is used to obtain the results in the next chapter.

6.3 Summary

While the abstract semantics provides a direct path to the implementation of the flow analysis framework, effort must be taken to realize a practical implementation. In this chapter the implementation of the framework in a production Scheme compiler was described. The framework is implemented in four compiler passes. Three of the passes take care of converting the program between the compiler's intermediate form and CPS A-normal form. Furthermore, the conversion passes are careful to ensure that the conversion back to direct style will preserve the original structure of the program as much as possible.

Three issues must be resolved in a practical implementation of the analysis. First, it must be possible to analyze programs with free variables, so the implementation must be prepared to safely handle free references. This requires a mechanism for allowing abstract values to escape the flow graph computed by the analysis.

The second issue is the efficiency of the analysis and in particular the implementation of the abstract store. Three algorithms were presented to improve the performance of the analysis. The first algorithm offers only a marginal improvement but has the advantage of being easy to implement. The second algorithm is significantly more complicated but improves the performance considerably by eliminating a great deal of redundant analysis. Furthermore, the data structures maintained by the algorithm can be used directly to construct both the control- and data-flow graphs for the analyzed program. The third algorithm is an incremental improvement on the second algorithm. It optimizes the common case of a location being updated at only one point in the control-flow graph. Consequently, the third algorithm yields better results in practice than the second algorithm and performs considerably better than the naïve algorithm: running times are reduced from hours to minutes or seconds. Although the algorithms improve the average running time of the analysis, they do not improve on the theoretical complexity of the analysis. Assuming trivial projection and polyvariance operators, the analysis remains $O(n^4)$ with respect to the size of the program being analyzed.

This motivates the third issue, which is the need for an intuitive projection operator that can be tuned to vary the speed/accuracy tradeoff of the analysis. The operator can be used to vary the analysis between two extremes. At one extreme the operator enforces a one-pass analysis that is $O(n^2)$ in the presence of assignment and closer to $O(n)$ since assignments are rare in mostly-functional languages. At the other extreme is a $O(n^4)$, 0CFA-style analysis that is slower but collects more precise information. At the same time, the procedure used for restricting the accuracy of the analysis is intuitive to the programmer, and the programmer can understand the effects of parameterizing the projection operator directly in terms of the source program.

Chapter 7

Applying the analysis

In this chapter we use the flow analysis to optimize procedure representation and procedure call. Procedures are of central importance in Scheme and other higher-order languages. Procedures are first-class citizens, and programming style for higher-order languages encourages the use of many procedures with relatively small bodies. Loops in Scheme, for example, are written as tail recursive procedures as opposed to using explicit loop syntax. Creating procedures and calling procedures are not unusually expensive operations, but because of their frequency, optimizing them can yield good performance improvements.

7.1 Procedure representation and calling convention

Procedure creation is straightforward. A procedure is represented using a display closure [24]. Thus procedure creation corresponds to heap allocating a variable-length data structure and filling it with a code pointer and the procedure's free variable values.

Procedure calling conventions vary depending on whether the call is a tail call or nontail call. For a nontail call, registers that are live across the call must be saved to the stack. The callee's frame is then constructed by evaluating the arguments and operator. Some of the arguments will go on the stack. Other arguments will go in registers, and the operator will go in a distinguished *cp* register. After the arguments and operator are evaluated, a sequence of operations is performed to complete the transfer of control to the callee:

1. a type test is done to ensure that the operator is a closure,

2. the accumulator is loaded with the number of arguments being passed,

3. the frame pointer is adjusted to point to the base of the new frame,
4. the code pointer is fetched from the closure and control transferred to it, and
5. on entry to the called procedure, the argument count is checked for the proper number of arguments.

On return, the frame pointer is decremented to its original value and the live registers restored.

The calling convention for tail calls is similar. No registers are live across the call, so register saves are unnecessary. Also, because Scheme implementations are required to be properly tail recursive [17], the callee's frame must be moved down to overwrite the caller's frame before control is transferred to the callee.

Procedure call and procedure creation can be optimized by

Eliminating code pointer references If only one procedure is called from a call site, and the identity of the procedure is known, the code generated for the call can avoid referencing the closure's code pointer and instead jump directly to the procedure's entry point.

Eliminating *cp* loads Again, if only one procedure is called from a call site, and the identity of the procedure is known, the code generated for a call can avoid loading the *cp* register if it is not live in the called procedure's code.

Eliminating closure type checks If the operator will always evaluate to a procedure, the closure type check can be avoided.

Eliminating argument count checks If all calls to a fixed-arity procedure pass the correct number of arguments, the argument count check can be avoided.

Eliminating argument count loads If all procedures called from a call site avoid the argument count check, the call site can avoid loading the argument count.

Eliminating closure construction The closure for a lambda expression with no free lexical variables can be created at compile time. Furthermore, if no call site needs to reference the closure for the code pointer, then the closure can be eliminated.

Recognizing loops A loop is a recursive procedure where calls to itself occur only in tail position, and the loop is initiated from just one call site. A loop is optimized in two ways. First, a closure is not constructed for the procedure. Second, instead of generating a procedure call at the call site, the compiler puts the arguments (iterands) in registers and generates the code for the loop body inline. Tail recursive calls are then treated as branches to the loop head.

Some of these optimizations were being done in the compiler before the flow analysis was incorporated. In particular, loop optimization and closure elimination were done, and certain calls were recognized as *direct calls*. A direct call could be optimized by not loading the *cp* register unless necessary as well as skipping the closure type check and the argument count load and check.

Although several optimizations were done, the analysis necessary to justify them was collected on an *ad hoc* basis and performed no interprocedural analysis. The analysis used lexical properties of the program to determine when a variable could be bound to just one procedure and how the variable was used. Consider the following code as an example.

```
(letrec ((even? (lambda (x) (if (= x 0) #t (odd? (- x 1)))))
         (odd? (lambda (x) (if (= x 1) #t (not (even? x))))))
  (odd? 10))
```

The *ad hoc* analysis would determine that both even? and odd? are bound to procedures, eliminate both procedures, and treat all calls to them as direct calls.

The compiler could not optimize programs like the following, however.

```
(let ((f (lambda (g) (g))))
  (f (lambda () 2)))
```

In this example the compiler would treat the call to g as a generic call, since it did not do the interprocedural analysis necessary to determine that only one procedure arrived at the call site (g). Despite its limitations, the *ad hoc* analysis was still effective. It could, for example, reduce a DFA implemented using **letrec**-bound procedures to a state machine that created no closures and used direct calls to effect transitions between states.

The *ad hoc* analysis can be replaced by an instantiation of our flow analysis framework. Flow information alone is sufficient to justify procedure call optimizations. Additional lexical information is used to eliminate closures and detect loops. Potential loops are recognized syntactically as instances of one of the two following forms.

$(\textbf{letrec}\ ((f\ (\textbf{lambda}\ (x_1\ \ldots\ x_n)\ e_0)))\ (f\ e_1\ \ldots\ e_n))$

$((\textbf{letrec}\ ((f\ (\textbf{lambda}\ (x_1\ \ldots\ x_n)\ e_0)))\ f)\ e_1\ \ldots\ e_n)$

Forms matching one of these two patterns are potential loops only, because although the loop is initiated from just one call site, static information is necessary to ensure that references to the loop in the loop body are used only as operators in tail calls. In the old compiler, the *ad hoc* information collected determined this. In the new compiler the flow information collected by the analysis makes the determination.

Eliminating closures is more complex. If the lambda expression has no free lexical variables, it can be trivially built at compile time. If all of the callers reference the lambda expression's code directly, the closure may be eliminated instead. Although this is seemingly straightforward consider the even?/odd? program above. Since all call sites are known, the closures are not used, but each procedure has the other as a free lexical variable and thus neither can the closures be eliminated nor can the *cp* loads be eliminated at the call sites. This is unfortunate, since both closures are indeed unnecessary.

To handle programs like the even?/odd? example, the compiler uses a more complex algorithm for determining the free variables of a lambda expression. If a free variable is a reference to the procedure itself, then the variable is not considered free, since its value does not need to be stored in the closure. It does not need to be stored in the closure, because its value is located in the *cp* register at run time. Other free variables are considered genuinely free if they are referenced for value in nonfunction position or are assigned. A free variable in function position is considered free if its value is unknown at compile time or if it is a procedure whose closure pointer is needed.

These conditions are cyclic, however. A procedure will need its closure pointer if it has free variables, but determining if it has free variables requires knowing if other called procedures need their closure pointer. Thus the optimizer optimistically assumes that no procedure will need its closure pointer. If on subsequent analysis it is determined that a procedure P bound to v needs its closure, the optimizer adjusts the procedures which close over v to note that v is indeed free. The effect is similar to garbage collection: assume all data is garbage and use the roots to determine what is live.

7.2 Instantiating the analysis

The original compiler consisted of six passes excluding macro expansion. The compiler kept the input program in direct style throughout compilation, and introduced and dropped record types as needed to convey information to subsequent passes. Since the compiler was optimized for compile-time speed, each pass performed several operations, and some operations were spread across multiple passes. For example, *call-with-values* optimization [4] and loop optimizations occurred in the first pass, assignment conversion happened in the first two passes, and call optimizations were spread across the first three passes. While the compiler was indeed fast, this design was not modular and prevented a straightforward integration of the analysis.

Consequently the compiler was restructured in order to use the flow analysis. The first two passes were divided into ten passes that perform roughly the following actions:

1. Inlining and some minor simplifications,

2-3. Assignment conversion,

4. Conversion to continuation-passing style,

5. Computation of solution to flow analysis,

6. Annotation of program with flow information,

7. Conversion back to direct style,

8. *call-with-values* optimization, **or** recognition, compile-time environment construction,

9. Loop recognition, interrupt check insertion, and

10. Call optimization, establishing locations for variables, more code simplification.

Although the number of passes increased considerably, the cost of the extra passes except for pass 5 do not cost much.

Passes 4-7 constitute the flow analysis. It is a monovariant instantiation of the general analysis framework, and the implementation is described in Chapter 6 with two deviations. The first deviation is that the solver assumes assigned locations will eventually escape. Thus no attempt is made to preserve the accuracy of a location's abstract value before and

after assignment. As discussed in Chapter 6, maintaining the accuracy is not much more expensive that losing it, but the implementation is more complex. An early implementation of the analysis did in fact implement the more complex algorithm, but there was no measurable benefit for the call optimizations described in this chapter. Thus the more complex algorithm was discarded in favor of a simpler implementation in which all locations are viewed as being assigned once.

The second deviation is that loop recognition is anticipated in the conversion back to direct style. Although the implemented solver does not impose a fixed order of evaluation, the conversion back to direct style does not assume this and thus maintains the order of evaluation fixed by the conversion to CPS. A naïve transformation back to direct style would therefore **let**-bind the operator in a direct style code fragment matching

((**letrec** ((v (**lambda** (v_0 ...) b_0 b_1 ...))) v) e_0 ...)

This pattern, however, is one of the patterns recognized for loop optimization. Therefore, in the conversion back to direct style the CPS equivalent of this pattern is recognized and folded back into a direct-style pattern recognized by the loop optimizer.

As described in Chapter 6, the projection operator is a "dial" that the user can tune to vary the cost/accuracy tradeoff. When the dial is set to 0, the result is a fast one-pass analysis. When set to ∞, the result is a 0CFA-style analysis that can be slower but more accurate.

7.3 Experiments

The restructured compiler was run on a series of 27 benchmarks. Fifteen of the benchmarks are the non-I/O benchmarks from the Gabriel benchmark suite [29]. The other twelve are described in Figure 7.1. The texer, softscheme, similix, ddd, and compiler benchmarks are constructed from existing software packages. The other seven benchmarks are in the public domain. The dynamic, graphs, lattice, nbody and tcheck benchmarks for example, have been used by Wright [57] and Jagannathan and Wright [37] to study the benefits of type check elimination.

Some changes were made to the benchmarks. All of the benchmarks were modified to interface to the benchmarking code. Also, the Gabriel benchmarks were encapsulated using a **let** expression so that the entire benchmark could be submitted to the compiler at

Benchmark	lines	Description
compiler	30,000	Chez Scheme recompiling itself
texer	3,000	Scheme pretty-printer with TEX output
softscheme	10,000	Wright's soft typer [57] checking a 2,500 line program
similix	7,000	self-application of the Similix [6] partial evaluator
ddd	15,000	hardware derivation system [8] deriving Scheme machine [11]
em-fun	490	EM clustering algorithm in functional style
em-imp	460	EM clustering algorithm in imperative style
dynamic	2,200	a dynamic type inferencer applied to itself
graphs	500	counts the number of directed graphs with particular properties
lattice	200	enumerates the lattice of maps between two lattices
nbody	850	Greengard multipole algorithm for computing gravitational forces
tcheck	260	a polymorphic type inferencer for Scheme

Table 7.1: Benchmarks in addition to the Gabriels.

one time. The flow analysis must have large portions of the program available in order to construct a useful flow graph, and for the gabriel benchmarks in particular, the top-level definitions are too small for the flow analysis to analyze profitably. With the exception of the texer, softscheme, similix, ddd, and compiler benchmarks, the other benchmarks were already encapsulated.

Each benchmark was compiled using the modified *Chez Scheme* compiler with inlining enabled and run-time type checking disabled. The compiled benchmarks was then run on test data. Benchmarking was completely automated. One baseline was established by compiling the benchmark with the flow analysis-justified optimizations disabled. Another baseline was established with the optimizations enabled and the analysis dial set to ∞. Using the results of this second run, the benchmarking code determined the number of iterations n required for the analysis to stabilize. It then ran the benchmark three more times, setting the dial at 0, $n/3$, and $2n/3$. For runs on test data, three runs were done and the average of the three runs taken for timing results. For smaller benchmarks, *e.g.*, the Gabriels, on each of the three iterations the benchmark was run multiple times to accumulate more substantial CPU times and the divided over the number of runs to obtain a single-run time. The raw data collected from the benchmarks was then synthesized and LATEX tables summarizing the results automatically produced to eliminate manual, *i.e.*, error prone, processing.

Both static and dynamic data was collected from the benchmark runs. Static data included the time for the analysis to run, the total compile time, and the number of operations, *e.g.*, closure loads, inserted into the code that had the potential to be eliminated

by optimization but were not since the analysis could not justify it. These numbers were obtained by instrumenting the compiler and maintaining running sums over the evaluation of each top-level expression in the benchmark. The static statistics were then retrieved after the entire benchmark was compiled. Dynamic data included the run time of the compiled benchmark and dynamic execution counts of the operations measured statically at compile time. Dynamic counts were obtained by generating instrumented code and retrieving the results after the compiled benchmark had run. The run time of the benchmark was measured using uninstrumented code.

7.4 Results

Figure 7.2 gives compile times and run-time speedups for the benchmarks. Two compile times are given in seconds. The first is the time for the analysis, passes four through seven, to run. The second is the total compile time. The run-time speedups for each benchmark are the speedups over the unoptimized version of the benchmark. These times were collected on an SGI Onyx with two Mips 4400 processors running at 150Mhz, 128 Mb main memory, 16 Kb each of first-level instruction and data cache, and 1 Mb of unified second-level instruction and data cache.

The compile times for the benchmarks are encouraging. They indicate that even for large, closed programs such as softscheme, the analysis is able to perform acceptably fast for interactive use at $n = 0$, and the time at $n = \infty$ is not too excessive.

Given the processor configuration, the run-time speedups are reliable only for the first five benchmarks, all of which are large. For the smaller benchmarks, it appears that cache behavior prohibits reliable timings. For example, at $n = 0$ the lattice benchmarks reports a 6% slowdown, while the numbers below indicate that in fact the compiler was able to optimize the program.

A more reliable measure of performance to examine static and dynamic counts of checks inserted and checks executed. Tables 7.3 and 7.4 report the improvements the compiler was able to make statically. Figure 7.3 reports closures eliminated, loops recognized, and arity checks inserted. For each category, the number of potential candidates for optimization are given along with the number affected at $n = 0$ and $n = \infty$. For the compiler and softscheme benchmarks, there was a reduction in closures eliminated and loops recognized between $n = 0$ and $n = \infty$. This is due to necessary dead code elimination, as procedures

	compile time (sec)				run time	
	analysis		total			
	$n = 0$	$n = \infty$	$n = 0$	$n = \infty$	$n = 0$	$n = \infty$
compiler	14.3	35.9	120.4	144.3	14%	24%
texer	0.4	1.1	5.1	6.0	36%	41%
softscheme	7.7	25.8	84.5	101.3	5%	6%
similix	2.6	5.0	34.7	35.5	15%	11%
ddd	2.0	2.0	20.4	20.0	5%	8%
em-fun	0.2	0.3	2.4	2.3	49%	42%
em-imp	0.4	0.4	2.4	2.3	66%	72%
dynamic	0.6	3.3	7.4	10.3	14%	16%
graphs	0.1	0.1	0.9	0.8	29%	24%
lattice	0	0	0.4	0.4	53%	54%
nbody	0.3	0.3	2.8	2.6	50%	51%
tcheck	0	0.1	0.5	0.5	7%	11%
triang	0	0	0.2	0.2	94%	94%
traverse	0	0	0.4	0.4	4%	-6%
takr	0.4	5.9	4.3	10.0	33%	33%
takl	0	0	0.1	0.1	21%	21%
tak	0	0	0	0	31%	31%
puzzle	0.1	0	0.5	0.5	41%	41%
fft	0	0	0.4	0.4	17%	17%
div-iter	0	0	0.1	0.1	64%	64%
destruct	0	0	0.1	0.1	62%	62%
deriv	0	0	0.1	0.1	-10%	-10%
dderiv	0	0	0.1	0.1	0%	0%
ctak	0	0	0.1	0.1	19%	19%
cpstak	0	0	0.1	0.1	46%	53%
boyer	0	0	0.3	0.3	35%	35%
browse	0	0	0.3	0.4	25%	47%

Table 7.2: Compile times and run-time speedups for all benchmarks. Run time speedups are improvements over the unoptimized benchmarks.

	closures eliminated			loops recognized			arity checks inserted		
	max	$n = 0$	$n = \infty$	max	$n = 0$	$n = \infty$	max	$n = 0$	$n = \infty$
compiler	1962	569	568	438	311	310	4391	2220	1896
texer	133	21	21	47	38	37	232	70	47
softscheme	971	21	19	57	13	13	1706	869	358
similix	865	231	231	100	46	46	1772	814	617
ddd	778	186	186	147	58	58	1518	684	676
em-fun	56	29	32	38	38	38	90	43	4
em-imp	41	25	25	54	54	54	58	16	4
dynamic	154	34	34	1	0	0	239	91	59
graphs	24	13	13	12	7	7	45	20	17
lattice	13	4	4	4	4	4	26	14	4
nbody	114	71	71	20	15	15	157	50	19
tcheck	23	15	15	2	1	1	34	11	8
triang	4	0	0	2	2	2	9	4	4
traverse	34	25	25	8	8	8	36	1	1
takr	101	100	100	0	0	0	103	1	1
takl	4	3	3	0	0	0	6	1	1
tak	2	1	1	0	0	0	4	1	1
puzzle	8	2	2	15	15	15	12	3	3
fft	2	0	0	7	7	7	4	1	1
div-iter	6	5	5	3	3	3	8	1	1
destruct	2	1	1	7	7	7	4	1	1
deriv	3	0	0	1	1	1	4	3	3
dderiv	7	0	0	1	1	1	9	7	7
ctak	3	2	2	0	0	0	9	5	5
cpstak	3	2	2	1	0	0	9	5	1
boyer	19	10	10	0	0	0	21	1	1
browse	7	3	3	11	11	11	8	1	1

Table 7.3: Static measurements of the compiler's ability to optimize the representation of procedures and eliminate argument count checks.

that are not flow analyzed cannot be compiled correctly by the back-end of the compiler. Figure 7.4 reports the optimization of procedure calls. For each benchmark, the number of operations inserted into the code is given. The number of call sites in the benchmark excludes calls that can be trivially detected as calls to library routines or system procedures.

Since the run-time speedups are not a reliable indicator of improvement, and the static counts do not indicate how frequently the inserted operations are executed, Figure 7.5 gives a more precise characterization of the run-time improvements. The figure shows the percentage of optimizable operations eliminated at run-time. For each benchmark, the number of calls made to values other than library routines and top-level values is given. The reductions in operations are then given for the benchmark optimized at differing values

		closure reg loads		closure code references		closure type checks		argument count loads	
benchmark	calls	$n=0$	$n=\infty$	$n=0$	$n=\infty$	$n=0$	$n=\infty$	$n=0$	$n=\infty$
compiler	7022	3989	3987	2015	2010	2015	1718	2015	1723
texer	418	338	338	118	118	118	64	118	64
softscheme	4616	4567	4426	3857	3745	3857	3728	3857	3744
similix	2027	1123	1123	233	219	233	167	233	167
ddd	651	249	249	34	34	34	31	34	31
em-fun	217	63	59	40	38	40	34	40	34
em-imp	146	47	47	27	26	27	22	27	22
dynamic	771	711	711	82	81	82	77	82	77
graphs	77	34	33	14	9	14	5	14	9
lattice	44	37	37	21	20	21	15	21	15
nbody	252	98	98	15	5	15	1	15	1
tcheck	90	38	38	4	4	4	0	4	0
triang	4	4	4	0	0	0	0	0	0
traverse	43	15	15	0	0	0	0	0	0
takr	401	0	0	0	0	0	0	0	0
takl	11	0	0	0	0	0	0	0	0
tak	5	0	0	0	0	0	0	0	0
puzzle	22	21	21	1	1	1	1	1	1
fft	1	1	1	0	0	0	0	0	0
div-iter	6	0	0	0	0	0	0	0	0
destruct	1	0	0	0	0	0	0	0	0
deriv	8	8	8	0	0	0	0	0	0
dderiv	9	1	1	1	1	1	1	1	1
ctak	7	1	1	1	1	1	1	1	1
cpstak	7	1	1	1	1	1	0	1	0
boyer	32	14	14	0	0	0	0	0	0
browse	16	3	3	0	0	0	0	0	0

Table 7.4: Static measurements of the compiler's ability to optimize procedure call sequences on the caller's side. All counts are the number of operations inserted into the code.

of n. In the cpstak benchmark, for example, statistics are reported at four points. The ∞ labeling the last point indicates that the analysis was effectively running at $n = \infty$ at that point. Benchmarks for which increasing values of n had no benefit show statistics for only those points at which the increase in n had a benefit. The statistics measure the percentage reduction in operations over the unoptimized benchmark. The reduction in calls indicates how many calls were converted into loop entry and branches. The other optimizations correspond to those described above.

Increasing the precision of the flow analysis was not helpful in loop recognition and eliminating closure loads, and in eliminating code pointer references it was helpful only for two benchmarks (em-fun and graphs). This is not surprising. The first two optimizations use lexical information together with flow information to justify the transformations, and the extra information collected by the analysis is not helpful. A code pointer reference can be eliminated only when only one procedure can arrive at a call site. This implies that almost all opportunities for optimization will be found at small values for n, since procedures that accept procedures as arguments often accept more than one, and the increasing precision of the analysis for greater values of n will not be helpful.

As expected, improvements in the optimizations as n increased came primarily in the elimination of closure type checks and argument count checks and loads. In some cases the effects were significant. In the cpstak benchmark, for example, letting the analysis run uninhibited allowed the compiler to eliminate all type checks and argument count loads and checks.

The reductions in argument count load and check in some cases seem conflicting. For example, in the deriv benchmark all of the argument count loads were eliminated, but only 43% of the checks were eliminated. This is because some procedures, which were not applied locally, were passed to library routines such as map. These escaping procedures thus required an argument count check.

Several conclusions can be drawn from these two tables of dynamic statistics. First, the flow analysis framework can be instantiated for practical use. At $n = 0$ the compiler is still fast enough for interactive use, and at $n = \infty$ the compile times are not excessive. The analysis is able to justify useful optimizations, and the tunable widening operator is effective at balancing compile-time speed versus optimizations. Higher-order programs benefit when n is increased, but even at low values of n the analysis is useful. From a user-interface perspective, it might be more straightforward for these optimizations to give the user a

benchmark	calls	iters	calls	closure load	closure code ref	closure checks	argument count load	argument count check
compiler	48389965	0	28%	51%	77%	77%	77%	68%
texer	5794071	0	93%	93%	100%	100%	100%	100%
softscheme	18059350	0	0%	0%	10%	10%	10%	12%
		39	0%	0%	10%	11%	10%	94%
similix	16654113	0	12%	58%	82%	82%	82%	25%
		24	12%	58%	89%	95%	95%	30%
ddd	6112550	0	22%	28%	100%	100%	100%	61%
em-fun	162287125	0	41%	75%	76%	76%	76%	83%
		11	41%	76%	77%	78%	78%	98%
		23	41%	76%	77%	78%	78%	100%
em-imp	42598195	0	59%	93%	97%	97%	97%	97%
		11	59%	93%	97%	99%	99%	100%
dynamic	4353290	0	0%	27%	67%	67%	67%	93%
graphs	288730222	0	29%	55%	70%	70%	70%	70%
		9	29%	55%	83%	85%	83%	97%
lattice	154101002	0	26%	28%	55%	55%	55%	55%
		2	26%	28%	56%	60%	60%	61%
nbody	124615459	0	64%	89%	94%	94%	94%	91%
		9	64%	89%	94%	99%	99%	98%
		17	64%	89%	94%	100%	100%	98%
tcheck	11508001	0	0%	79%	93%	93%	93%	91%
		7	0%	79%	93%	100%	100%	98%
triang	11777155	0	51%	51%	100%	100%	100%	100%
traverse	10775691	0	39%	71%	100%	100%	100%	100%
takr	954135	0	0%	100%	100%	100%	100%	100%
takl	2683340	0	0%	100%	100%	100%	100%	100%
tak	1908270	0	0%	100%	100%	100%	100%	100%
puzzle	1691792	0	97%	97%	100%	100%	100%	100%
fft	205182	0	100%	100%	100%	100%	100%	100%
div-iter	1840530	0	99%	100%	100%	100%	100%	100%
destruct	1008672	0	100%	100%	100%	100%	100%	100%
deriv	322007	0	2%	2%	100%	100%	100%	43%
dderiv	462007	0	2%	70%	70%	70%	70%	37%
ctak	333951	0	0%	57%	57%	57%	57%	57%
cpstak	3339510	0	0%	57%	57%	57%	57%	57%
		2	0%	57%	57%	57%	57%	71%
		3	0%	57%	57%	57%	57%	86%
		∞ 5	0%	57%	57%	100%	100%	100%
boyer	5181738	0	0%	1%	100%	100%	100%	100%
browse	296708	0	28%	100%	100%	100%	100%	100%

Table 7.5: Dynamic measurements of operations eliminated compared to the unoptimized benchmarks. Statistics are given as percentage of operations eliminated.

switch selecting between $n = 0$ and $n = \infty$ instead of a dial ranging between $n = 0$ and $n = \infty$. A final decision will require more experience with the compile-time cost of the analysis.

Although it is certain that the *ad hoc* analysis is faster than the more general flow analysis, the effects of the two analyses has not been carefully measured. For the compiler benchmark, the static counts of operations eliminated are almost identical when $n = 0$. This is not surprising, and it is likely that that the results would be similar for the other benchmarks. Dynamic counts were not measured in the old compiler, so it is not possible to quantitatively compare the dynamic effects of the optimizations justified by the two analyses. Again, however, it is likely that the *ad hoc* analysis would be comparable to the new analysis when $n = 0$.

7.5 Summary

In this chapter the implemented flow analysis was used to justify compiler optimizations in a production-quality Scheme compiler. The optimizations included optimizations to procedure calls and closure representations. The analysis was integrated into the compiler by restructuring the front end and replacing an *ad hoc* lexical analysis that was already in place. The optimizations were implemented by adapting and extending the optimizations justified by the *ad hoc* analysis.

The modified compiler was applied to a series of small and large benchmarks, which included large, unmodified software packages such as the *Similix* partial evaluator and the *Chez Scheme* compiler itself. The benchmarks were run using a projection operator that provided a "dial" with which the user could tune the accuracy/speed tradeoffs of the analysis. In particular, at one end of the dial's range the analysis was similar to the *ad hoc* analysis. At the other end the analysis was similar to a 0CFA analysis. The results of the benchmarking demonstrated that the analysis was fast enough for interactive use and useful for justifying optimizations, even when the analysis was relatively inaccurate. The benchmarking also showed that providing a dial to vary the accuracy/speed tradeoffs was useful, but further experimentation will be necessary to know whether a dial ranging between the two extremes or a switch selecting between the two extremes would be more useful to the user.

Chapter 8

Conclusions

In this dissertation we developed a practical and flexible flow analysis framework for higher-order, mostly-functional languages. Although previous work has identified several important aspects of the flow analysis problem, our analysis framework is the first to combine all of the following aspects.

- the accurate treatment of (mutable) data structures,
- the use of type tests to constrain abstract values,
- the use of polyvariance to increase accuracy,
- the use of projection to accelerate convergence to a solution.

The framework is flexible, because it can express a wide variety of analyses that vary in accuracy, speed, and the information they collect. It is parameterized over a polyvariance operator and a projection operator. Different polyvariance operators can be used to regulate the analysis' accuracy, and different projection operators can be used to regulate the analysis' speed. The framework is practical, because its implementation is able to analyze the full Scheme language, is accurate enough to justify useful program transformations, and is fast enough for interactive use, even on large programs.

As opposed to an *ad hoc* formulation, the analysis is formally specified and proved correct. The analysis is specified as an operational semantics. It is proved correct by showing that it is sound with respect to a collecting operational semantics for the language. Storage layout relations is the primary proof technique used to drive the correctness proof.

While the analysis framework is developed using Scheme as the language to be analyzed, the framework developed in this dissertation is useful for analyzing other programming languages as well. For example, while ML has a static type system that would avoid the need for type checks on procedure calls, a flow analysis for ML can be used to further optimize procedure call sequences when the called procedure can be determined by the analysis. As another example, the analysis would be useful for analyzing dynamic object-oriented languages such as Smalltalk to determine at compile time what methods will be invoked from call sites. Furthermore, the fact that object-oriented programs are typically imperative makes the flow-sensitive analysis framework developed in this dissertation a more attractive candidate over flow-insensitive analyses.

8.1 Related work

Shivers [49] and Harrison [32] describe flow analyses for Scheme that are based on abstract interpretation. They differ primarily in the details of the source language and the range of accuracy they can express. Our analysis draws from the advantages of each approach. Like Shivers', we use CPS to make control transfers explicit. Like Harrison's, primitive operations are ordered and bound to temporary variables, and an abstract program state is computed for each basic block of the program. Our analysis subsumes their analyses by expressing a wider range of polyvariance and using projection to accelerate convergence to a fixed point. Also, their analyses are prototypes that do not handle the full language, while our analysis is completely implemented and can analyze arbitrary Scheme programs.

Jagannathan and Weeks [36] describe an analysis for the polyvariant analysis of higher-order applicative programs that also accommodates side-effects and call/cc. Their analysis is parameterized over a polyvariance operator, but it is not parameterized over a projection operator. Also, their characterization of program state is different from ours. In our analysis, the program state is an environment and store. In their analysis, the program state is an environment mapping variables to locations in a global store. Their characterization of program state has consequences on polyvariance and the accuracy of assignment. An assignment to a location pollutes the location's dataflow by adding the assigned value to the dataflow at the point at which the location is bound instead of at the point of assignment. In our analysis, the assigned value replaces the old value of the location in the continuation of the assignment. With respect to polyvariance, their approach limits how program points

can be split, since it is not possible to use the π function to split on arbitrary locations at any time. In both cases, our approach implies that more accurate analyses can be expressed. On the other hand, their polyvariance operator is able to clearly capture polynomial-time polyvariant analyses, while in our formulation, it is not possible to express such analyses.

Jagannathan and Wright [37] apply a polyvariant analysis for the elimination of run-time type checks. Similar to ML's let-bound polymorphism, their polyvariance strategy is to split on locally bound variables. Their strategy is effective, but the analysis can be more expensive than a monovariant analysis, and exponential code duplication in the transformed program is possible. Our application, on the other hand, has explored a different range of the spectrum of analyses. It has demonstrated that monovariant analyses that are less accurate than 0CFA are able to justify useful transformations.

Basing the analysis on a collecting semantics for the language is similar to the approach taken by Young [59]. In his approach, a denotational collecting semantics is defined that collects information about the value of an expression. The static analysis is then expressed as an abstraction of the collecting semantics. Our semantics differs in that both the collecting and abstract semantics are defined operationally.

Some static analyses do not construct a flow graph. Wright [57] describes a soft type system for Scheme that is based on an extension of the Hindley-Milner type system. Heintze [34] describes a set-based analysis for ML that handles assignment and call/cc. For each expression in the input program, the analysis computes a set value that approximates the values to which the expression may evaluate at run time. Both of these approaches are *constraint-based* in that they involve setting up a system of constraints and then solving them. They are attractive because they are easy to understand and can be implemented efficiently. Their limitation is that they specify only the abstract value of an expression and say nothing about the data- or control-flow contexts in which the expression appears, and they are flow insensitive with respect to assignment.

The hybrid algorithm used to implement the store in the abstract machine combines some elements of Flanagan and Felleisen's implementation of set-based analysis [26]. In there implementation data structures are used to represent locations in a global environment that corresponds to our store. Our hybrid algorithm differs by switching representations for assigned variables resulting in a flow-sensitive analysis.

It is interesting to observe that the output of our analysis is similar in form to the output from Heintze's analysis. For each expression, Heintze's analysis generates a context-

free grammar that describes the set value of that expression. The nonterminals of the grammar correspond to constraint variables. Our output is almost exactly the same. For each expression corresponding to a program point, our analysis generates a context-free grammar describing the execution state after evaluating that expression. The nonterminals of the grammar correspond to store locations.

Other researchers have observed that projection operators can be used in practice to reduce the number of iterations needed to stabilize. Yi and Harrison [58] describe a framework for the automatic generation of abstract interpreters. This framework incorporates a notion of projection that is similar to ours. The difference is that they apply projections to values, and we apply them to the entire computation state. Furthermore, they always project a value to the top of the value lattice, while we permit the operator to project a value to any other value above it in the lattice. Bourdoncle [9, 10] uses widening operators in a theoretical setting to accelerate convergence to a solution. A widening operator ∇ is a substitute for the least upper bound operator. The constraint is that given two points x and y, $x \sqcup y \sqsubseteq x \nabla y$. Both of these approaches have the potential to accelerate the analysis without losing too much information. Our work realizes the potential by exhibiting a practical projection operator that still enables useful optimizations.

8.2 Future work

Several aspects of the analysis could be improved or extended. One improvement is to close the gap between the specification of the analysis and its implementation. Since the naïve implementation of the analysis is far too slow for practical use, significant effort was taken to devise an alternative store representation to recover performance. This revised representation meets the formal specification on the live locations at a given program point, but is not completely accurate. One option is to modify the formal specification to incorporate the revised implementation strategy. This may not be worthwhile, since the revised strategy is complex and may hinder correctness proofs.

Another option is to adopt a completely different strategy and switch to a constraint-based specification of the analysis. This would cleanly separate the specification of *what* solution the analysis should compute from *how* that solution is obtained. Devising an appropriate constraint system is not necessarily straightforward, since the analysis is flow

sensitive with respect to assignment, and previous constraint-based flow analyses are flow insensitive.

Another improvement would be to incorporate techniques developed for optimizing polyvariant flow analyses [3]. Such optimizations would complement the improvements made to the naïve implementation of the abstract machine and could substantially improve the performance of analyses that are aggressively polyvariant.

The analysis uses a projection and polyvariance operator to control the accuracy and speed of the analysis. As was mentioned, it might be useful to use local projection and polyvariance operators to provide a finer degree of control over the analysis. Implementing local operators and also developing a family of useful operators would make the analysis more useful in a generic program transformation environment.

Open programs are difficult to analyze, since unknown values can enter the flow graph, and known values can escape the flow graph. A module system would help with this problem, but it is no panacea. First, if the module system permits mutually recursive module definitions, then performing a flow analysis on a per module basis would still result in at least one module with free variables whose abstract values are unknown. Second, the source code for all modules may not be available, and unless module interfaces specify the abstract values of exported bindings, the flow analysis will be forced to assume that the exported bindings have unknown values.

There are two alternatives for addressing the problem of free variables. One alternative is to delay the flow analysis until link time. At link time more of the program is available, reducing the number of free variables in a module. Another alternative is to perform better type recovery at compile time. Although the analysis can recover some type information on conditionals, that is not enough to solve the problem adequately. Better techniques need to be developed to recover flow information, and since the analysis is flow sensitive, it is likely that such techniques can be developed.

A final aspect of the analysis that could be improved is that the implementation is essentially a large interpreter. It should be possible to obtain significant speedups by compiling the analysis. Of course, the benefits of compilation come only when the analysis is run over many iterations, so on a practical basis, the decision to compile would not be automatic and instead would be driven by the selection of the projection operator.

Bibliography

[1] Alfred D. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] J. Michael Ashley and Charles Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, 1994.

[4] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 140–149, 1994.

[5] Hank P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

[6] Anders Bondorf. *Similix Manual, System Version 5.0*. DIKU, University of Copenhagen, Denmark, 1993.

[7] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on Partial Evaluation*, 1993.

[8] Bhaskar Bose. DDD—A transformation system for Digital Design Derivation. Technical Report 331, Indiana University, Computer Science Department, May 1991.

[9] Francois Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–436, October 1992.

[10] Francois Bourdoncle. Efficient chaotic iteration strategies with widening. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer-Verlag, 1993.

[11] Robert G. Burger. The Scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994.

[12] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[13] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme System Manual, Rev. 2.4*, July 1994.

[14] Baudouin Le Charlier, Olivier Degimbe, Laurent Michel, and Pascal Van Hetenryck. Optimization techniques for general purpose fixpoint algorithms: Practical efficiency for the abstract interpretation of Prolog. In *WSA'93, Third International Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 15–26. Springer-Verlag, 1994.

[15] Baudouin Le Charlier and Pascal Van Hentenryck. On the design of generic abstract interpretation frameworks. In *WSA'92, Second International Workshop on Static Analysis*, pages 229–246. Bigre, Irisa, Rennes, France, September 1992.

[16] Chris Clack and Simon Peyton Jones. Strictness analysis: A practical approach. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 35–49. Springer-Verlag, 1985.

[17] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 5(3):1–55, July-September 1991.

[18] Charles Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93*, pages 66–77, 1993.

[19] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93*, pages 145–154, 1993.

[20] Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[21] Patrick Cousot and Rhadia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1978.

[22] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 299–310, 1992.

[23] R. Kent Dybvig. *The Scheme Programming Languages*. Addison-Wesley, 1987.

[24] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 1987.

[25] Cormac Flanagan Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 209–220. ACM, 1995.

[26] Cormac Flanagan and Matthias Felleisen. Well-founded touch optimization for futures. Computer Science Department Technical Report 289, Rice University, PO Box 1892, Houston, TX, 77251-1892, October 1994.

[27] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247. ACM, 1993.

[28] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1991.

[29] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press series in computer systems. MIT Press, 1985.

[30] Joshua D. Guttman, John Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, March 1995.

[31] Joshua D. Guttman, Vipin Swarup, and John Ramsdell. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, March 1995.

[32] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[33] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages*, 12(2):109–121, 1987.

[34] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317, 1994.

[35] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1994.

[36] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 393–407, 1995.

[37] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 854 of *Lecture Notes in Computer Science*, pages 207–224. Springer-Verlag, 1995.

[38] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. Efficient strictness analysis of Haskell. In *SAS'94, Fourth International Symposium on Static Analysis*, volume 854 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, 1994.

[39] Neil Jones and Mads Rosendahl. Higher-order functional graphs. In *Algebraic and Logic Programming, Fourth International Conference, ALP'94*, volume 850 of *Lecture Notes in Computer Science*, pages 242–252. Springer-Verlag, 1994.

[40] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, 1986.

[41] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964.

[42] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In *FPCA'87, 2th International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 426–445. Springer-Verlag, 1989.

[43] Alan Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[44] Flemming Nielson. A denotational framework for flow analysis. *Acta Informatica*, 18:265–287, 1982.

[45] Richard A. O'Keefe. Finite fixed-point problems. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 729–743. MIT Press, May 1987.

[46] Mads Rosendahl. Higher-order chaotic iteration sequences. In *Programming Language Implementation and Logic Programming, Fifth International Symposium, PLILP '93*, volume 714 of *Lecture Notes in Computer Science*, pages 332–345. Springer-Verlag, 1993.

[47] Guillermo Rozas. Taming the Y operator. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 226–234, 1992.

[48] David A. Schmidt. Natural-semantics-based abstract interpretation. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 1 – 18. Springer-Verlag, 1995.

[49] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991. CMU-CS-91-145.

[50] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 47–88. MIT Press, 1991.

[51] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '91*, New Haven, Connecticut, 1991.

[52] Bjarne Steensgaard. A polyvariant closure analysis with dynamic abstraction. Unpublished manuscript, 1994.

[53] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In *SAS'94, Fourth International Symposium on Static Analysis*, volume 854 of *Lecture Notes in Computer Science*, pages 314–328. Springer-Verlag, 1994.

[54] Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, pages 266–275. Prentice-Hall, 1986.

[55] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 435–445, 1994.

[56] Glynn Winskel. *The Formal Semantics of Programming Languages: an Introduction.* MIT Press, 1993.

[57] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, 1994.

[58] Kwangkeun Yi and William Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 246–259. ACM, 1993.

[59] Johnathan Hood Young. *The Theory and Practice: Semantic Program Analysis for Higher-Order Functional Programming Languages.* PhD thesis, Yale University, 1987.

Curriculum Vitae

J. Michael Ashley received a Bachelor of Arts, with High Honors, in Computer Science from Oberlin College in 1990. During the summers of 1989 and 1990 he was a research intern at Los Alamos National Laboratory. From 1990-1994 he was an associate instructor in the Indiana University Computer Science Department. During the summers of 1991 and 1992 he was a research intern at Xerox PARC, and in the summer of 1993 he worked both as a research assistant at Oregon Graduate Institute and as a contractor on a commerical implementation of Scheme. From 1994-1995 he was a research assistant in the Indiana University Computer Science Department. Since 1995 he has been on the faculty of the Electrical Engineering and Computer Science Department at the University of Kansas.