Control-Flow Analysis of Higher-Order Languages
or
Taming Lambda

Olin Shivers

May 1991

CMU-CS-91-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the Office of Naval Research under Contract N00014-84-K-0415. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR or the U.S. Government.

Keywords: data-flow analysis, Scheme, LISP, ML, CPS, type recovery, higher-order functions, functional programming, optimising compilers, denotational semantics, non-standard abstract semantic interpretations

Abstract

Programs written in powerful, higher-order languages like Scheme, ML, and Common Lisp should run as fast as their FORTRAN and C counterparts. They should, but they don't. A major reason is the level of optimisation applied to these two classes of languages. Many FORTRAN and C compilers employ an arsenal of sophisticated global optimisations that depend upon data-flow analysis: common-subexpression elimination, loop-invariant detection, induction-variable elimination, and many, many more. Compilers for higher-order languages do not provide these optimisations. Without them, Scheme, LISP and ML compilers are doomed to produce code that runs slower than their FORTRAN and C counterparts.

The problem is the lack of an explicit control-flow graph at compile time, something which traditional data-flow analysis techniques require. In this dissertation, I present a technique for recovering the control-flow graph of a Scheme program at compile time. I give examples of how this information can be used to perform several data-flow analysis optimisations, including copy propagation, induction-variable elimination, useless-variable elimination, and type recovery.

The analysis is defined in terms of a non-standard semantic interpretation. The denotational semantics is carefully developed, and several theorems establishing the correctness of the semantics and the implementing algorithms are proven.

To my parents, Julia and Olin.

Contents

Acknowledgments

It is a great pleasure to thank the people who helped me with my graduate career.

First, I thank the members of my thesis committee: Peter Lee, Allen Newell, John Reynolds, Jeannette Wing, and Paul Hudak.

- Peter Lee, my thesis advisor and good friend, taught me how to do research, taught me how to write, helped me publish papers, and kept me up-to-date on the latest developments in Formula-1 Grand Prix. I suppose that I recursively owe a debt to Uwe Pleban for teaching Peter all the things he taught me.

- Allen Newell was my research advisor for my first four years at CMU, and co-advised me with Peter after I picked up my thesis topic. He taught me about having "a bright blue flame for science." Allen sheltered me during the period when I was drifting in research limbo; I'm reasonably certain that had I chosen a different advisor, I'd have become an ex-Ph.D.-candidate pretty rapidly. Allen believed in me — certainly more than I did.

- John Reynolds taught (drilled/hammered/beat into) me the importance of formal rigour in my work. Doing mathematics with John is an exhilarating experience, if you have lots of stamina and don't mind chalk dust or feelings of inadequacy. John's emphasis on formal methods rescued my dissertation from being just another pile of hacks.

- Jeannette Wing was my contact with the programming-systems world while I was officially an AI student. She supported my control-flow analysis research in its early stages. She also has an eagle eye for poor writing and a deep appreciation for edged weapons.

- Paul Hudak's work influenced me a great deal. His papers on abstract semantic interpretations caused me to consider applying the whole approach to my problem.

It goes without saying that all of these five scholars are dedicated researchers, else they would not have attained their current positions. However—and this is not a given in academe—they also have a deep committment to their students' welfare and development. I was very, very lucky to be trained in the craft of science under the tutelage of these individuals.

In the summer of 1984, Forest Baskett hired the entire T group to come to DEC's Western Research Laboratory and write a Scheme compiler. I was going to do the data-flow analysis module of the compiler; it would be fair to say that I am now finally finishing my 1984 summer job. The compiler Forest hired us to write turned into the ORBIT Scheme compiler, and this dissertation is the third one to come out of that summer project (so far). Both Forest and DEC have a record of sponsoring grad students for summer research projects that are interesting longshots, and I'd like to acknowledge their foresight, scope and generosity.

If one is going to do research on advanced programming languages, it certainly helps to be friends with someone like Jonathan Rees. In 1982, Jonathan was the other guy at Yale who'd read the Sussman/Steele papers. We agreed that hacking Scheme and spinning great plans was much more fun than interminable theory classes. Jonathan's deep understanding of computer languages and good systems taste has influenced me repeatedly.

Norman Adams, like Jonathan a member of the T project, contributed his technical and moral support to my cause. The day you need your complex, forty-page semantics paper reviewed and proofread in forty-eight hours is the day you really think hard about who your friends are.

David Dill has an amazing ability. He will listen to you describe a problem he's never thought about before. Then, after about twenty minutes, he will make a two-sentence observation; it will cut right to the heart of your problem. (Stanford grad students looking for an advisor should note this fact carefully.) David patiently listened to me ramble on about doing data-flow analysis in Scheme long before I had the confidence to discuss it with anyone else. Our discussions helped me to overcome early discouragement and his suggestions led, as they usually do, straight toward the eventual solution.

CMU Computer Science is well known for its cooperative, friendly environment. When I was deep into the formal construction of the analyses in some precursor work to this dissertation, I got completely hung up on a tricky, non-standard recursive domain construction. The prospect of having to do my very own inverse-limit construction was not a thrilling one. One Saturday morning, the phone rang about three hours after I had gone to sleep. It was Dana Scott, who proceeded to read to me over the phone a four-line solution to my problem. John Reynolds had described my problem to him in the hall on Friday; Dana had taken it home and solved it. This episode is typical of the close support and cooperation that has made my time here as a graduate student so pleasant and intellectually fulfilling.

Alan Perlis, Josh Fisher, and John Ellis were instrumental in teaching me the art of programming at Yale and persuaded me by example that getting a Ph.D. in computer science was obviously the most interesting and fun thing one could possibly do after college. Mary-Claire van Leunen, also at Yale, did what she could to improve my writing style. Professors Ferdinand Levy and Hyland Chen were also, by way of example, large influences on my decision to pursue a research degree. Perlis and Levy, in particular, sold me on the idea of coming to CMU.

The core of my knowledge of denotational semantics comes from a grad student class given by Stephen Brookes. His lecture notes, which he is currently converting into a book, were a clear and painless introduction to the field. When I began work on the formal semantics part of this dissertation, I repeatedly returned to them.

Heather, Lin, Olga, and Wenling provided emotional first-aid and kept me glued together during black times. Mark Shirley, my hacking partner and good friend, gets credit for handing me a set of Sussman and Steele's papers in 1980, which, it could be argued, triggered off the next decade of my life. Had Manuel Hernandez and Holt Sanders not been with me on the deck of the *Jing Lian Hao* during a typhoon in the South China Sea in the summer of 1986, I would not have *had* a next decade of my life, and this dissertation, obviously, would never have been written. Small decisions...

Having the office one down from Hans Moravec's for five years is bound to warp anybody, particularly if you share the same circadian cycle.[1] Late-night exposure to Hans, a genuine Mad Scientist, taught me about relativistic Star Wars, skyhooks, neural development in chordates, life in the sun, methods for destroying the universe, robot pals, the therapeutic value of electrical shock, material properties of magnetic monopoles, downloading your software, and the tyranny of the 24-hour cycle. Generally speaking, Hans kept my mind loosened up.

It must be said that my officemates, Rich Wallace, Hans Tallis, Derek Beatty, Allan Heydon, and Hank Wan, are very patient people. Particularly with respect to phone bills.

Finally, of course, I want to thank my family, particularly my parents, Julia and Olin; my sisters, Julia and Mary; and my uncle, Laurence. It has helped a great deal knowing that their unstinting love and support was always there.

[1] *I.e.*, the rotational period of Mars.

Chapter 1

Introduction

1.1 My Thesis

Control-flow analysis is feasible and useful for higher-order languages.

Higher-order programming languages, such as Scheme and ML, are more difficult to implement efficiently than more traditional languages such as C or FORTRAN. Complex, state-of-the-art Scheme compilers can produce code that is roughly as efficient as that produced by simple, non-optimising C compilers [Kranz$^+$ 86]. However, Scheme and ML compilers still cannot compete successfully with complex, state-of-the-art C and FORTRAN compilers.

A major reason for this gap is the level of optimisation applied to these two classes of languages. Many FORTRAN and C compilers employ an arsenal of sophisticated global optimisations that depend upon data-flow analysis: common-subexpression elimination, loop-invariant detection, induction-variable elimination, and many, many more. Scheme and ML compilers do not provide these optimisations. Without them, these compilers are doomed to produce code that runs slower than their FORTRAN and C counterparts.

One key problem is the lack of an explicit control-flow graph at compile time, something traditional data-flow analysis techniques require. In this dissertation, I develop an analysis for recovering the control-flow graph from a Scheme program at compile time. This analysis uses an intermediate representation called CPS, or continuation-passing style, and a general analysis framework called non-standard abstract semantic interpretation.

After developing the analysis, I present proofs of its correctness, give algorithms for computing it, and discuss implementation tricks that can be applied to the algorithms. This establishes the first half of my thesis: control-flow analysis is feasible for higher-order languages.

The whole point of control-flow analysis is that it enables important data-flow program optimisations. To demonstrate this, I use the results of the control-flow analysis to develop several of these classical data-flow analyses and optimisations for higher-order languages. This establishes the second half of my thesis: control-flow analysis is useful for higher-order languages.

1.2 Structure of the Dissertation

This dissertation has the following structure:

- The rest of this chapter discusses data-flow analysis, higher-order languages, and why it is difficult to do the one in the other.
- The following chapter (2) introduces the two basic tools we'll need to develop the analysis: the CPS intermediate representation, and the technique of non-standard abstract semantic interpretation.
- Next we develop the basic control-flow analysis. Chapter 3 presents the analysis; this is the core of the dissertation. Chapter 4 contains proofs that establish the analysis' correctness and computability.
- After developing the analysis as a mathematical function, we turn to algorithms for computing this function. Chapter 5 develops the algorithms, and discusses some details of an implementation that I wrote. Chapter 6 contains proofs showing that the algorithms compute the analysis function.
- Chapter 7 develops three applications using control-flow analysis: induction-variable elimination, useless-variable elimination, and constant propagation.
- Control-flow analysis has limitations. Chapter 8 discusses these limits, and presents an extension (reflow analysis) that can be used to exceed them. Chapters 9 and 10 discuss four more applications that rely on reflow analysis: type recovery, future analysis, copy propagation, and lambda propagation.
- The dissertation proper closes with chapters on future directions, related work, and conclusions (chapters 11–13).
- This is followed by two appendices, one containing a code listing for a control-flow analysis implementation, and one giving the results of applying this program to some simple Scheme procedures.

1.2.1 Skipping the Math

This dissertation has a fair amount of mathematics in it, mostly denotational semantics. In spite of this, I have made a particular effort to make it comprehensible to the reader who prefers writing programs to proving theorems. Most of the serious mathematics is carefully segregated into distinct chapters; each of these chapters begins with a half-page summary that states the chapter's main results. Feel free to read the summary, take the results on faith, and skip the math.

1.3 Flow Analysis and Higher-Order Languages

1.3.1 Flow Analysis

Flow analysis is a traditional optimising compiler technique for determining useful information about a program at compile time. Flow analysis determines path-invariant facts about textual points in a program. A flow analysis problem is a question of the form:

> "What is true at a given textual point p in my program, independent of the execution path taken to p from the start of the program?"

Example domains of interest might be the following:

- Range analysis: What range of values is a given reference to an integer variable constrained to lie within? Range analysis can be used, for instance, to do array bounds checking at compile time.

- Loop invariant detection: Do all possible prior assignments to a given variable reference lie outside its containing loop?

Over the last thirty years, standard techniques have been developed [Aho+ 86, Hecht 77] to answer these questions for the standard imperative languages (*e.g.*, Pascal, C, Ada, Bliss, and chiefly FORTRAN). Flow analysis is perhaps the chief tool in the optimising compiler writer's bag of tricks; figure 1.1 gives a partial list of the code optimisations that can be performed using flow analysis.

1.3.2 Higher-Order Languages

A higher-order programming language (HOL) is one that permits higher-order procedures — procedures that can take other procedures as arguments and return them as results. More specifically, I am interested in the class of *lexically scoped*, *applicative order* languages that allow *side-effects* and provide *procedures as first-class data*. Members of this class are Scheme [Rees+ 86], ML [Milner 85, Milner+ 90], and Common Lisp [Steele 90]. Granting "first class" status to procedures means allowing them to be manipulated with the same freedom as other primitive data types, *e.g.*, passing them as arguments to procedures, returning them as values, and storing them into data structures.

In this dissertation, I shall use Scheme as the exemplar of higher-order languages, and only occasionally refer to other languages. Scheme is a small, simple language, which nonetheless has the full set of features that conspire to make flow analysis so difficult: higher-order procedures, side effects, and compound data structures. ML's principal semantic distinction from Scheme, its type system, does not fundamentally affect the task of flow analysis. The size and complexity of Common Lisp make it difficult to work with, but do not introduce any interesting new semantic features that change the flow analysis problem.

- global register allocation
- global common-subexpression elimination
- loop-invariant detection
- redundant-assignment detection
- dead-code elimination
- constant propagation
- range analysis
- code hoisting
- induction-variable elimination
- copy propagation
- live-variable analysis
- loop unrolling
- loop jamming
- type inference

Figure 1.1: Some flow-analysis optimisations.

1.3.3 The Problem

Traditional flow-analysis techniques have never successfully been applied to HOLs. These languages are sufficiently different from the traditional imperative languages that the classical techniques developed for them are not applicable. HOLs can be contrasted with the traditional imperative languages in the following ways:

- **Binding versus assignment:**
  Both classes of language have the same two mechanisms for associating values with variables: parameter binding and variable assignment. However, there are differences in frequency of usage. The traditional imperative languages tend to encourage the use of assignment statements; higher-order languages tend to encourage binding.

- **Procedures as first-class citizens:**
  Procedures in higher-order languages are data that can be passed as arguments to procedures, returned as values from procedure calls, stored into arrays, and so forth.

Traditional flow-analysis techniques concentrate on tracking assignment statements. Thus, the HOL emphasis on variable binding changes the complexion of the problem. Generally, however, variable binding is a simpler, weaker operation than the extremely powerful operation of side-effecting assignments. Analysis of binding is a more tractable problem than analysis of assignments because the semantics of binding are simpler and there are more invariants for program-analysis tools to invoke. In particular, the invariants of the λ-calculus are usually applicable to HOLs.

On the other hand, the higher-order, first-class nature of HOL procedures can hinder efforts to derive even basic control-flow information. I claim that it is this aspect of higher-order languages which is chiefly responsible for the absence of flow analysis from their compilers to date. A brief discussion of traditional flow analysis techniques will show why this is so.

Consider the following piece of Pascal code:

```
FOR i := 0 to 30 DO BEGIN
    s := a[i];
    IF s < 0 THEN
        a[i] := (s+4)^2
    ELSE
        a[i] := cos(s+4);
    b[i] := s+4;
    END
```

Traditional flow analysis requires construction of a *control-flow graph* for the code fragment (fig. 1.2).

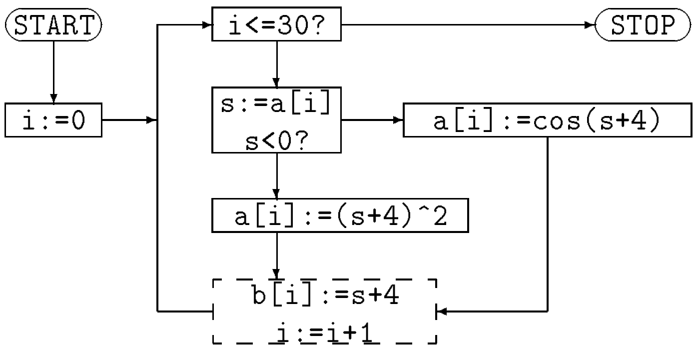


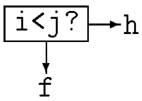
Figure 1.2: Control-flow graph

Every vertex in the graph represents a *basic block* of code: a sequence of instructions such that the only branches into the block are branches to the beginning of the block, and the only branches from the block occur at the end of the block. The edges in the graph represent possible transfers of control between basic blocks. Having constructed the control-flow graph, we can use graph algorithms to determine path invariant facts about the vertices.

In this example, for instance, we can determine that on all control paths from START to the dashed block (`b[i]:=s+4; i:=i+1`), the expression `s+4` is evaluated with no subsequent assignments to `s`. Hence, by caching the result of `s+4` in a temporary, we can eliminate the redundant addition in `b[i]:=s+4`. We obtain this information through consideration of the paths through the control-flow graph.

The problem with HOLs is that *there is no static control-flow graph at compile time.* Consider the following fragment of Scheme code:

```
(let ((f (foo 7 g k))
      (h (aref a7 i j)))
  (if (< i j) (h 30) (f h)))
```

Consider the control flow of the `if` expression. Its graph is: 

After evaluating the conditional's predicate, control can transfer either to the procedure that is the value of `h`, or to the procedure that is the value of `f`. But what's the value of `f`? What's the value of `h`? Unhappily, they are computed at run time.

If we knew all the procedures that `h` and `f` could possibly be bound to, independent of program execution, we could build a control-flow graph for the code fragment. So, if we wish to have a control-flow graph for a piece of Scheme code, we need to answer the following question: for every procedure call in the program, what are the possible lambda expressions that call could be a jump to? But this is a flow analysis question! So with regard to flow analysis in an HOL, we are faced with the following unfortunate situation:

- In order to do flow analysis, we need a control-flow graph.
- In order to determine control-flow graphs, we need to do flow analysis.

Chapter 2

Basic Techniques: CPS and NSAS

2.1 CPS

The fox knows many things, but the hedgehog knows one great thing.
— Archilocus

The first step towards finding a solution to this conundrum is to develop a representation for our programs suitably adapted to the analysis we're trying to perform.

In Scheme and ML, we must represent and deal with transfers of control caused by procedure calls. In the interests of simplicity, then, we adopt a representation where *all* transfers of control — sequencing, looping, procedure call/return, conditional branching — are represented with the same mechanism: the tail-recursive procedure call. This type of representation is called Continuation-Passing Style or *CPS*.

Our intermediate representation, CPS Scheme, stands in contrast to the intermediate representation languages commonly chosen for traditional optimising compilers. These languages are conventionally some form of slightly cleaned-up assembly language: quads, three-address code, or triples. The disadvantage of such representations is their *ad hoc*, machine-specific, and low-level semantics. The advantages of CPS Scheme lie in its appeal to the formal semantics of the λ-calculus and its representational simplicity.

CPS conversion can be referred to as the "hedgehog" approach, after a quotation by Archilocus. All control and environment structures are represented in CPS by lambda expressions and their application. After CPS conversion, the compiler need only know "one great thing" — how to compile lambda expressions very well. This approach has an interesting effect on the pragmatics of Scheme compilers. Basing the compiler on lambda expressions makes lambda expressions very cheap, and encourages the programmer to use them explicitly in his code. Since lambda is a very powerful construct, this is a considerable boon for the programmer.

2.1.1 Definition of CPS Scheme

CPS can be summarised by stating that procedure calls in CPS are one-way transfers — they do not return. So a procedure call can be viewed as a `GOTO` that passes values. If we are interested in the value computed by a procedure f of two values, we must also pass to f a third argument, the *continuation*. The continuation is a procedure; after f computes its value v, instead of "returning" the value v, it calls the continuation on v. Thus the continuation represents the control point to which control should transfer after the execution of f.

For example, if we wish to print the value computed by $(x + y) * (z - w)$, we do not write:

```
(print (* (+ x y) (- z w)))
```

Instead, we write:

```
(+ x y (λ (xy)
         (- z w (λ (zw)
                  (* xy zw (λ (prod) (print prod *c*)))))))
```

Here, the original procedures — `+`, `-`, `*`, and `print` — are all redefined to take a continuation as an extra argument. The `+` procedure calls its third argument, `(λ (xy) ...)`, on the sum of its first two arguments, `x` and `y`. Likewise, the `-` procedure calls its third argument, `(λ (zw) ...)`, on the difference of its first two arguments, `z` and `w`. The `*` procedure calls its third argument on the product of its first two arguments, $x + y$ and $z - w$. This product is finally passed to the `print` procedure, along with some top-level continuation `*c*`. The `print` procedure prints out the value of `prod` and then calls the `*c*` continuation.

Standard non-CPS Scheme can be easily transformed into an equivalent CPS program, so this representation carries no loss of generality. Once committed to CPS, we can make further simplifications:

- **No special syntactic forms for conditional branch**
  The semantics of primitive conditional branch is captured by the primitive procedure `if` which takes one boolean argument, and two continuation arguments: `(if b c a)`. If the boolean `b` is true, the true, or "consequent," continuation `c` is called; otherwise, the false, or "alternate," continuation `a` is called.

  We can also assume a class of `test` primitives that perform various basic conditional tests on their first argument. For example, `(test-integer x c a)` branches to continuation `c` if `x` is an integer, otherwise to continuation `a`.

- **No side effects to variables**
  Side effects are allowed to data structures only. That is, the programmer is allowed to alter the contents of data structures, with operations like `rplaca` and `vector-set!`, but he is not allowed to alter the bindings of variables — there is no `set!` syntax. This means that all side-effects are handled by primitive operations, and special syntax is not required.

- Lambda expressions: (λ ($v_1 \ldots v_n$) *call*)
  where $n \geq 0$.
- Variable references: `foo`, `bar`, ...
- Constants: `3`, `"doghen"`, `'(a 3 elt list)`, ...
- Primitive operations: `+`, `if`, `rplaca`, ...
- Simple calls: (*fun* arg^*)
  where *fun* is a lambda, var, or primop, and the *arg*s are lambdas, vars, or constants.
- `letrec` calls: (`letrec` ((f_1 l_1)...) *call*)
  where the f_i are variables, and the l_i are lambda expressions.

Figure 2.1: CPS Scheme language grammar

Programs that use variable assignment can be automatically converted to assignment-free programs by inserting extra mutable data structures into the program. An expression that looks like

```
(λ (x) ... (set! x y) ... x ...)
```

is converted to

```
(λ (x')
  (let ((x (make-cell x')))
    ... (set-cell x y) ... (contents x) ...))
```

This process is called "assignment conversion" [Kranz+ 86].

- **No `call/cc` operator**
  Languages like Scheme and ML often have the troublesome `call-with-current-continuation` or `call/cc` operator. CPS Scheme doesn't need the operator. When translating programs into their CPS Scheme representations, every occurrence of `call/cc` can be replaced with its CPS definition:

```
(λ (f k) (f (λ (v k0) (k v)) k))
```

Scheme code violating any of these restrictions is easily mapped into equivalent Scheme code preserving them, so they also carry no loss of generality. The details of the CPS transformation are treated in a number of papers [Kelsey 89, Kranz 88, Kranz+ 86, Steele 76]. These restrictions leave us with a fairly simple language to deal with: There are only five syntactic classes: lambdas, variables, constants, primitive operations (or *primops*) and calls (fig. 2.1). Note that primops are not first-class citizens — in this grammar, they may only be called, not used as arguments, or passed around as data. This is not a problem, since

```
(λ (n) (letrec ((lp (λ (i sum) (if (zero? i) sum
                                   (lp (- i 1) (+ i sum))))))
         (lp n 0)))


(λ (n k)
  (letrec ((lp (λ (i sum c)
                 (test-zero i
                   (λ () (c sum))
                   (λ ()
                     (- i 1 (λ (i1)
                        (+ sum i (λ (sum1)
                           (lp i1 sum1 c))))))))))
    (lp n 0 k)))
```

Figure 2.2: Standard and CPS Scheme to sum 1 through n

everywhere the + primop appears as data, for instance, we can replace it with an equivalent lambda expression: `(λ (a b c) (+ a b c))`.

Figure 2.2 shows a procedure that sums the first n integers in both standard Scheme and its CPS Scheme representation. It bears repeating that this extremely simple language is a practical and useful intermediate representation for languages such as Scheme and ML. In fact, the dialect we are using here is essentially identical to the one used by the optimising Scheme compiler ORBIT[Kranz+ 86].

2.1.2 History

Continuation-passing style is not a new idea, dating back to at least the mid-Sixties [Wijngaarden 66]. Reynolds [Reynolds 72] pointed out the semantically constraining features of CPS as a language for describing language interpreters.

The central challenge of a Scheme compiler is dealing with lambda. The extreme simplicity and power of Scheme comes from the power of the lambda operator. Programmers capitalise on this power: typical Scheme programming style uses lambdas frequently, with the assumption that they will be implemented cheaply.

CPS-based compilers raise the stakes even further. Since all control and environment structures are represented in CPS Scheme by lambda expressions and their application, after CPS conversion, lambdas become not merely common, but ubiquitous. The compiler lives and dies by its ability to handle lambdas well. In short, the compiler has made an explicit commitment to compile lambda cheaply — which further encourages the programmer to use them explicitly in his code.

Implementing such a powerful operator efficiently, however, is quite tricky. Thus, research in Scheme compilation has had, from the inception of the language, the theme of taming lambda: using compile-time analysis techniques to recognise and optimise the lambda expressions that do not need the fully-general closure implementation.

This thesis extends a thread of research on CPS-based compilers that originates with the work of Steele and Sussman in the mid-seventies. CPS and its merits are treated at length by Steele and Sussman. Their 1976 paper, "Lambda: The Ultimate Declarative" [Steele 76], first introduced the possibility of using CPS as an intermediate representation for a Scheme compiler. Steele designed and implemented such a compiler, Rabbit, for his Master's thesis [Steele 78]. Rabbit demonstrated that Scheme could be compiled efficiently, and showed the effectiveness of CPS-based compiler technology. Rabbit introduced the idea of code analysis to aid the compiler in translating lambdas into efficient code.

The next major development in CPS-based compilation was the ORBIT Scheme compiler [Kranz$^+$ 86], designed and implemented by the T group [Rees$^+$ 82] at Yale. ORBIT was a strong demonstration of the power of CPS-based compilers: it produced code that was competitive with non-optimising C and Pascal compilers. ORBIT, again, relied on extensive analysis of the CPS code tree to separate the "easy" lambdas, which could be compiled into efficient code, from the lambdas that needed general but less-efficient run-time realisations. The most detailed explication of these analyses is found in Kranz's dissertation [Kranz 88]. ORBIT is the most highly optimising Scheme compiler extant.

Kelsey, also a member of the T group, has designed a compiler, TC, whose middle and back ends employ a CPS intermediate representation [Kelsey 89]. He has front ends which translate BASIC, Scheme and Pascal into CPS.

A team at Princeton and Bell Labs, led by Andrew Appel and David MacQueen, has applied the Orbit technology to ML with good results [Appel$^+$ 89]. Their CPS-based compiler is currently the standard compiler for most ML programming in the U.S.

CPS is now an accepted representation for advanced language compilers. None of these compilers, however, has made a significant attempt to perform control- or data-flow analysis on the CPS form. This thesis, then, is an attempt to push CPS-based compilers to the next stage of sophistication.

2.2 Control Flow in CPS

The point of using CPS Scheme for an intermediate representation is that all transfers of control are represented by procedure call. This gives us a uniform framework in which to define the control-flow analysis problem. In the CPS Scheme framework, the control-flow problem is defined as follows:

> For each call site c in program P, find a set $L(c)$ containing all the lambda expressions that could be called at c. *I.e.*, if there is a possible execution of P such that lambda l is called at call site c, then l must be an element of $L(c)$.

That is, control-flow analysis in CPS Scheme consists of determining what call sites call which lambdas.

This definition of the problem does not define a unique function L. The trivial solution is the function $L(c) =$ AllLambdas, *i.e.*, the conclusion that all lambdas can be reached from any call site. What we want is the tightest possible L that we can derive at compile time with reasonable computational expense.

2.3 Non-Standard Abstract Semantic Interpretation

2.3.1 NSAS

Casting our problem into this CPS-based framework gives us a structure to work with; we now need a technique for analysing that structure. The method of non-standard abstract semantics (NSAS) is an elegant technique for formally describing program analyses. It forms the tool we'll use to solve our control-flow problem as described in the previous section.

Suppose we have programs written in some programming language L, and we wish to determine some property X about our programs at compile time. For example, X might be the property "the set of variables in a program whose values may be stack-allocated instead of heap-allocated." There is a three-step process we can use in the NSAS framework to construct a computable analysis for X:

1. We start with a standard denotational semantics S for our language. This gives us a precise definition of what the programs written in our language mean.

2. Then, we develop a *non-standard semantics* S_X for L that precisely expresses property X. We typically derive this non-standard semantics from our original standard semantics S. So, whereas semantics S might say the meaning of a program is a function mapping the program's inputs to its results, semantics S_X would say the meaning of a program is a function mapping the program's inputs to the property X. The point of this semantics is that it constitutes a precise, formal definition of the property we want to analyse.

3. S_X is a precise definition of the property we wish to determine, but its precision typically implies that it cannot be computed at compile time. It might be uncomputable; it might also depend on the run-time inputs. The final step, then, is to *abstract* S_X to a new semantics, $\widehat{S}_X$ which trades accuracy for compile-time computability. This sort of approximation is a typical program-analysis tradeoff — the real answers we seek are uncomputable, so we settle for computable, conservative approximations to them.

When we are done, we have a computable analysis $\widehat{S}_X$, and two other semantics that link it back to the real, standard semantics for our language.

The method of abstract semantic interpretation has several benefits. Since an NSAS-based analysis is expressed in terms of a formal semantics, it is possible to prove important properties about it. In particular, we can prove that the non-standard semantics S_X correctly expresses properties of the standard semantics S, and that the abstract semantics $\widehat{S}_X$ is computable and safe with respect to S_X. Further, due to its formal nature, and because of its relation to the standard semantics of a programming language, simply expressing an analysis in terms of abstract semantic interpretations helps to clarify it.

NSAS-based analyses have been developed for an array of program optimisations. Typical examples are strictness analysis of normal-order languages [Bloss$^+$ 86] and static reclamation of dynamic data-structures ("compile-time gc") [Hudak 86b].

2.3.2 Don't Panic

The reader who is more comfortable with computer languages than denotational semantics equations should not despair. The semantic equations presented in this dissertation can quite easily be regarded as interpreters in a functional disguise. The important point is that these "interpreters" do not compute a program's actual value, but some other property of the program (in our case, the call graph for the program). We will compute this call graph with a non-standard, abstract "interpreter" that abstractly executes the program, collecting information as it goes.

2.3.3 NSAS Example

The traditional toy example [Cousot 77, Mycroft 81] of an NSAS-based analysis uses the arithmetic rule of signs. The rule of signs is simply the property of arithmetic that tells us the sign of simple arithmetic operations based on the sign of their operands: a positive number times a negative number is negative, a negative number minus a positive number is negative, and so forth.

Consider a simple language that allows us to add, subtract, and multiply integer constants. A typical expression in our language might be

```
-3 * (3 + (-4 - 2))
```

Suppose we're interested in determining the sign of an expression without having to actually compute its full value.

- Our standard semantics S maps expressions to their arithmetic values:

  $$S[\![\texttt{-3 * (3 + (-4 - 2))}]\!] = 9.$$

  Without going to the trouble to write out a recursive definition of S, suffice it to say that S specifies the necessary calculations to compute the value of an expression.

- Our sign-analysis semantics S_{sign} maps an expression to one of the symbols $\{-, 0, +\}$, depending on the sign of the result. The exact semantics would perform the arithmetic calculations, actually calculating the arithmetic value of the expression, and then producing one of $+$, 0, or $-$, depending on the sign of the result:

  $$S_{sign}[\![\texttt{-3 * (3 + (-4 - 2))}]\!] = +$$

  This precisely defines the sign of any expression written in our language.

- Our abstract sign semantics $\widehat{S}_{sign}$ performs the computation in an abstract domain using the rule of signs. Instead of calculating with integer values, it only retains the possible signs of the intermediate calculations. The required additions, subtractions and multiplications are performed on these abstract values. So $\widehat{S}_{sign}$ will add $+$ to $+$ and get $+$; multiply $+$ by $-$ and get $-$, and so forth. Unfortunately, this abstract calculation can result in loss of information. For example, if $\widehat{S}_{sign}$ adds a $+$ to a $-$, the result value can have any sign. In this case, $\widehat{S}_{sign}$ must produce the value $\{-, 0, +\}$. Thus, while S_{sign} computes with the integers $\mathcal{Z}$, $\widehat{S}_{sign}$ computes with sets of signs from $\mathcal{P}(\{-, 0, +\})$. So

$$\begin{aligned} \widehat{S}_{sign}[\![\texttt{3 * (-4 - 2)}]\!] &= \{+\} * (\{-\} - \{+\}) \\ &= \{+\} * \{-\} \\ &= \{-\} \end{aligned}$$

  but

$$\begin{aligned} \widehat{S}_{sign}[\![\texttt{3 + (-4 - 2)}]\!] &= \{+\} + (\{-\} - \{+\}) \\ &= \{+\} + \{-\} \\ &= \{-, 0, +\}\,. \end{aligned}$$

In the second example, $\widehat{S}_{sign}$ produced a result that was correct (after all, it is certainly true that the sign of -3 is either $+$, 0, or $-$), but less precise than the exact semantics S_{sign}. This is an important property: the abstract semantics must approximate the exact semantics in a conservative way.

Because we are expressing our analysis with formal semantics, this conservative-approximation property can be formally expressed and verified. If we were to write out complete definitions for S_{sign} and $\widehat{S}_{sign}$, for example, we could prove fairly easily that

$$\forall exp, \quad S_{sign}(exp) \in \widehat{S}_{sign}(exp).$$

That is, for any given expression, $\widehat{S}_{sign}$ returns a set containing at least the true sign of the expression, and possibly others. This is the sort of conservative-approximation property which is important to show in general for our abstract analyses.

This simple sign-analysis example shows how a non-standard semantics can express the ideal analysis we want, and how we can abstract our non-standard analysis to arrive at a computationally simpler analysis that is a safe approximation to the exact analysis.

Chapter 3

Control Flow I: Informal Semantics

Interesting if true — and interesting anyway.
— M. Twain

We can develop a computable control-flow analysis technique by following the NSAS three-step construction process:

1. We start with a standard semantics for CPS Scheme.

2. From the standard semantics, we develop a non-standard semantics that precisely expresses control-flow analysis. This semantics will be uncomputable.

3. We abstract this semantics to develop a computable approximation.

The emphasis of this chapter will be on an intuitive development of the main ideas behind the analysis. The mathematical machinery will be kept in check, and proofs will be avoided altogether. In fact, this initial treatment will view a denotational semantics as a kind of Scheme interpreter written in a functional language. A reader familiar with Scheme but not with denotational semantics should be able to follow the development without too much trouble. In the next chapter, we will delve into the mathematical fine detail and the proofs of correctness.

In other words, this chapter presents all the interesting ideas; in the next chapter, we'll prove that our intuitions about them are, in fact, true.

3.1 Notation

D^* is used to indicate all vectors of finite length over the set D. Functions are updated with brackets: $e[a \mapsto b,\ c \mapsto d]$ is the function mapping a to b, c to d, and everywhere else identical to function e. An update standing by itself is taken to be an update to the bottom function that is everywhere undefined, so $[a \mapsto b,\ c \mapsto d] = \bot[a \mapsto b,\ c \mapsto d]$, and $[\,] = \bot$. Vectors are written $\langle x_1, \ldots, x_n \rangle$, and are concatenated with the $\S$ operator:

$v_1 \S v_2$. The ith element of vector v is written $v{\downarrow}i$. The power set of A is $\mathcal{P}(A)$. Function application is written with juxtaposition: $f\,x$. We extend a lattice's meet and join operations to functions into the lattice in a pointwise fashion, *e.g.*: $f \sqcap g = \lambda x.\ (f\ x) \sqcap (g\ x)$. The "predomain" operator $+$ is used to construct the disjoint union of two sets: $A + B$. This operator does *not* introduce a new bottom element, and so the result structure is just a set, not a domain.

A half-arrow $f\colon A \rightharpoonup B$ indicates a partial function. The domain of a partial function f is written $\mathit{Dom}(f)$. The image of a function f is written $\mathit{Im}(f)$. So if g is the partial function $[1 \mapsto 2,\ 3 \mapsto 4,\ 17 \mapsto 0]$, then $\mathit{Dom}(g) = \{1, 3, 17\}$, and $\mathit{Im}(g) = \{2, 4, 0\}$.

In this dissertation, I use the term *domain* to mean a pointed, chain-complete, partially-ordered set, or *cpo*. Continuous functions on cpo's have least fixed points. The least fixed point of such a function f is written fix f, where fix $f = \bigsqcup_i f^i \bot$.

When bits of program syntax appears as constants in mathematical equations, they will be quoted with "Quine quotes" or "semantics brackets": $[\![\ ]\!]$. For example, suppose we have some mathematical function f whose domain is Scheme expressions. To apply f to the program text `(λ (x y) (g x x))`, we write $f\,[\![\texttt{(}\lambda\texttt{ (x y) (g x x))}]\!]$. Within semantics brackets, we can "unquote" an expression with an italicised variable. So if $c = [\![\texttt{(f x x)}]\!]$, and we write $l = [\![\texttt{(}\lambda\texttt{ (x y) }c\texttt{)}]\!]$, then we mean that $l = [\![\texttt{(}\lambda\texttt{ (x y) (f x x))}]\!]$.

Conditional mathematical expressions are written with a "guarded expression" notation, *e.g.*:

$$\begin{aligned} \text{abs}\ x = {} & x > 0 \longrightarrow x \\ & x < 0 \longrightarrow -x \\ & \text{otherwise } 0. \end{aligned}$$

3.2 Syntax

The abstract syntax for CPS Scheme is:

$$\begin{array}{rcll}
\text{PR} & ::= & \text{LAM} & \\
\text{LAM} & ::= & (\lambda\ (v_1 \ldots v_n)\ c) & [v_i \in \text{VAR},\ c \in \text{CALL}] \\
\text{CALL} & ::= & (f\ a_1 \ldots a_n) & [f \in \text{FUN},\ a_i \in \text{ARG}] \\
 & & (\texttt{letrec}\ ((f_1\ l_1) \ldots)\ c) & [f_i \in \text{VAR},\ l_i \in \text{LAM},\ c \in \text{CALL}] \\
\text{FUN} & ::= & \text{LAM} + \text{REF} + \text{PRIM} & \\
\text{ARG} & ::= & \text{LAM} + \text{REF} + \text{CONST} & \\
\text{REF} & ::= & \text{VAR} & \\
\text{VAR} & ::= & \{\texttt{x}, \texttt{z}, \texttt{foo}, \ldots\} & \\
\text{CONST} & ::= & \{\texttt{3}, \texttt{\#f}, \ldots\} & \\
\text{PRIM} & ::= & \{\texttt{+}, \texttt{if}, \texttt{test-integer}, \ldots\} & \\
\text{LAB} & ::= & \{\ell_i, r_i, \ldots\} &
\end{array}$$

This abstract syntax defines the language we informally specified in the last chapter. All the various semantic definitions we'll be constructing will be defined over programs from this syntax. PR is the set of programs, which are just top-level lambdas. LAM, CALL, FUN, ARG, REF, VAR, CONST, and PRIM are just the sets of lambdas, call expressions, function expressions, argument expressions, variable references, variables, constants, and primops, as discussed earlier.

The set LAB is the set of expression labels. In order to have unique tags for each component of a program being analysed, we prefix each part of a program with a unique label. For example,

$$\ell\texttt{:(}\lambda\ \texttt{(x)}\ c\texttt{:(}r_1\texttt{:f}\ r_2\texttt{:x}\ k\texttt{:3}\ r_3\texttt{:x))}$$

is the fully-labelled expression for

$$\texttt{(}\lambda\ \texttt{(x) (f x 3 x)).}$$

Each lambda, call, constant, and variable reference in this expression is tagged with a unique label. Different occurrences of identical expressions receive distinct labels, so the two references to `x` have the different labels r_2 and r_3. Labels allow us to uniquely identify different pieces of a program. The abstract syntax just given doesn't show the labels; to be completely correct, the syntax description must have labels added to the right-hand sides of the LAM, CALL, REF, CONST, and PRIM definitions. Because they visually clutter our examples, we will suppress labels from expressions whenever convenient. We will also play fast and loose with the distinction between expressions and their associated labels; the meaning will always be clear.

We'll additionally assume a couple of useful syntactic properties about programs written in our syntax. First, we'll assume that all programs are "alphatised," that is, any given variable in the program is bound by only one lambda or `letrec` expression. It is a simple task to rename variables to enforce this restriction. Second, we'll assume our programs are closed; *i.e.*, have no free variables. This is also a simple property to check at compile time. Third, we'll assume that the arity of all primop applications is syntactically enforced. For example, we declare that `(if a b c d e f)` is not a legal expression, since the `if` primop only takes three arguments. Primops are not first-class, so it's easy to check this property at compile time. Doing so will simplify our upcoming semantic equations.

A useful syntactic function is the $binder$ function, which maps a variable to the label of the lambda or `letrec` construct that binds it. In the previous expression, $binder\,[\![\texttt{x}]\!] = \ell$. The alphatisation assumption makes this a well-defined function — no variable is bound by two different lambda or `letrec` expressions.

3.3 Standard Semantics

Figures 3.1 and 3.2 present a complete semantics for CPS Scheme. To keep things simple, the semantics leaves out side-effects (we will return to them in a later section).

The run-time values manipulated by the semantics are given by the sets Bas, Clo, Proc, D, and Ans:

Bas is the set of basic values. Our toy language has integers and a special boolean false value. We'll take any non-false value as a true value.

Clo is the set of lambda closures. A closure is a lambda/environment pair $\langle \ell,\ \beta \rangle$ (we'll return to environments in a moment).

Proc is the set of CPS Scheme procedures. A procedure is either a closure, a primop, or the special $stop$ value. Primops are represented by their syntactic identifier, from PRIM. The $stop$ procedure is how a program halts itself: when $stop$ is called on some value d, the program terminates with result d.

D is the set of run-time values a program can manipulate. A run-time datum is either a basic value or a procedure.

Ans is the domain of final answers the program can produce. If the program halts with no error, it produces a value from D. If the program encounters a run-time error (*e.g.*, divide-by-zero or trying to call a non-procedure), it halts with a special error value. The remaining possibility is that a program can get into an infinite loop, never halting. This is represented by the bottom value $\perp$.

$$
\begin{array}{rcl}
\text{Bas} & = & \mathcal{Z} + \{\text{false}\} \\
\text{Clo} & = & \text{LAM} \times \text{BEnv} \\
\text{Proc} & = & \text{Clo} + \text{PRIM} + \{stop\} \\
\text{D} & = & \text{Bas} + \text{Proc} \\
\text{Ans} & = & (\text{D} + \{\text{error}\})_\perp \\
\text{CN} & & \mathit{Contours} \\
\text{BEnv} & = & \text{LAB} \rightharpoonup \text{CN} \\
\text{VEnv} & = & (\text{VAR} \times \text{CN}) \rightharpoonup \text{D}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{PR} & : & \text{PR} \to \text{Ans} \\
\mathcal{K} & : & \text{CONST} \to \text{Bas} \\
\mathcal{A} & : & \text{ARG} \cup \text{FUN} \to \text{BEnv} \to \text{VEnv} \rightharpoonup \text{D} \\
\mathcal{C} & : & \text{CALL} \to \text{BEnv} \to \text{VEnv} \rightharpoonup \text{Ans} \\
\mathcal{F} & : & \text{Proc} \to \text{D}^* \to \text{VEnv} \rightharpoonup \text{Ans}
\end{array}
$$

Figure 3.1: Standard Semantics Domains and Functionalities

Our interpreter factors the environment into two parts: the *variable environment* ($ve \in$ VEnv), which is essentially a global structure, and the lexical *contour environment* ($\beta \in$ BEnv). A contour environment β maps syntactic binding constructs — lambda and `letrec` expressions — to *contours* or dynamic frames. Each time a lambda is called, a new contour is allocated for its bound variables. Contours are taken from the set CN (the integers will suffice). A variable paired with a contour is a *variable binding* $\langle v,\ b \rangle$. The variable environment ve, in turn, maps these variable bindings to actual values. The contour part of the variable binding pair $\langle v,\ b \rangle$ is what allows multiple bindings of the same identifier to coexist peacefully in the single variable environment.

For example, suppose we have a closure over some CPS Scheme lambda expression ℓ:`(`λ` (x y) ...x...y...)` which is bound to `f` and called twice during program evaluation, once with arguments `(f 3 7)`, and once with arguments `(f 22 46)`. The first time we enter the lambda, we allocate a new contour (say, 1), and bind `x` and `y` to 3 and 7 in the new contour. The second time we enter the lambda, we again allocate a new contour (this time, 2), and bind `x` and `y` to 22 and 46 in the new contour. The two different binding contexts are distinguished in the variable environment by the contours. We would end up with the variable environment

$$ve = [\ldots, \langle [\![\mathrm{x}]\!],\ 1\rangle \mapsto 3, \langle [\![\mathrm{y}]\!],\ 1\rangle \mapsto 7, \ldots, \langle [\![\mathrm{x}]\!],\ 2\rangle \mapsto 22, \langle [\![\mathrm{y}]\!],\ 2\rangle \mapsto 46, \ldots],$$

and two different lexical contexts, given by the contour environments

$$\beta' = [\ldots, \ell \mapsto 1],$$
$$\beta'' = [\ldots, \ell \mapsto 2].$$

Since contour environments determine how we look up variables in the variable environment, we pair them up with lambdas to get Scheme's lexical closures. This sort of factored-environment representation is a semantic model called the "contour model of block-structured processes" [Johnston 71]; as we'll see, it will help us with our eventual abstractions.

There are five semantic functions that comprise our CPS Scheme interpreter: $\mathcal{PR}$, $\mathcal{A}$, $\mathcal{K}$, $\mathcal{C}$, and $\mathcal{F}$:

- $\mathcal{PR}$ takes a top-level lambda, and runs the program to completion, returning the final value the program halts with. If the program never halts, $\mathcal{PR}$ returns $\bot$ (we are allowed to do this because $\mathcal{PR}$ is just a mathematical function — we are only pretending that it is an interpreter).

- $\mathcal{A}$ evaluates function and argument expressions in a given environmental context. Function and argument expressions overlap a good deal; $\mathcal{A}$ is defined on their union ARG $\cup$ FUN $=$ LAM $+$ REF $+$ PRIM $+$ CONST.

- $\mathcal{K}$ evaluates constants. It maps numerals to the corresponding integer, and the false identifier to the false value. We won't bother to specify $\mathcal{K}$ in detail.

- $\mathcal{C}$ evaluates a call expression in a given environmental context. Since calls don't return in CPS, this means it runs the program forward from the call to completion, returning the final value the program halts with.

- $\mathcal{F}$ applies a procedure to an argument vector, and runs the program forward from the application, again returning the final value the program halts with. $\mathcal{F}$ also takes the variable environment as an argument, so that variable lookups that occur during program execution can be performed.

The semantic functions are defined in figure 3.2. The figure defines a complete, if minimal, CPS Scheme interpreter.

- The $\mathcal{PR}$ function is simply a cover function that starts up the interpreter. $\mathcal{PR}$ calls $\mathcal{A}$ to evaluate its lambda ℓ in the empty environment, and then calls $\mathcal{F}$ to apply the result closure to an argument vector with one argument: the $stop$ continuation. In other words, the top-level lambda is closed in the empty environment, and applied to the $stop$ continuation. This runs the program forward until the $stop$ continuation is actually called, producing the final value passed as the argument to the $stop$ continuation.

$$
\begin{aligned}
&\mathcal{PR}\,\ell = \mathcal{F}\,f\ \langle stop\rangle\ [\,]\\
&\qquad \text{where } f = \mathcal{A}\,\ell\ [\,]\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}\,[\![k]\!]\,\beta\ ve &= \mathcal{K}\ k\\
\mathcal{A}\,[\![prim]\!]\,\beta\ ve &= prim\\
\mathcal{A}\,[\![v]\!]\,\beta\ ve &= ve\,\langle v,\ \beta(\mathit{binder}\,v)\rangle\\
\mathcal{A}\,[\![\ell]\!]\,\beta\ ve &= \langle \ell,\ \beta\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\,[\![(f\ \ a_1 \ldots a_n)]\!]\,\beta\ ve = {} & f' \notin \text{Proc} \ \longrightarrow\ \text{error}\\
& \text{otherwise } \mathcal{F}\ f'\ av\ ve\\
& \text{where } f' = \mathcal{A}\ f\ \beta\ ve\\
& \qquad\quad av{\downarrow}i = \mathcal{A}\ a_i\ \beta\ ve
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\,[\![c\texttt{:(letrec ((}f_1\ \ l_1\texttt{)...)}\ \ c'\texttt{)}]\!]\,\beta\ ve = {} & \mathcal{C}\ c'\ \beta'\ ve'\\
& \text{where } b = nb\\
& \qquad\quad \beta' = \beta[c \mapsto b]\\
& \qquad\quad ve' = ve\,[\langle f_i,\ b\rangle \mapsto \mathcal{A}\ l_i\ \beta'\ ve]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}\ \langle [\![\ell\texttt{:(}\lambda\ \ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!],\ \beta\rangle\ \ av\ ve =\\
&\qquad \text{length } av \neq n \ \longrightarrow\ \text{error}\\
&\qquad \text{otherwise } \mathcal{C}\ c\,(\beta[\ell \mapsto b])\,(ve\,[\langle v_i,\ b\rangle \mapsto av{\downarrow}i])\\
&\qquad\qquad \text{where } b = nb
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}\ stop\ av\ ve = {} & \text{length } av \neq 1 \ \longrightarrow\ \text{error}\\
& \text{otherwise } av{\downarrow}1
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}\ [\![\texttt{+}]\!]\ \langle x,\ y,\ k\rangle\ \ ve = {} & \textit{bad argument} \ \longrightarrow\ \text{error}\\
& \text{otherwise } \mathcal{F}\ k\ \ \langle x + y\rangle\ \ ve
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}\ [\![\texttt{if}]\!]\ \langle x,\ k_0,\ k_1\rangle\ \ ve = {} & \{k_0, k_1\} \not\subset \text{Proc} \ \longrightarrow\ \text{error}\\
& x \neq \text{false} \qquad\quad\ \longrightarrow\ \mathcal{F}\ k_0\ \ \langle\rangle\ \ ve\\
& \text{otherwise } \mathcal{F}\ k_1\ \ \langle\rangle\ \ ve
\end{aligned}
$$

Figure 3.2: Standard CPS Scheme Semantics

- The $\mathcal{A}$ function evaluates function and argument expressions (*i.e.*, variables, constants, primops, and lambdas) given the current variable environment ve and contour environment β. $\mathcal{A}$ has four cases. A constant k is evaluated by the constant specialist $\mathcal{K}$. A primop $prim$ evaluates to itself — that is, we use the actual syntactic token $prim$ for the primop's value. A variable reference v is evaluated in a two step process. First, the contour environment is indexed with the variable's binding lambda or `letrec` expression $binder\, v$ to find this variable's current contour. The contour and the variable are then used to index into the variable environment ve, giving the actual value. A lambda expression ℓ is evaluated to a closure by pairing it up with the current contour environment.

  Note that $\mathcal{A}$ can't produce a bottom or error value. This is because $\mathcal{A}$ only evaluates simple expressions that are guaranteed to terminate: variables, constants, primops and lambdas. The only possible run-time error evaluation of these expressions could produce is an unbound variable, and we explicitly restricted our syntax to rule out that possibility. This simplicity of evaluation is one of the pleasant features of CPS Scheme.

- The $\mathcal{C}$ function evaluates call expressions in a given environment context. It has two cases, one for simple calls $(f\ \ a_1 \ldots a_n)$, and one for `letrec`'s. To evaluate a simple call, $\mathcal{C}$ uses $\mathcal{A}$ to evaluate the function expression f and each argument a_i. The argument values are packaged up into an argument vector av. If f evaluates to a procedure, $\mathcal{C}$ uses $\mathcal{F}$ to apply the procedure to the argument vector and the current variable environment; $\mathcal{F}$ returns the final value produced by the program. If f evaluates to a non-procedure, then the program is aborted with the error value.

  To evaluate a `letrec` expression, $\mathcal{C}$ first allocates a new binding contour b. The new contour is allocated by the function nb, which is a contour "gensym" — it is defined to return a new, unused value each time it is called.[1] The contour environment β is augmented to create the inner contour environment $\beta' = \beta[c \mapsto b]$. Looking up the contour in β' for any of the variables f_i bound by the `letrec` will produce b. Then, the `letrec`'s lambdas l_i are evaluated in the inner environment β': $\mathcal{A}\ l_i\ \beta'\ ve$. Note that evaluation of a lambda just closes it with the contour environment — the variable environment ve is not used by $\mathcal{A}$, so it doesn't matter what we pass $\mathcal{A}$ for the variable environment.

  After evaluating the `letrec`'s lambdas, $\mathcal{C}$ binds the procedures to their corresponding variables f_i using the new contour b. This creates the new variable environment ve'. $\mathcal{C}$ then evaluates the inner call expression c' in the new environment, running the program forward from the `letrec`.

- The $\mathcal{F}$ function takes a procedure f, an argument vector av, and the variable environment ve; it runs the program forward from the procedure application, returning

[1] As defined, nb is not a proper function. This is easy to patch by adding an extra argument to the $\mathcal{F}$ and $\mathcal{C}$ functions. This is the kind of obfuscatory detail we are avoiding in this chapter. We will return to these issues in the next chapter.

the final value produced by the program. $\mathcal{F}$ is defined by cases: one for closure application, one for the $stop$ continuation, and one for each primop.

When a closure is applied to an argument vector, $\mathcal{F}$ first checks the arity of the closure's lambda against the length of the argument vector. If the closure is being applied to the wrong number of arguments, the program is aborted with the error value. Otherwise, the lambda's variables v_i are bound to their corresponding arguments $av{\downarrow}i$ in a new contour b. The variable environment ve is augmented with the new bindings, giving the new variable environment $ve\,[\langle v_i,\, b\rangle \mapsto av{\downarrow}i]$. The lexical contour environment β is augmented to show that the lambda's variables are bound in contour b, producing an inner contour environment of $\beta[\ell \mapsto b]$. The lambda's inner call expression c' is evaluated in the new environment context by $\mathcal{C}$, producing the result value for the program.

The definition of $\mathcal{F}$ for the terminating $stop$ procedure is simple: if the procedure is being applied to exactly one argument, that argument is returned as the result of the evaluation. Otherwise, the program is aborted with a run-time error.

The primops also have simple definitions. I show $\mathcal{F}$ for the `+` and `if` primop cases; other primops are similar. The `+` primop is applied to a three element argument vector $\langle x,\, y,\, k\rangle$. The addends x and y are added, and the sum packaged into a singleton argument vector which is passed to the continuation k. If the `+` primop is called on a bad argument vector, the program is aborted with the error value for result. For the purposes of `+`, an argument vector is bad if its first or second element is not an integer or if its third element is not a procedure.

The `if` primop takes three arguments $\langle x,\, k_0,\, k_1\rangle$ — a boolean value x and two continuations. If one of the continuations isn't a procedure, the program is aborted. If b is a true value, the "then" continuation k_0 is applied to an empty argument vector; otherwise the "else" continuation k_1 is applied to an empty argument vector.

These five functions — $\mathcal{PR}$, $\mathcal{A}$, $\mathcal{K}$, $\mathcal{C}$, and $\mathcal{F}$ — completely define the standard semantics of CPS Scheme. They specify that the meaning of a program is the value it produces when it's run. For example, applying $\mathcal{PR}$ to the following factorial program

```
(λ (k)
  (letrec ((f (λ (n c)
                (zero? n (λ (z)
                  (if z (λ () (c 1))
                        (λ () (- n 1 (λ (n1)
                                (f n1 (λ (a) (* a n c)))))))))))
    (f 5 k)))
```

produces 120.

3.4 Exact Control-Flow Semantics

We can derive a control-flow semantics from our standard semantics by "instrumenting the interpreter." We find every place in the semantics where a procedure is called, and we modify the semantics to record the call in some table. When the program terminates, the semantics discards the actual value computed, and returns the table instead. Our semantics is now one that maps a program, not to its result value, but to a record of all the calls that happened during the program execution.

Clearly, while this approach gives a well-defined mathematical definition of control-flow analysis, the definition is unrealisable as a computable algorithm. If the program never terminates, the interpreter will never halt, and never finish constructing the table. Further, the table produced reflects only the calls that happen during one particular execution of the program — it captures precisely the transfers of control that take place during that execution. Suppose we had more realistically included i/o in our semantics. To get a general picture of the program's control-flow structure, we would have to interpret the program for each possible input set, and join the result tables together. These issues are not a concern for now; we'll worry about how to make the analysis computable later. For now, it suffices that our instrumented interpreter serves as a very precise, formal definition of control-flow analysis.

3.4.1 Internal Call Sites

There is one fine point we must consider before proceeding to our control-flow semantics: not all procedures are called from program call sites. Consider the fragment `(+ a b (λ (s) (f a s)))`. Where is `(λ (s) (f a s))` called from? It is called from the innards of the `+` primop; there is no corresponding call site in the program syntax to mark this. We need to endow primops with special *internal call sites* to mark calls to functions that happen internal to the primop.

For each occurrence of a primop $(p{:}prim\ \ a_1 \ldots a_n)$ in a program, we associate a related set of internal call sites $ic_p^1, \ldots, ic_p^j$ for the primop's use. Simple primops, *e.g.* `+`, have a single internal call site, which marks the call to the primop's continuation. Conditional primops, *e.g.* `if`, have two internal call sites, one for the consequent continuation, and one for the alternate continuation. These ic identifiers give us names for the hidden call sites where primop continuations are called. So, we include them in the LAB set:

$$\mathrm{LAB} = \left\{\ell_i, r_i, c_i, ic_p^j, \ldots\right\}.$$

Now our label set includes the labels of all possible call sites in the program, both visible and hidden.

Some notational conveniences: Most ordinary primops only require a single internal call site. When this is the case, we'll drop the superscript index. Furthermore, when possible we'll include the actual primop with its label in the subscript. So if the primop with label p happens to be $[\![p{:}\texttt{+}]\!]$, we'll write its single internal call site as $ic_{p:\texttt{+}}$ instead of ic_p^1.

3.4.2 Instrumenting the Semantics

We only need to change the standard semantics slightly to produce our exact control-flow semantics (figure 3.3). The only domain we must change is the answer domain, Ans. Our new semantics produces *call caches*. A call cache ($\gamma \in$ CCache) is a partial map from call-site/contour-environment pairs to procedures:

$$\begin{aligned} \text{CCache} &= (\text{LAB} \times \text{BEnv}) \rightharpoonup \text{Proc} \\ \text{Ans} &= \text{CCache} \end{aligned}$$

Note that this definition of a call cache is more detailed than one that simply maps a call site to the procedures called from that site. This definition distinguishes the multiple calls that happen from a single call site by including the environment context (from BEnv) as part of the input to the cache.

Figure 3.3 shows the new interpreter. Instrumenting the standard interpreter involves two basic changes.

1. The places where procedures are called — from $\mathcal{PR}$'s top-level call, from $\mathcal{C}$'s simple-call case, and from $\mathcal{F}$'s continuation calls inside primop applications — must add their call to the call cache being built up by the interpretation.

2. Program termination, whether by run-time error or application of the $stop$ continuation, must discard the actual result value and produce instead the empty call cache [ ].

The $\mathcal{PR}$ function is similar to its previous definition: it closes the top-level lambda in the empty environment and applies it to the $stop$ continuation. This procedure application, however, evaluates to the call cache produced by running the rest of the program. $\mathcal{PR}$ must add to this cache the fact that the top-level procedure f was called from the top call in the empty environment. This is represented by the cache entry $[\langle c_{\text{top}}, [\,]\rangle \mapsto f]$, where the label c_{top} represents the call site for the top-level call. The updated call cache is the call cache for the entire program execution.

The $\mathcal{A}$ function hasn't changed at all, since evaluating argument expressions doesn't involve calling procedures.

The simple-call case of the $\mathcal{C}$ function changes in two respects. First, in the error case, it aborts the program and returns the empty call cache [ ]. $\mathcal{C}$ is responsible for returning the call cache for program execution from its call onwards; since the program has halted at the call, the empty call cache is the right value to produce. Second, if it actually performs a call, it records the fact that procedure f' was called from call c in context β, represented by call cache entry $[\langle c, \beta\rangle \mapsto f']$. This entry is added to the call cache produced by running the rest of the program.

The `letrec` case doesn't change, since the `letrec`'s local computation is simply a binding operation, and doesn't perform any procedure calls itself.

The $\mathcal{F}$ function performs calls to its primops' continuations, and these calls must be recorded. For example, the `+` primop performs a call to its continuation k from its internal

$$
\begin{aligned}
&\mathcal{PR}\,\ell = (\mathcal{F}\ f\ \langle stop\rangle\ [\,])[\langle c_{\text{top}},\,[\,]\rangle \mapsto f]\\
&\qquad \text{where } f = \mathcal{A}\,\ell\ [\,]\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}\,[\![k]\!]\ \beta\ ve &= \mathcal{K}\ k\\
\mathcal{A}\,[\![prim]\!]\ \beta\ ve &= prim\\
\mathcal{A}\,[\![v]\!]\ \beta\ ve &= ve\,\langle v,\ \beta(\text{binder}\,v)\rangle\\
\mathcal{A}\,[\![\ell]\!]\ \beta\ ve &= \langle \ell,\ \beta\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\,[\![c{:}(f\ \ a_1\ldots a_n)]\!]\ \beta\ ve = {}& f' \notin \text{Proc} \ \longrightarrow\ [\,]\\
& \text{otherwise } (\mathcal{F}\ f'\ av\ ve)[\langle c,\ \beta\rangle \mapsto f']\\
& \text{where } f' = \mathcal{A}\ f\ \beta\ ve\\
& \qquad\quad av{\downarrow}i = \mathcal{A}\ a_i\ \beta\ ve
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\,[\![c{:}(\texttt{letrec}\ \ ((f_1\ \ l_1)\ldots)\ \ c')]\!]\ \beta\ ve = {}& \mathcal{C}\ c'\ \beta'\ ve'\\
& \text{where } b = nb\\
& \qquad \beta' = \beta[c \mapsto b]\\
& \qquad ve' = ve\,[\langle f_i,\ b\rangle \mapsto \mathcal{A}\ l_i\ \beta'\ ve]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{F}\ \langle[\![\ell{:}(\lambda\ \ (v_1\ldots v_n)\ \ c)]\!],\ \beta\rangle\ \ av\ ve =\\
&\qquad \text{length } av \neq n\ \ \longrightarrow\ \ [\,]\\
&\qquad \text{otherwise } \mathcal{C}\ c\ (\beta[\ell \mapsto b])\ (ve[\langle v_i,\ b\rangle \mapsto av{\downarrow}i])\\
&\qquad\qquad\quad \text{where } b = nb
\end{aligned}
$$

$$
\mathcal{F}\ stop\ av\ ve = [\,]
$$

$$
\begin{aligned}
\mathcal{F}\ [\![p{:}\texttt{+}]\!]\ \langle x,\ y,\ k\rangle\ \ ve = {}& \textit{bad argument}\ \ \longrightarrow\ \ [\,]\\
& \text{otherwise } (\mathcal{F}\ k\ \ \langle x+y\rangle\ \ ve)\Big[\Big\langle ic_{p:\texttt{+}},\ \beta\Big\rangle \mapsto k\Big]\\
& \qquad \text{where } b = nb\\
& \qquad\qquad \beta = [p \mapsto b]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}\ [\![p{:}\texttt{if}]\!]\ \langle x,\ k_0,\ k_1\rangle\ \ ve = {}& \{k_0, k_1\} \not\subset \text{Proc}\ \ \longrightarrow\ \ [\,]\\
& x \neq \text{false} \qquad\quad\ \longrightarrow\ \ (\mathcal{F}\ k_0\ \ \langle\rangle\ \ ve)\Big[\Big\langle ic^{0}_{p:\texttt{if}},\ \beta\Big\rangle \mapsto k_0\Big]\\
& \text{otherwise } (\mathcal{F}\ k_1\ \ \langle\rangle\ \ ve)\Big[\Big\langle ic^{1}_{p:\texttt{if}},\ \beta\Big\rangle \mapsto k_1\Big]\\
& \text{where } b = nb\\
& \qquad \beta = [p \mapsto b]
\end{aligned}
$$

Figure 3.3: Exact Control-Flow Semantics

call site $ic_{p:\texttt{+}}$. This is recorded by call cache entry $\left[\left\langle ic_{p:\texttt{+}},\ \beta\right\rangle \mapsto k\right]$ which is added to the cache produced by running the rest of the program.

Note that + allocates a new contour when it is entered, even though it binds no variables. In the course of executing a program, control could pass through this primop several times; each time, there will be a call from $ic_{p:\texttt{+}}$ to a continuation. We need to distinguish these multiple $ic_{p:\texttt{+}}$ calls in the call cache we are constructing. Multiple calls from the same call site are distinguished in the call cache by an environment context β. So, even though it binds no variables, when control enters a + primop, we allocate a new contour $b = nb$, and construct a new contour environment $\beta = [p \mapsto b]$ to make a unique index $\left\langle ic_{p:\texttt{+}},\ \beta\right\rangle$ in the call cache.

Similarly, `if` adds the appropriate entry to the final call cache, either recording that the true continuation k_0 was called from the internal call site $ic^0_{p:\texttt{if}}$, or the false continuation k_1 was called from $ic^1_{p:\texttt{if}}$.

3.5 Abstract Control-Flow Semantics

Now that we've defined our control-flow analysis semantics, we have a formal description of the control-flow problem. The final step is to abstract our semantics to a computable approximate semantics that is useful and safe.

The major reason the exact semantics is uncomputable is because the domains and ranges of the meaning functions are (uncountably) infinite. This is due to the infinite number of distinct environments that can be created at run time.

For example, the call cache constructed by the analysis is a potentially infinite table. If a call cache simply mapped call sites to lambdas, it would necessarily be finite, because a finite program only has a finite number of call sites and lambdas. A call cache, however, maps *call contexts* to *closures*. The difference is environment information. A call context $\langle c,\ \beta\rangle$ is a call paired with a contour environment; similarly, a closure $\langle \ell,\ \beta\rangle$ is a lambda paired with a contour environment. Although there are only a finite number of lambdas and calls in a given program, execution of that program can give rise to an unbounded number of distinct environments.

For example, consider the following Scheme loop:

```
(letrec ((loop (λ (f) (loop (λ (n) (* 2 (f n)))))))
  (loop (λ (m) m)))
```

The `loop` procedure calls itself with a function that is its input function doubled. What is the set of procedures that `f` could be bound to during execution of this loop? It could be bound to any of the procedures

$$\{n \mapsto n, n \mapsto 2n, n \mapsto 4n, \ldots\}\,.$$

This example shows how a finite program can give rise to an infinite set of procedures. There are only a finite number of lambda expressions in a given program; the infinite sets of procedures arise because we can close these lambdas with an infinite set of environments. If we can collapse our infinite set of environments down to a finite approximation, then we can successfully compute a control-flow cache function.

The source of the infinite environments are the infinite contours allocated by nb during program interpretation. If we force nb to allocate contours from a finite set, then we'll only have a finite set of contour environments to deal with. This, in turn, will restrict our analysis to a finite set of call contexts and closures, and our table construction will converge — giving us a computable analysis. Of course, collapsing an infinite set of binding contours down to a finite set is going to merge some variable bindings together — that is, looking up a variable will produce a set of possibilities, instead of a single definite value. This is the loss of precision we must accept for a computable abstraction.

To repeat the point: the key to making our analysis computable lies in restricting our contour set to be finite. This is the central abstraction of the computable analysis.

It should now be clear why we factored the environment structure from the beginning. The variable binding mechanism is what gave rise to the infinite environment structure; factoring the environment exposed this mechanism to possible abstraction. Abstracting the contours and merging bindings was the critical step that allowed us to reduce this infinite structure to a finite, computable one.

The other major change we must make in abstracting our semantics concerns conditional branches. Since we do not in general know at compile time which way a conditional branch will go, we must abstract away conditional dependencies. The `if` primop now "branches both ways." That is, the caches arising from the consequent and alternate paths are both computed; these are joined together to give the result cache returned by the `if` primop. Removing this data dependency has a further consequence: because basic values (from Bas) are no longer tested by conditional branches, the basic value set can be dispensed with. Since our semantics concerns itself solely with control flow, the only values that need to be considered are those representing CPS Scheme procedures. (Had we included I/O in our original perfect control-flow semantics, this change would allow us to abstract away those dependencies, as well.)

The domains and functionalities for the abstract control-flow semantics are shown in figure 3.4. The abstract domains and functions are distinguished by putting "hats" on the variable names. Closures ($\widehat{\mathrm{Clo}}$) are still lambda/environment pairs, but the environment is now an abstract environment (from $\widehat{\mathrm{BEnv}}$). The set of run-time values ($\widehat{\mathrm{D}}$) has changed significantly. As discussed, the set of basic values (Bas) has been dropped entirely. Further, an element from $\widehat{\mathrm{D}}$ is a *set* of abstract procedures instead of a single procedure, reflecting the fact that expressions now evaluate to sets of possibilities in the abstract semantics.

We've only specified in figure 3.4 that the abstract contour set $\widehat{\mathrm{CN}}$ is finite — we haven't said anything about what particular finite set it is. We can choose different abstractions by choosing different finite sets for $\widehat{\mathrm{CN}}$ and different functions $\widehat{nb}$ for allocating its contours. These different choices have varying cost/precision tradeoffs; we'll be considering various possibilities in following sections. However, the general model we are developing in this section will apply to all of the possible choices. The important particular to keep in mind is that the infinite set of exact contours has been reduced to a finite set of abstract contours.

$$
\begin{array}{rcl}
\widehat{\text{Clo}} & = & \text{LAM} \times \widehat{\text{BEnv}} \\
\widehat{\text{Proc}} & = & \widehat{\text{Clo}} + \text{PRIM} + \{stop\} \\
\widehat{\text{D}} & = & \mathcal{P}(\widehat{\text{Proc}}) \\
\widehat{\text{CCache}} & = & (\text{LAB} \times \widehat{\text{BEnv}}) \to \widehat{\text{D}} \\
\widehat{\text{CN}} & = & \textit{Contours (finite)} \\
\widehat{\text{BEnv}} & = & \text{LAB} \rightharpoonup \widehat{\text{CN}} \\
\widehat{\text{VEnv}} & = & (\text{VAR} \times \widehat{\text{CN}}) \to \widehat{\text{D}}
\end{array}
\qquad
\begin{array}{rcl}
\widehat{\mathcal{PR}} & : & \text{PR} \to \widehat{\text{CCache}} \\
\widehat{\mathcal{A}} & : & \text{ARG} \cup \text{FUN} \to \widehat{\text{BEnv}} \to \widehat{\text{VEnv}} \rightharpoonup \widehat{\text{D}} \\
\widehat{\mathcal{C}} & : & \text{CALL} \to \widehat{\text{BEnv}} \to \widehat{\text{VEnv}} \to \widehat{\text{CCache}} \\
\widehat{\mathcal{F}} & : & \widehat{\text{Proc}} \to \widehat{\text{D}}^* \to \widehat{\text{VEnv}} \to \widehat{\text{CCache}}
\end{array}
$$

Figure 3.4: Abstract Semantics Domains and Functionalities

Because we are collapsing variable bindings together, evaluating a variable now yields a set of possibilities, instead of a single value. This is reflected in the functionality of abstract variable environments: a variable environment $\widehat{ve} \in \widehat{\text{VEnv}}$ maps abstract variable bindings to sets of abstract procedures (from $\widehat{\text{D}}$).

If it helps you to keep in mind a particular contour abstraction when reading these semantics, here is the simplest possible abstraction, which we'll be discussing in the next section: all contours are identified together to a single abstract contour. So $\widehat{\text{CN}} = \{1\}$, *and* $\widehat{nb}$ *is just the constant function that always returns 1. For a given variable* `x`, *this abstraction puts all values bound to* `x` *into the same set.*

Figure 3.4 shows the critical effect of the contour abstraction. By making $\widehat{\text{CN}}$ finite (and assuming a finite program, so that LAB, LAM, and PRIM are also finite), *all other sets become finite.*

The functionalities of our semantic functions $\widehat{\mathcal{PR}}$, $\widehat{\mathcal{F}}$, $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{A}}$ remain essentially unchanged from their exact counterparts — we've altered them only by adding hats to all the appropriate sets.

Figure 3.5 shows the definition of the abstract semantic functions. $\mathcal{PR}$, again, is just a cover function, performing the initial call to the top-level lambda. $\widehat{\mathcal{A}}$ now evaluates argument expressions to sets of abstract procedures. Since a constant k can not possibly evaluate to a procedure, $\widehat{\mathcal{A}}$ returns the empty set for this case. Evaluating a primop $prim$ can only produce that primop, so $\widehat{\mathcal{A}}$ produces the singleton set $\{prim\}$ for this case. Similarly, evaluating a lambda expression ℓ must result in a closure $\left\langle \ell,\, \widehat{\beta} \right\rangle$ over that lambda, so $\widehat{\mathcal{A}}$ produces the corresponding singleton set. $\widehat{\mathcal{A}}$ evaluates variables using exactly the same lookup process that its exact counterpart uses — the difference being that the index into the abstract variable environment $\widehat{ve}$ produces a value set instead of a specific value.

$\widehat{\mathcal{C}}$ is very similar to its exact counterpart. The major difference for the simple-call case lies in the fact that the function expression f now evaluates to a set of procedures F. $\widehat{\mathcal{C}}$ must run the program forward from the call for each possible procedure $f' \in F$ that we could potentially be branching to. The caches produced by each application are combined (by joining the caches together, which gives a cache producing the union of all the individual caches' contributions: $(\hat{\gamma}_1 \sqcup \hat{\gamma}_2)\left\langle c,\, \widehat{\beta} \right\rangle = \hat{\gamma}_1 \left\langle c,\, \widehat{\beta} \right\rangle \cup \hat{\gamma}_2 \left\langle c,\, \widehat{\beta} \right\rangle$). To this result cache we add $\widehat{\mathcal{C}}$'s local contribution $\left[\left\langle c,\, \widehat{\beta} \right\rangle \mapsto F\right]$, which states that from call c and context $\widehat{\beta}$, we

$$\widehat{\mathcal{PR}}\ \ell = \left(\widehat{\mathcal{F}}\ f\ \langle\{stop\}\rangle\ [\,]\right) \sqcup [\langle c_{\text{top}}, [\,]\rangle \mapsto \{f\}]$$
$$\text{where } \{f\} = \widehat{\mathcal{A}}\ \ell\ [\,]\ [\,]$$

$$\widehat{\mathcal{A}}\ [\![k]\!]\ \widehat{\beta}\ \widehat{ve} = \emptyset$$
$$\widehat{\mathcal{A}}\ [\![prim]\!]\ \widehat{\beta}\ \widehat{ve} = \{prim\}$$
$$\widehat{\mathcal{A}}\ [\![\ell]\!]\ \widehat{\beta}\ \widehat{ve} = \left\{\left\langle \ell,\ \widehat{\beta}\right\rangle\right\}$$
$$\widehat{\mathcal{A}}\ [\![v]\!]\ \widehat{\beta}\ \widehat{ve} = \widehat{ve}\ \left\langle v,\ \widehat{\beta}(\mathrm{binder}\ v)\right\rangle$$

$$\widehat{\mathcal{C}}\ [\![c\!:\!(f\ \ a_1 \ldots a_n)]\!]\ \widehat{\beta}\ \widehat{ve} = \left(\bigsqcup_{f' \in F} \widehat{\mathcal{F}}\ f'\ \widehat{av}\ \widehat{ve}\right) \sqcup \left[\left\langle c,\ \widehat{\beta}\right\rangle \mapsto F\right]$$
$$\text{where } F = \widehat{\mathcal{A}}\ f\ \widehat{\beta}\ \widehat{ve}$$
$$\widehat{av}{\downarrow}i = \widehat{\mathcal{A}}\ a_i\ \widehat{\beta}\ \widehat{ve}$$

$$\widehat{\mathcal{C}}\ [\![c\!:\!(\texttt{letrec}\ ((f_1\ \ l_1)\ldots)\ \ c')]\!]\ \widehat{\beta}\ \widehat{ve} = \widehat{\mathcal{C}}\ c'\ \widehat{\beta}'\ \widehat{ve}'$$
$$\text{where } \hat{b} = \widehat{nb}$$
$$\widehat{\beta}' = \widehat{\beta}\left[c \mapsto \hat{b}\right]$$
$$\widehat{ve}' = \widehat{ve} \sqcup \left[\left\langle f_i,\ \hat{b}\right\rangle \mapsto \widehat{\mathcal{A}}\ l_i\ \widehat{\beta}'\ \widehat{ve}\right]$$

$$\widehat{\mathcal{F}}\ \left\langle [\![\ell\!:\!(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!],\ \widehat{\beta}\right\rangle\ \widehat{av}\ \widehat{ve} =$$
$$\text{length}\ \widehat{av} \neq n \ \longrightarrow\ [\,]$$
$$\text{otherwise}\ \widehat{\mathcal{C}}\ c\ \left(\widehat{\beta}\left[\ell \mapsto \hat{b}\right]\right)\ \left(\widehat{ve} \sqcup \left[\left\langle v_i,\ \hat{b}\right\rangle \mapsto \widehat{av}{\downarrow}i\right]\right)$$
$$\text{where } \hat{b} = \widehat{nb}$$

$$\widehat{\mathcal{F}}\ [\![p\!:\!\texttt{+}]\!]\ \left\langle \hat{x},\ \hat{y},\ \hat{k}\right\rangle\ \widehat{ve} = \left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ f\ \langle\emptyset\rangle\ \widehat{ve}\right) \sqcup \left[\left\langle ic_{p:\texttt{+}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \hat{b} = \widehat{nb}$$
$$\widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ [\![p\!:\!\texttt{if}]\!]\ \left\langle \hat{x},\ \hat{k}_0,\ \hat{k}_1\right\rangle\ \widehat{ve} = \left(\bigsqcup_{f \in \hat{k}_0} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \widehat{ve}\right) \sqcup \left(\bigsqcup_{f \in \hat{k}_1} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \widehat{ve}\right)$$
$$\sqcup \left[\left\langle ic^0_{p:\texttt{if}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}_0,\ \left\langle ic^1_{p:\texttt{if}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}_1\right]$$
$$\text{where } \hat{b} = \widehat{nb}$$
$$\widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ stop\ \widehat{av}\ \widehat{ve} = [\,]$$

Figure 3.5: Abstract Semantic Functions

could have called any of the procedures in F. Because we are collapsing contours together, we may already have a set of procedures associated with call cache entry $\left\langle c,\, \widehat{\beta}\right\rangle$. This is why the cache addition is made with a join operation, so that we are in effect *adding* the set of procedures F to any that we may have already recorded in the call cache under the $\left\langle c,\, \widehat{\beta}\right\rangle$ entry. By not overwriting any old information we have, only adding to it, we keep our analysis conservative.

$\widehat{\mathcal{C}}$ in the `letrec` case is similar to the exact version. Because we might already have entries in the variable environment $\widehat{ve}$ for the variable bindings $\left\langle f_i,\, \hat{b}\right\rangle$, we again must join the new procedures in with the current entries to produce the new environment $\widehat{ve}'$.

$\widehat{\mathcal{F}}$ applies an abstract procedure to a vector of value sets. If the procedure is a closure $\left\langle \ell,\, \widehat{\beta}\right\rangle$, the arity of the lambda is checked. If the closure is being applied to the wrong number of arguments, the abstract interpretation is aborted, and $\widehat{\mathcal{F}}$ returns the empty call cache $[\,]$. Otherwise, $\widehat{\mathcal{F}}$ allocates an abstract contour ($\hat{b} = \widehat{nb}$) in which to bind the lambda's variables. Just as in $\widehat{\mathcal{C}}$'s `letrec` case, the variable bindings $\left\langle v_i,\, \hat{b}\right\rangle$ might already have entries in the variable environment $\widehat{ve}$. We add the set of values $\widehat{av}{\downarrow}i$ to whatever values we already have bound to entry $\left\langle v_i,\, \hat{b}\right\rangle$ in the variable environment $\widehat{ve}$, producing the new environment $\widehat{ve} \sqcup \left[\left\langle v_i,\, \hat{b}\right\rangle \mapsto \widehat{av}{\downarrow}i\right]$. The lambda's inner call c is evaluated in the updated environment, giving the call cache for executing the rest of the program.

If the procedure is the `+` primop, the argument vector passed to it contains three values: $\left\langle \hat{x},\, \hat{y},\, \hat{k}\right\rangle$. The argument $\hat{k}$ is a set of possible continuations, and the abstract addends $\hat{x}$ and $\hat{y}$ are irrelevant — they have no procedural interpretation. $\widehat{\mathcal{F}}$ runs the interpretation forward for each abstract continuation $f \in \hat{k}$. In the exact semantics, the continuation was applied to the sum of the primop's addends; in the abstract semantics, we know that whatever the sum could be, it couldn't possibly be a procedure, so its abstraction as a value set is $\emptyset$. So each abstract continuation f is applied to the argument vector $\langle\emptyset\rangle$. The resulting call caches are all joined together, and `+`'s local contribution $\left[\left\langle ic_{p:+},\, \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$ is added, which states that all of the continuations passed in as `+`'s argument $\hat{k}$ could have been called from `+`'s internal call site in the environment context $\widehat{\beta}$.

Note that the abstract definition of the `+` primop does less error checking than the exact one: it doesn't check to ensure that the addends are integer values. For example, if the exact control-flow semantics was applied to the expression `(+ 3 #f k)`, it would immediately halt the interpretation, making no further entries to the call cache. The approximate semantics, however, would record the primop's call to the continuation, and continue the interpretation. This is acceptable, since including some extra calls in the final call cache is erring on the conservative side – the abstract analysis hasn't left out any of the actual calls, only included a few extras. This (conservative) weakening of the run-time error checking is a pervasive feature of the abstract analysis.

The `if` primop case is similar to the `+` case, except that we branch both ways from the conditional.

The $stop$ case is again quite simple: no calls are made after applying the $stop$ procedure, so the empty call cache is returned.

3.6 0CFA

The abstract semantics of the last section didn't specify which finite set to choose for the abstract contours $\widehat{\text{CN}}$, or what rule to use for allocating these contours on procedure entry. In this section, we'll look at the simplest possible abstraction, in which there is only a single abstract contour: $\widehat{\text{CN}} = \{1\}$. Thus the $\widehat{nb}$ function is just the constant function always returning 1. In the exact semantics, all bindings of a given variable were kept distinct; in this simple abstraction, all bindings of a given variable are merged together. Suppose that, over the lifetime of a program some variable `x` is bound to seven different values: $\{d_1, d_2, \ldots, d_7\}$. In 0CFA, after these bindings have been made, evaluation of `x` in any environmental context produces not one of the d_i, but the entire set. This trivial abstraction is called "0th-order Control-Flow Analysis" (0CFA) (we'll consider other, more detailed abstractions in the next section).

Even though 0CFA is quite simple, it is still very useful, particularly for intra-procedural flow analysis. For example, the implementation of induction-variable elimination discussed in chapter 7 is based on a 0CFA analysis. Additionally, 0CFA has the attraction of being fast.

We could write down the equations for the 0CFA semantics simply by replacing "$\widehat{nb}$" with "1" throughout the equations of the previous section. However, once we do this, we can simplify the equations further (figures 3.6, 3.7). 0CFA is a very special abstraction because it is the trivial case. Many of the structures in the general abstract semantics collapse to degenerate cases in 0CFA. For example, once we specify that $\widehat{\text{CN}} = \{1\}$, then contour environments become degenerate: every environment β maps all its labels to 1. So we can dispose of contour environments in the special case of 0CFA. Similarly, the variable/contour pairs $\langle v,\, b\rangle$ that index variable environments become degenerate, since the contour, again, must always be 1. So, we can simplify the variable environment domain to be maps from variables to sets of values: VAR $\rightarrow \widehat{\text{D}}$. The same reasoning applies to closures and call caches: the context information β that is paired with a lambda ℓ to make a closure, and paired with a call to make a call cache index is degenerate. So we can represent an abstract procedure with just a lambda, and a call cache now maps calls to sets of lambdas.

$$
\begin{array}{rcl@{\qquad}rcl}
\widehat{\text{Proc}} & = & \text{LAM} + \text{PRIM} + \{\mathit{stop}\} & \widehat{\mathcal{PR}} & : & \text{PR} \rightarrow \widehat{\text{CCache}} \\
\widehat{\text{D}} & = & \mathcal{P}(\widehat{\text{Proc}}) & \hat{\mathcal{A}} & : & \text{ARG} \cup \text{FUN} \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{D}} \\
\widehat{\text{CCache}} & = & \text{LAB} \rightarrow \widehat{\text{D}} & \hat{\mathcal{C}} & : & \text{CALL} \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{CCache}} \\
\widehat{\text{VEnv}} & = & \text{VAR} \rightarrow \widehat{\text{D}} & \hat{\mathcal{F}} & : & \widehat{\text{Proc}} \rightarrow \widehat{\text{D}}^* \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{CCache}}
\end{array}
$$

Figure 3.6: 0CFA Domains and Functionalities

The 0CFA analysis is interesting because it analyses pure control-flow structure; environment structure is ignored completely. It provides a direct answer to the question "Which lambdas are called from which call sites?" Because it is so simple, it can be computed faster than other, more detailed abstractions. However, the simplicity carries a price of reduced

$$\widehat{\mathcal{PR}}\,\ell = \left(\widehat{\mathcal{F}}\ f\ \langle\{stop\}\rangle\ [\,]\right) \sqcup [c_{\text{top}} \mapsto \{f\}]$$
$$\text{where } \{f\} = \widehat{\mathcal{A}}\,\ell\ [\,]$$

$$\begin{aligned}
\widehat{\mathcal{A}}\,[\![k]\!]\ \widehat{ve} &= \emptyset \\
\widehat{\mathcal{A}}\,[\![prim]\!]\ \widehat{ve} &= \{prim\} \\
\widehat{\mathcal{A}}\,[\![\ell]\!]\ \widehat{ve} &= \{\ell\} \\
\widehat{\mathcal{A}}\,[\![v]\!]\ \widehat{ve} &= \widehat{ve}\ v
\end{aligned}$$

$$\widehat{\mathcal{C}}\,[\![c\text{:}(f\ \ a_1 \ldots a_n)]\!]\ \widehat{ve} = \left(\bigsqcup_{f' \in F} \widehat{\mathcal{F}}\ f'\ \widehat{av}\ \widehat{ve}\right) \sqcup [c \mapsto F]$$
$$\begin{aligned}\text{where } F &= \widehat{\mathcal{A}}\ f\ \widehat{ve} \\ \widehat{av}{\downarrow}i &= \widehat{\mathcal{A}}\ a_i\ \widehat{ve}\end{aligned}$$

$$\widehat{\mathcal{C}}\,[\![c\text{:}(\texttt{letrec}\ \ ((f_1\ \ l_1)\ldots)\ \ c')]\!]\ \widehat{ve} = \widehat{\mathcal{C}}\ c'\ \widehat{ve}'$$
$$\text{where } \widehat{ve}' = \widehat{ve} \sqcup \left[f_i \mapsto \widehat{\mathcal{A}}\ l_i\ \widehat{ve}\right]$$

$$\widehat{\mathcal{F}}\,[\![\ell\text{:}(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]\ \widehat{av}\ \widehat{ve} = \text{length}\ \widehat{av} \neq n \ \longrightarrow\ [\,]$$
$$\text{otherwise } \widehat{\mathcal{C}}\ c\,(\widehat{ve} \sqcup [v_i \mapsto \widehat{av}{\downarrow}i])$$

$$\widehat{\mathcal{F}}\,[\![p\text{:}\texttt{+}]\!]\ \left\langle \hat{x},\ \hat{y},\ \hat{k}\right\rangle\ \widehat{ve} = \left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ f\ \langle\emptyset\rangle\ \ \widehat{ve}\right) \sqcup \left[ic_{p:\texttt{+}} \mapsto \hat{k}\right]$$

$$\widehat{\mathcal{F}}\,[\![p\text{:}\texttt{if}]\!]\ \left\langle \hat{x},\ \hat{k}_0,\ \hat{k}_1\right\rangle\ \widehat{ve} = \left(\bigsqcup_{f \in \hat{k}_0} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \ \widehat{ve}\right) \sqcup \left(\bigsqcup_{f \in \hat{k}_1} \widehat{\mathcal{F}}\ f\ \langle\rangle\ \ \widehat{ve}\right)$$
$$\sqcup \left[ic^0_{p:\texttt{if}} \mapsto \hat{k}_0,\ ic^1_{p:\texttt{if}} \mapsto \hat{k}_1\right]$$

$$\widehat{\mathcal{F}}\ stop\ \widehat{av}\ \widehat{ve} = [\,]$$

Figure 3.7: 0CFA Semantics Functions

precision. Collapsing call contexts together can merge together calls from the same call site that happen in otherwise distinct environment contexts, causing distinct control paths to merge unnecessarily. For example, consider a single subroutine `f` which is called from two different places in a program:

$$\ldots c_1\texttt{:(f 3 } k_1\texttt{:(}\lambda\texttt{ (a1) ...)) ... } c_2\texttt{:(f 4 } k_2\texttt{:(}\lambda\texttt{ (a2) ...)) ...}$$

$$\ell_3\texttt{:(}\lambda\texttt{ (n k) ... } c_3\texttt{:(k answer))}$$

In 0CFA, we record that

1. we could branch to ℓ_3 from c_1;
2. we could branch to ℓ_3 from c_2; and
3. we could branch to either k_1 or k_2 from c_3.

This information allows for spurious control-flow paths — it allows for the paths $c_1\ell_3 \ldots c_3k_2$ and $c_2\ell_3 \ldots c_3k_1$. That is, the analysis includes the possibility that the call from c_1 could return to k_2, and *vice versa*.

A more detailed analysis might distinguish the abstract contours allocated when entering ℓ_3, so that the two different bindings of the continuation `k` are kept distinct. This would eliminate the spurious control-flow paths. We would record that

1. we could branch to ℓ_3 from c_1;
2. we could branch to ℓ_3 from c_2;
3. we could branch to k_1 from c_3 *in the context created by entering ℓ_3 from c_1*; and
4. we could branch to k_2 from c_3 *in the context created by entering ℓ_3 from c_2*.

The more detailed environment information in the call cache eliminates the spurious control-flow paths. These sorts of considerations lead us to consider other, more detailed abstractions.

3.7 Other Abstractions

A more precise abstraction, which we won't explore in detail in this chapter, is to distinguish the contours allocated when a lambda is called from two distinct call sites. In this abstraction, which we'll call 1st-Order Control-Flow Analysis (1CFA), a contour is a call site: $\widehat{\mathrm{CN}} =$ CALL. We change our semantics so that whenever a call is performed, the call site's label is passed along as an extra argument c_{from} to $\widehat{\mathcal{F}}$. Then, abstract contour allocation is just using c_{from} — that is, we just replace every occurrence of "$\widehat{nb}$" in the $\widehat{\mathcal{F}}$ equations with "c_{from}".

So, if a given lambda is called from five different call sites, we will bind its variables in five different contours. All values passed to the lambda from a given call site will be merged, but values passed from different call sites will be distinct.

Since contours are now call sites, and a finite program has only a finite number of call sites, the analysis remains computable. The greater number of contours, however, means that the analysis will take longer to converge than the simple 0CFA analysis. So 1CFA buys greater precision with increased analysis time.

The idea of 1CFA is similar to some of the abstractions used in Hudak's work on reference counting [Hudak 86b]. It isn't difficult to come up with other abstractions, with different cost/precision tradeoffs. Choosing a good abstraction that is well-tuned to typical program usage is not a topic that I have explored in depth, although it certainly merits study. Deutsch has experimented with a number of different procedural abstractions in his work on semantic descriptions in a normal-order λ-calculus [Deutsch 90]. These abstractions all seem adaptable to the control-flow semantics in this dissertation, and thus might be considered.

3.8 Semantic Extensions

We restricted our language and its semantics in order to simplify the presentation of the analysis. Now that we've developed the basic analysis, we can extend it to handle other semantic features: side-effects, external procedures and calls, and user-procedure/continuation partitioning. These extensions will be presented in less detail than the basic analysis.

3.8.1 Side Effects

Scheme and ML allow side-effects, a feature missing from the semantics in the preceding sections. We need to extend the basic analysis to handle this feature.

To keep the mechanisms as simple as possible, we'll use a very simple side-effects system. We'll add a single mutable data structure, the *cell*. A cell is a first-class data structure that contains one value, called its *contents*. Cells can be allocated, and their contents can be altered or fetched. We need three new primops to manipulate cells: `new`, `set`, and `contents`.

- We allocate a cell with the `new` primop:
  `(new 3 (λ (cell) ...))` binds `cell` to a new cell with contents 3.
- We alter a cell's contents with the `set` primop:
  `(set cell 5 (λ () ...))` changes `cell`'s contents to 5.
- We fetch a cell's contents with the `contents` primop:
  `(contents cell (λ (val) ...))` binds `val` to the contents of `cell`.

Scheme users can think of cells as one-element vectors, or cons cells with just a car slot but no cdr slot. ML users can think of cells as ML `ref`'s. Cells serve to capture the basics of a mutable data-structure. The techniques we'll use to analyse them can be equally applied to the full range of mutable data types in the languages we are analysing: pairs, vectors, records, and so forth.

Note, however, that we are not considering side-effects to *variables* (*i.e.*, variable assignment). The techniques presented here could be extended to cover this semantic feature, but it isn't necessary to do so. For one, some higher-order languages, such as ML, do not allow side-effects to variables. Other languages, such as Scheme, do; but this is not a problem. Compiler front-ends that reduce Scheme programs to the CPS Scheme internal representation can transform the programs so that all variable side-effects are converted to operations on cells. This allows our semantics and analysis to assume that variable bindings are immutable, a convenient assumption — it allows the compiler to invoke more of the invariants of the lambda-calculus on the expressions in the internal representation.[2] This sort of transformation is called "assignment conversion," and is performed by the ORBIT Scheme compiler.

How must we alter our standard and control-flow semantics to add mutable cells? We need to introduce the set of addresses Addr to model cells. We can represent an address with just an integer. Addresses (cells) are first-class citizens, so we must include them into the set of run-time values D. Our interpreter now has to pass around a store $\sigma \in$ Store, which is modelled by a table mapping allocated addresses to values:

$$\begin{aligned} \mathrm{Addr} &= \mathcal{Z} \\ \mathrm{D} &= \mathrm{Bas} + \mathrm{Proc} + \mathrm{Addr} \\ \mathrm{Store} &= \mathrm{Addr} \rightharpoonup \mathrm{D} \end{aligned}$$

New addresses are allocated by the $alloc$ function, which is guaranteed to return a new address each time it is called.

The $\mathcal{C}$ and $\mathcal{F}$ functions pass the store around as they interpret the program, so they must be redefined to take an extra argument:

$$\begin{aligned} \mathcal{F}&\colon \mathrm{Proc} \to \mathrm{D}^* \to \mathrm{VEnv} \to \mathrm{Store} \to \mathrm{Ans} \\ \mathcal{C}&\colon \mathrm{CALL} \to \mathrm{BEnv} \to \mathrm{VEnv} \to \mathrm{Store} \to \mathrm{Ans} \end{aligned}$$

The only place the store is actually referenced or altered is inside the three primops `new`, `contents`, and `set`:

$$\begin{aligned} \mathcal{F}\,[\![\texttt{new}]\!]\ \langle d,\ k\rangle\ \ ve\ \sigma = {}& \textit{bad argument} \longrightarrow \text{error} \\ & \text{otherwise}\ \mathcal{F}\ k\ \ \langle a\rangle\ \ ve\ (\sigma[a \mapsto d]) \\ & \qquad \text{where}\ a = alloc \end{aligned}$$

[2]…and rewards programmers who use binding instead of assignment by producing tighter code — a bit of positive reinforcement that may appeal to compiler-writers with ideological agenda.

$$
\begin{aligned}
\mathcal{F}\,[\![\texttt{contents}]\!]\ \langle a,\, k\rangle\ \mathit{ve}\ \sigma &= \textit{bad argument}\ \longrightarrow\ \text{error} \\
&\quad\ \text{otherwise}\ \mathcal{F}\ k\ \langle \sigma\ a\rangle\ \mathit{ve}\ \sigma \\[1ex]
\mathcal{F}\,[\![\texttt{set}]\!]\ \langle a,\, d,\, k\rangle\ \mathit{ve}\ \sigma &= \textit{bad argument}\ \longrightarrow\ \text{error} \\
&\quad\ \text{otherwise}\ \mathcal{F}\ k\ \langle\rangle\ \mathit{ve}\ (\sigma[a \mapsto d])
\end{aligned}
$$

We can extend the exact control-flow semantics in a similar fashion.

Now we need to choose an abstraction for the store. Just as with contours, we need an abstraction that reduces the infinite numbers of distinct cells that could be allocated at run time to a finite abstraction.

For the purposes of doing control-flow analysis of languages like Scheme, tracking side effects is not of primary importance. Since Scheme makes variable binding convenient and cheap, side effects occur relatively infrequently. (This is especially true of well-written Scheme.) Loops, for instance, tend to update their iteration variables by rebinding them via tail-recursive calls, instead of by assigning them.

To further editorialise, I believe that the updating of loop variables is a task best left to loop packages such as Waters' series package ([Waters 82, Waters 89, Waters 89b] and [Steele 90, appendix A]) or the Yale Loop [Riesbeck], where the actual updating technique can be left to the macro writer to implement efficiently, and ignored by the application programmer.

For these reasons, we can afford to settle for a solution that is merely correct, without being very informative. Therefore, the abstraction we'll choose uses a very weak technique to deal with side effects: all addresses are merged together into a single abstract address. This means that once a procedure is stored into a cell during abstract interpretation, any fetch operation on any cell could potentially produce that procedure. We can represent the abstract store as the set of procedures that have been stashed out in the store. The `new` and `set` primops add procedures to this set; the `contents` primop produces the entire set.

$$
\begin{aligned}
\widehat{\mathrm{D}} &= \mathcal{P}(\widehat{\mathrm{Proc}}) \\
\widehat{\mathrm{Store}} &= \widehat{\mathrm{D}}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{\mathcal{F}}\,[\![p{:}\texttt{new}]\!]\ \left\langle \hat{d},\, \hat{k}\right\rangle\ \widehat{\mathit{ve}}\ \hat{\sigma} &= \left(\bigsqcup_{f\in\hat{k}} \widehat{\mathcal{F}}\ f\ \langle\emptyset\rangle\ \widehat{\mathit{ve}}\ (\hat{\sigma}\cup\hat{d})\right) \sqcup \left[\left\langle ic_{p:\texttt{new}},\, \hat{\beta}\right\rangle \mapsto \hat{k}\right] \\
&\quad \text{where } \hat{b} = \widehat{nb} \\
&\quad\phantom{\text{where }} \hat{\beta} = \left[p \mapsto \hat{b}\right] \\
\widehat{\mathcal{F}}\,[\![p{:}\texttt{contents}]\!]\ \left\langle \hat{a},\, \hat{k}\right\rangle\ \widehat{\mathit{ve}}\ \hat{\sigma} &= \left(\bigsqcup_{f\in\hat{k}} \widehat{\mathcal{F}}\ f\ \langle\hat{\sigma}\rangle\ \widehat{\mathit{ve}}\ \hat{\sigma}\right) \sqcup \left[\left\langle ic_{p:\texttt{contents}},\, \hat{\beta}\right\rangle \mapsto \hat{k}\right] \\
&\quad \text{where } \hat{b} = \widehat{nb} \\
&\quad\phantom{\text{where }} \hat{\beta} = \left[p \mapsto \hat{b}\right]
\end{aligned}
$$

$$\widehat{\mathcal{F}}\,[\![p\text{:}\mathtt{set}]\!]\,\left\langle \hat{a},\,\hat{d},\,\hat{k}\right\rangle\,\widehat{ve}\,\widehat{\sigma} = \left(\bigsqcup_{f\in\hat{k}} \widehat{\mathcal{F}}\,f\,\langle\rangle\,\widehat{ve}\,(\widehat{\sigma}\cup\hat{d})\right) \sqcup \left[\left\langle ic_{p:\mathtt{set}},\,\widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \hat{b} = \widehat{nb}$$
$$\widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

We can again choose more detailed abstractions if the programs being analysed require more a precise analysis of the store. An example is the abstraction used in Hudak's work on static deallocation of run-time data structures [Hudak 86b]. We have an address for each appearance of an allocation primop in our program. So if the `new` (or `cons`, or `make-vector`) primop appears five times in the program, we have five abstract addresses. Each occurrence of the primop allocates a different abstract address, and the store distinguishes values assigned to the different addresses.

If we choose a store abstraction that has multiple distinct addresses, then we must track addresses as they are passed around during abstract interpretation as well as procedures. This requires extending the $\widehat{\mathrm{D}}$ set in a straightforward way.

A further extension of the simple store abstraction would be to distinguish addresses that were allocated by a given occurrence of a primop in different abstract environment contexts. There are many possibilities here; I have not explored them in this dissertation.

3.8.2 External Procedures and Calls

The control-flow analysis we have developed is a "closed-world" analysis — it assumes that the entire program is presented to the analysis. However, most compilers support separate compilation and most Scheme and ML implementations support dynamic loading and incremental definition. This means that our analysis needs to take into account calls and lambdas that are external to the block of code being analysed.

Our abstract analysis can handle this by defining two special tokens: the external procedure *xproc*, and the external call *xcall*. The *xproc* represents unknown procedures that are passed into our program from the outside world at run time. The *xcall* represents calls to procedures that happen external to the program text. For example, suppose we have the following fragment in our program:

```
(foo (λ (k) (k 7)))
```

In our main development, we restricted our programs to have no free variables. If we relax this restriction, and allow `foo` to be free over the program, then in general we have no idea at compile time what procedures `foo` could be bound to at run time. The call to `foo` is essentially an escape from the program to the outside world, and we must record this fact. We do this by binding `foo` to *xproc* at analysis time. Further, since `(λ (k) (k 7))` is passed out of the program to external routines, it can be called from outside the program. We do this by recording the lambda in the set of lambdas that could be called from *xcall*.

Lambdas such as `(λ (k) (k 7))` in the above example that are passed to the external call have escaped to program text unavailable at compile time, and so can be called in arbitrary, unobservable ways. They have escaped close scrutiny, and in the face of limited information, we are forced simply to assume the worst. We maintain a set ESCAPED of escaped procedures, which initially contains *xproc* and the top-level lambda of the program. The rules for the external call, the external procedure and escaped functions are simple:

1. Any procedure passed to the external procedure escapes.
2. Any escaped procedure can be called from the external call.
3. When a procedure is called from the external call, it may be applied to any escaped procedure.

This provides very weak information, but for external procedures whose properties are completely unknown at compile time, it is the best we can do. (On the other hand, many external procedures, *e.g.* `print`, or `length`, have well known properties. For instance, both `print` and `length` only call their continuations. Thus, their continuations do not properly escape. Nor are their continuations passed escaped procedures as arguments. It is straightforward to extend the analysis presented here to utilise this stronger information.)

These rules can be easily integrated into our abstract semantics.

External procedures can potentially access the store, so these two semantic features interact. Suppose we choose the simple store abstraction discussed in the previous section.

- Any procedure placed in the abstract store can be accessed by external procedures that can reference the store.
- Therefore, any procedure placed in the abstract store can escape.
- Any escaping procedure can be placed in the abstract store by an external procedure.

So, given the simple store abstraction, it makes sense to identify the ESCAPED set and the abstract store set $\widehat{\sigma}$, since any element that is placed in one of these sets must also be placed in the other. (On the other hand, if we choose a more complex store abstraction, we would want to separate these two sets, since not all abstract addresses escape, and so not all elements of the abstract store are accessible by external procedures.)

3.8.3 Partitioning Procedures and Continuations

A worthwhile enhancement of the basic semantics is to distinguish user procedures from continuations.

CPS Scheme is an intermediate representation for full Scheme. In full Scheme, the user cannot write CPS-level continuations: all continuations, all variables bound to continuations, and all calls to continuations (*i.e.*, returns) are introduced by the CPS converter. This divides the procedural world into two halves: user procedures and continuations introduced by the CPS converter. It is easy for the CPS converter to mark these continuation lambdas, variables and call sites as it introduces them into the program. This partition is a powerful constraint on the sets propagated around by the analysis: a given call site either only calls user procedures, or only calls continuations; a given variable is either bound to only user procedures, or bound to only continuations. This partition holds throughout all details of the CFA semantics; exploiting it produces a much tighter analysis.

Distinguishing user procedures and continuations is useful for reasons besides improving the data-flow analysis; both ORBIT and Kelsey's transformational compiler do some form of this marking [Kranz$^+$ 86, Kelsey 89].

Note that this change doesn't alter the meaning of either user procedures or continuations — in CPS Scheme they both have the meaning of a continuation procedure. We are simply able to statically partition the procedural world into two disjoint sets, with consequent improvements in the precision and efficiency of our analysis. In this way, a CPS-based representation lets us "have it both ways": We can distinguish procedure calls and returns when it is useful to do so, and still enjoy the advantages of a unified representation and semantics, treating all lambdas identically when convenient.

The user-procedure/continuation partition interacts with the external call mechanisms. Instead of having a value *xproc* to represent external procedures, we need to provide both for external user-level procedures *xproc* and external continuations *xcont*. Similarly, we need to have both an external call *xcall* for calls to user-level procedures, and an external return *xret* to mark calls to continuations. The extensions to handle this are straightforward.

A criteria to separate user procedures and continuation procedures is whether or not the procedure takes a continuation argument itself. A user procedure is always called with an extra continuation argument to which it is to deliver its result. This implicit continuation is made an explicit parameter by the CPS conversion process, which introduces the extra argument into the procedure's formal parameter list and into all the calls to the procedure. A continuation, on the other hand, completely encapsulates the computational future of a process, so it requires no continuation argument itself.

It is an interesting feature of this partition that source-to-source transformations on a CPS Scheme program can convert a user procedure to a continuation and *vice versa*. For example, consider the following Scheme summation loop:

```
(λ (n)
  (letrec ((lp (λ (i sum)
                 (if (zero? i) sum (lp (- i 1) (+ sum i))))))
    (lp n 0)))
```

If we partially CPS-convert this loop, introducing continuation variables, calls, and lambdas, we get:

```
(λ (n k)
  (letrec ((lp (λ (i sum c)
                 (if (zero? i) (c sum)
                     (lp (- i 1) (+ sum i) c)))))
    (lp n 0 k)))
```

Copy propagation (section 10.1) will discover that `c` is always bound to the value of `k` — it is being passed around the loop needlessly. We can remove `c` from the program completely, and replace the call to `c` with one to `k`:

```
(λ (n k)
  (letrec ((lp (λ (i sum)
                 (if (zero? i) (k sum)
                     (lp (- i 1) (+ sum i))))))
    (lp n 0)))
```

Note that the `(λ (i sum) ...)` lambda has shifted status from a user procedure, which takes a continuation argument `k`, to a continuation itself. In other words, in the transformed code we do not pass `lp` a continuation to which it should return; `lp`'s definition already includes this information.

If a CPS-based compiler distinguishes user procedures from continuations, it must remember to update the status of altered lambdas after performing this type of source transformation.

Chapter 4

Control Flow II: Detailed Semantics

That which must be proved
cannot be worth much.
— fortune cookie,
by way of Robert Harper

The aim of the previous chapter was to develop a simple semantically-based definition of control-flow analysis, along with safe, computable abstractions for it. The development was kept simple for the sake of intuitive explication; the rigorous details of the necessary mathematical machinery were intentionally glossed over in order to get the "big picture" across.

Of course, one of the big attractions of defining an analysis with a denotational semantics is that we can then rigorously prove properties about our analysis. In this chapter, I will redevelop the semantics with enough precision and care to allow us to prove some important basic properties of the analysis. The major results will be to show that

- The equations defining the exact and abstract control-flow semantics have solutions.
- The abstract semantics safely approximates the exact semantics.
- The abstract semantics is computable.

This is what we need to show to put the computable control-flow analysis on a firm theoretical foundation. If you are uncomfortable with or unwilling to read detailed semantics proofs, you can simply take these three results on faith and skip to the next chapter.

The rest of the chapter follows this agenda:

1. First, we will change the simple equations given in the previous chapter to patch up some holes in the definitions. For example, we will add side-effects and a store, and restrict the domains of some functions to make them well-defined. (Section 4.1)

2. Next, we will show that the semantic functions ($\mathcal{PR}$, $\mathcal{A}$, $\mathcal{C}$, $\mathcal{F}$ and their abstract counterparts) exist. This requires showing that the recursive definitions are well-defined and have solutions. (Section 4.3)

$$
\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{Clo} &= \text{LAM} \times \text{BEnv} \\
\text{Proc} &= \text{Clo} + \text{PRIM} + \{stop\} \\
\text{Addr} &= \mathcal{Z} \\
\text{D} &= \text{Bas} + \text{Proc} + \text{Addr} \\
\text{Store} &= \text{Addr} \rightharpoonup \text{D} \\
\text{CN} &= (\textit{contours}) \\
\text{BEnv} &= \text{LAB} \rightharpoonup \text{CN} \\
\text{VEnv} &= (\text{VAR} \times \text{CN}) \rightharpoonup \text{D} \\
\text{CCtxt} &= \text{LAB} \times \text{BEnv} \\
\text{CCache}^1 &= \mathcal{P}(\text{CCtxt} \times \text{Proc}) \\
\text{CState} &\subset \text{CALL} \times \text{BEnv} \times \text{VEnv} \times \text{CN} \times \text{Store} \quad \text{(see figure 4.7)} \\
\text{FState} &\subset \text{Proc} \times \text{D}^* \times \text{VEnv} \times \text{CN} \times \text{Store} \quad \text{(see figure 4.7)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{PR}&: \text{PR} \to \text{CCache} \\
\mathcal{A}&: \text{ARG} \cup \text{FUN} \to \text{BEnv} \to \text{VEnv} \rightharpoonup \text{D} \\
\mathcal{K}&: \text{CONST} \to \text{Bas} \\
\mathcal{C}&: \text{CState} \to \text{CCache} \\
\mathcal{F}&: \text{FState} \to \text{CCache}
\end{aligned}
$$

Figure 4.1: Exact Semantics Domains and Functionalities

3. After establishing that all the mathematical structures we are dealing with actually exist, we can prove properties about them. In particular, we can prove the two main results of this chapter: the abstract semantics safely approximates the exact semantics, and the abstract semantics is computable. (Section 4.4)

4.1 Patching the Equations

For purposes of proving theorems, the simple equations developed in the previous chapter have several lapses. In this section, we will fix up the equations with enough precision and detail to make them useful in a formal context. The domain and function equations for the new semantics are collected together in figures 4.1–4.6.

[1]Later (Sec. 4.4) shown to be CCtxt $\rightharpoonup$ Proc

$$\mathcal{PR}\;\ell = \mathcal{F}\;\langle f,\, \langle stop\rangle,\, [\,],\, b_0,\, [\,]\rangle \cup \langle\langle c_{\text{top}},\, [\,]\rangle,\, f\rangle$$
$$\text{where } f = \mathcal{A}\,\ell\;[\,]\;[\,]$$

$$\mathcal{A}\;[\![k]\!]\;\beta\;ve = \mathcal{K}\;k$$
$$\mathcal{A}\;[\![prim]\!]\;\beta\;ve = prim$$
$$\mathcal{A}\;[\![v]\!]\;\beta\;ve = ve\;\langle v,\, \beta(binder\,v)\rangle$$
$$\mathcal{A}\;[\![\ell]\!]\;\beta\;ve = \langle \ell,\, \beta\rangle$$

$$\begin{aligned}
\mathcal{C}\;\langle[\![c{:}\texttt{(}f\;\;a_1\ldots a_n\texttt{)}]\!],\, \beta,\, ve,\, b,\, \sigma\rangle = {} & f' \notin \text{Proc} \;\longrightarrow\; [\,] \\
& \text{otherwise } \{\langle\langle c,\, \beta\rangle,\, f'\rangle\} \cup \mathcal{F}\;\langle f',\, av,\, ve,\, b',\, \sigma\rangle \\
& \text{where } f' = \mathcal{A}\;f\;\beta\;ve \\
& \qquad av{\downarrow}i = \mathcal{A}\;a_i\;\beta\;ve \\
& \qquad b' = nb\;b\;c
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}\;\langle[\![c{:}\texttt{(letrec ((}f_1\;\;l_1\texttt{)...)}\;\;c'\texttt{)}]\!],\, \beta,\, ve,\, b,\, \sigma\rangle = \mathcal{C}\;\langle c',\, \beta',\, ve',\, b',\, \sigma\rangle \\
&\quad \text{where } b' = nb\;b\;c \\
&\qquad\quad \beta' = \beta[c \mapsto b'] \\
&\qquad\quad ve' = ve[\langle f_i,\, b'\rangle \mapsto \mathcal{A}\;l_i\;\beta'\;ve]
\end{aligned}$$

Figure 4.2: Exact Semantic Functions (a)

4.1.1 Contour Allocation

Whenever control enters a lambda, a new contour is allocated to bind its variables to their arguments. In the previous chapter, we swept this contour allocation under the rug with the nb "gensym" function that returned a new contour each time it was called. Such a function is obviously not well-defined. In the detailed semantics, we add a "contour counter" argument to the $\mathcal{F}$ and $\mathcal{C}$ functions. This value is a contour that is an upper bound for all the currently allocated contours. The nb function is defined to increment this counter. Whenever a new contour is needed, the current contour counter is used, and nb called to generate the new bound.

Using terms like "counter" and "upper bound" means the contour set CN must be a partial order and nb must be an increasing function. Sequentially applying nb to successive contours is essentially moving up chains in the partial order of CN. This gives us that if the current contour b is $\geq$ all allocated contours, then $nb\,b$ is $>$ all allocated contours. This is necessary for the exact semantics (both standard and control flow) to be a correct specification of CPS Scheme. We'll also assume there is designated initial contour b_0 that is used by the $\mathcal{PR}$ function to start the interpretation.

The $\mathcal{C}$ and $\mathcal{F}$ functions adhere to the following invariant as they pass around the contour counter:

$\mathcal{F}\ \langle\langle[\![\ell\texttt{:(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ c\texttt{)}]\!],\ \beta\rangle,\ av,\ ve,\ b,\ \sigma\rangle =$
$\quad$ length $av \neq n \ \longrightarrow\ [\,]$
$\quad$ otherwise $\mathcal{C}\ \langle c,\ \beta[\ell \mapsto b],\ ve[\langle v_i,\ b\rangle \mapsto av{\downarrow}i],\ b,\ \sigma\rangle$

$\mathcal{F}\ \langle[\![p\texttt{:+}]\!],\ \langle x,\ y,\ k\rangle,\ ve,\ b,\ \sigma\rangle =$
$\quad$ *bad argument* $\longrightarrow\ [\,]$
$\quad$ otherwise $\left\{\left\langle\left\langle ic_{p:\texttt{+}},\ \beta\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}\ \left\langle k,\ \langle x+y\rangle,\ ve,\ nb\, b\, ic_{p:\texttt{+}},\ \sigma\right\rangle$
$\quad\quad$ where $\beta = [p \mapsto b]$

$\mathcal{F}\ \langle[\![p\texttt{:if}]\!],\ \langle x,\ k_0,\ k_1\rangle,\ ve,\ b,\ \sigma\rangle =$
$\quad$ $\{k_0, k_1\} \not\subset$ Proc $\longrightarrow\ [\,]$
$\quad$ $x \neq$ false $\longrightarrow\ \left\{\left\langle\left\langle ic^0_{p:\texttt{if}},\ \beta\right\rangle,\ k_0\right\rangle\right\} \cup \mathcal{F}\ \left\langle k_0,\ \langle\rangle,\ ve,\ nb\, b\, ic^0_{p:\texttt{if}},\ \sigma\right\rangle$
$\quad$ otherwise $\left\{\left\langle\left\langle ic^1_{p:\texttt{if}},\ \beta\right\rangle,\ k_1\right\rangle\right\} \cup \mathcal{F}\ \left\langle k_1,\ \langle\rangle,\ ve,\ nb\, b\, ic^1_{p:\texttt{if}},\ \sigma\right\rangle$
$\quad$ where $\beta = [p \mapsto b]$

$\mathcal{F}\ \langle stop,\ av,\ ve,\ b,\ \sigma\rangle = \emptyset$

$\mathcal{F}\ \langle[\![p\texttt{:new}]\!],\ \langle d,\ k\rangle,\ ve,\ b,\ \sigma\rangle =$
$\quad$ $k \notin$ Proc $\longrightarrow\ [\,]$
$\quad$ otherwise $\left\{\left\langle\left\langle ic_{p:\texttt{new}},\ \beta\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}\ \left\langle k,\ \langle a\rangle,\ ve,\ nb\, b\, ic_{p:\texttt{new}},\ \sigma[a \mapsto d]\right\rangle$
$\quad\quad$ where $\beta = [p \mapsto b]$
$\quad\quad\quad$ $a = alloc\, \sigma$

$\mathcal{F}\ \langle[\![p\texttt{:contents}]\!],\ \langle a,\ k\rangle,\ ve,\ b,\ \sigma\rangle =$
$\quad$ *bad argument* $\longrightarrow\ [\,]$
$\quad$ otherwise $\left\{\left\langle\left\langle ic_{p:\texttt{contents}},\ \beta\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}\ \left\langle k,\ \langle \sigma\, a\rangle,\ ve,\ nb\, b\, ic_{p:\texttt{contents}},\ \sigma\right\rangle$
$\quad\quad$ where $\beta = [p \mapsto b]$

$\mathcal{F}\ \langle[\![p\texttt{:set}]\!],\ \langle a,\ d,\ k\rangle,\ ve,\ b,\ \sigma\rangle =$
$\quad$ *bad argument* $\longrightarrow\ [\,]$
$\quad$ otherwise $\left\{\left\langle\left\langle ic_{p:\texttt{set}},\ \beta\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}\ \left\langle k,\ \langle\rangle,\ ve,\ nb\, b\, ic_{p:\texttt{set}},\ \sigma[a \mapsto d]\right\rangle$
$\quad\quad$ where $\beta = [p \mapsto b]$

Figure 4.3: Exact Semantic Functions (b)

$$
\begin{aligned}
\widehat{\text{Clo}} &= \text{LAM} \times \widehat{\text{BEnv}} \\
\widehat{\text{Proc}} &= \widehat{\text{Clo}} + \text{PRIM} + \{stop\} \\
\widehat{\text{D}} &= \mathcal{P}(\widehat{\text{Proc}}) \\
\widehat{\text{Store}} &= \mathcal{P}(\widehat{\text{Proc}}) \\
\widehat{\text{CN}} &= \text{LAB} \qquad \text{(for 1CFA)} \\
& \quad\ \{1\} \qquad \text{(for 0CFA)} \\
\widehat{\text{BEnv}} &= \text{LAB} \rightharpoonup \widehat{\text{CN}} \\
\widehat{\text{VEnv}} &= (\text{VAR} \times \widehat{\text{CN}}) \rightarrow \widehat{\text{D}} \\
\widehat{\text{CCtxt}} &= \text{LAB} \times \widehat{\text{BEnv}} \\
\widehat{\text{CCache}} &= (\text{LAB} \times \widehat{\text{BEnv}}) \rightarrow \widehat{\text{D}} \\
\widehat{\text{CState}} &\subset \text{CALL} \times \widehat{\text{BEnv}} \times \widehat{\text{VEnv}} \times \widehat{\text{CN}} \times \widehat{\text{Store}} \quad \text{(see figure 4.8)} \\
\widehat{\text{FState}} &\subset \widehat{\text{Proc}} \times \widehat{\text{D}}^* \times \widehat{\text{VEnv}} \times \widehat{\text{CN}} \times \widehat{\text{Store}} \quad \text{(see figure 4.8)}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{\mathcal{PR}}&\colon \text{PR} \rightarrow \widehat{\text{CCache}} \\
\widehat{\mathcal{A}}&\colon \text{ARG} \cup \text{FUN} \rightarrow \widehat{\text{BEnv}} \rightarrow \widehat{\text{VEnv}} \rightharpoonup \widehat{\text{D}} \\
\widehat{\mathcal{C}}&\colon \widehat{\text{CState}} \rightarrow \widehat{\text{CCache}} \\
\widehat{\mathcal{F}}&\colon \widehat{\text{FState}} \rightarrow \widehat{\text{CCache}}
\end{aligned}
$$

Figure 4.4: Abstract Semantics Domains and Functionalities

- The contour passed to $\mathcal{C}$ is the greatest contour that has been used to date.

- The contour passed to $\mathcal{F}$ is a contour that has never been used.

This particular convention is the right one for the eventual 1CFA abstraction we are going to make.

We could define nb as a simple function mapping contours to contours, with functionality $\text{CN} \rightarrow \text{CN}$. However, when we abstract the semantics, we'll be mapping these contours to abstract contours that can contain information about the dynamic behaviour of the program. For example, in 1CFA, the abstract contour allocated for entering a given lambda is the call from which we entered the lambda. It will help in establishing a connection between the exact and abstract semantics if we build this information into the exact contour from the beginning. We can do this by providing extra information to the nb function:

$$b_{i+1} = nb\ b_i\ x.$$

$$\widehat{\mathcal{PR}}\ \ell = \widehat{\mathcal{F}}\ \langle f,\ \langle\{stop\}\rangle,\ [\,],\ |b_0|,\ [\,]\rangle \sqcup [\langle c_{\text{top}},\ [\,]\rangle \mapsto \{f\}]$$
$$\text{where } \{f\} = \widehat{\mathcal{A}}\,\ell\ [\,]\ [\,]$$

$$\begin{aligned}
\widehat{\mathcal{A}}\,[\![k]\!]\ \widehat{\beta}\ \widehat{ve} &= \emptyset \\
\widehat{\mathcal{A}}\,[\![prim]\!]\ \widehat{\beta}\ \widehat{ve} &= \{prim\} \\
\widehat{\mathcal{A}}\,[\![\ell]\!]\ \widehat{\beta}\ \widehat{ve} &= \left\{\left\langle \ell,\ \widehat{\beta}\right\rangle\right\} \\
\widehat{\mathcal{A}}\,[\![v]\!]\ \widehat{\beta}\ \widehat{ve} &= \widehat{ve}\ \left\langle v,\ \widehat{\beta}(binder\, v)\right\rangle
\end{aligned}$$

$$\widehat{\mathcal{C}}\ \left\langle [\![c\!:\!\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!],\ \widehat{\beta},\ \widehat{ve},\ \hat{b},\ \widehat{\sigma}\right\rangle = \left(\bigsqcup_{f' \in F} \widehat{\mathcal{F}}\ \left\langle f',\ \widehat{av},\ \widehat{ve},\ \hat{b}',\ \widehat{\sigma}\right\rangle\right) \sqcup \left[\left\langle c,\ \widehat{\beta}\right\rangle \mapsto F\right]$$
$$\begin{aligned}
\text{where } F &= \widehat{\mathcal{A}}\, f\ \widehat{\beta}\ \widehat{ve} \\
\widehat{av}{\downarrow}i &= \widehat{\mathcal{A}}\, a_i\ \widehat{\beta}\ \widehat{ve} \\
\hat{b}' &= \widehat{nb}\ \hat{b}\, c
\end{aligned}$$

$$\widehat{\mathcal{C}}\ \left\langle [\![c\!:\!\texttt{(letrec ((}f_1\ \ l_1\texttt{)...)}\ \ c'\texttt{)}]\!],\ \widehat{\beta},\ \widehat{ve},\ \hat{b},\ \widehat{\sigma}\right\rangle = \widehat{\mathcal{C}}\ \left\langle c',\ \widehat{\beta}',\ \widehat{ve}',\ \hat{b}',\ \widehat{\sigma}\right\rangle$$
$$\text{where } \hat{b}' = \widehat{nb}\ \hat{b}\, c \qquad \widehat{\beta}' = \widehat{\beta}\left[c \mapsto \hat{b}'\right]$$
$$\widehat{ve}' = \widehat{ve} \sqcup \left[\left\langle f_i,\ \hat{b}'\right\rangle \mapsto \widehat{\mathcal{A}}\, l_i\ \widehat{\beta}'\ \widehat{ve}\right]$$

Figure 4.5: Abstract Semantics Functions (a)

This extra information x could be anything from the CState or FState argument that might be useful for abstraction. In 1CFA, this extra information is the label of the current call site, so we'll go ahead and build this into the functionality of nb:

$$nb\colon \text{CN} \rightarrow \text{LAB} \rightarrow \text{CN}.$$

Remember that this issue of adding extra information is irrelevant to the correctness of the exact semantics. The only thing that is important to guarantee correctness of the exact semantics is that the nb function behave as a counter gensym — that is, $b' \leq b \ \Rightarrow\ b' < nb\ b\, x$. Adding the extra information, of whatever type, is purely to facilitate establishing a connection between the exact semantics and useful abstractions. We'll return to this issue in detail in section 4.2.

4.1.2 Side Effects

Although side-effects were discussed in section 3.8.1, they were omitted from the equations in the main development of chapter 3 to keep the presentation simple. However, if we are going to prove theorems, we should prove them about the full-blown semantics that

$$\widehat{\mathcal{F}}\ \left\langle \left\langle [\![\ell\mathtt{:}(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!],\ \widehat{\beta} \right\rangle,\ \widehat{av},\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\text{length } \widehat{av} \neq n \ \longrightarrow\ [\,]$$
$$\text{otherwise } \widehat{\mathcal{C}}\ \left\langle c,\ \widehat{\beta}\left[\ell \mapsto \hat{b}\right],\ \widehat{ve} \sqcup \left[\left\langle v_i,\ \hat{b}\right\rangle \mapsto \widehat{av}{\downarrow}i\right],\ \hat{b},\ \widehat{\sigma} \right\rangle$$

$$\widehat{\mathcal{F}}\ \left\langle [\![p\mathtt{:+}]\!],\ \left\langle \hat{x},\ \hat{y},\ \hat{k} \right\rangle,\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle f,\ \langle \emptyset \rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic_{p\mathtt{:+}},\ \widehat{\sigma} \right\rangle\right) \sqcup \left[\left\langle ic_{p\mathtt{:+}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ \left\langle [\![p\mathtt{:if}]\!],\ \left\langle \hat{x},\ \hat{k}_0,\ \hat{k}_1 \right\rangle,\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\left(\bigsqcup_{f \in \hat{k}_0} \widehat{\mathcal{F}}\ \left\langle f,\ \langle\rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic^0_{p\mathtt{:if}},\ \widehat{\sigma} \right\rangle\right) \sqcup \left(\bigsqcup_{f \in \hat{k}_1} \widehat{\mathcal{F}}\ \left\langle f,\ \langle\rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic^1_{p\mathtt{:if}},\ \widehat{\sigma} \right\rangle\right)$$
$$\sqcup \left[\left\langle ic^0_{p\mathtt{:if}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}_0,\ \left\langle ic^1_{p\mathtt{:if}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}_1\right]$$
$$\text{where } \widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ \left\langle stop,\ \widehat{av},\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle = [\,]$$

$$\widehat{\mathcal{F}}\ \left\langle [\![p\mathtt{:new}]\!],\ \left\langle \hat{d},\ \hat{k} \right\rangle,\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle f,\ \langle \emptyset \rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic_{p\mathtt{:new}},\ \widehat{\sigma} \sqcup \hat{d} \right\rangle\right) \sqcup \left[\left\langle ic_{p\mathtt{:new}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ \left\langle [\![p\mathtt{:contents}]\!],\ \left\langle \hat{a},\ \hat{k} \right\rangle,\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle f,\ \langle \widehat{\sigma} \rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic_{p\mathtt{:contents}},\ \widehat{\sigma} \right\rangle\right) \sqcup \left[\left\langle ic_{p\mathtt{:contents}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

$$\widehat{\mathcal{F}}\ \left\langle [\![p\mathtt{:set}]\!],\ \left\langle \hat{a},\ \hat{d},\ \hat{k} \right\rangle,\ \widehat{ve},\ \hat{b},\ \widehat{\sigma} \right\rangle =$$
$$\left(\bigsqcup_{f \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle f,\ \langle\rangle,\ \widehat{ve},\ \widehat{nb}\ \hat{b}\ ic_{p\mathtt{:set}},\ \widehat{\sigma} \sqcup \hat{d} \right\rangle\right) \sqcup \left[\left\langle ic_{p\mathtt{:set}},\ \widehat{\beta}\right\rangle \mapsto \hat{k}\right]$$
$$\text{where } \widehat{\beta} = \left[p \mapsto \hat{b}\right]$$

Figure 4.6: Abstract Semantics Functions (b)

a real CPS-based compiler for Scheme or ML would have to deal with, and this means side-effects.

We can bring over the machinery developed in section 3.8.1 directly. The only detail not dealt with in section 3.8.1 is how unused addresses are allocated by the $alloc$ function. The $alloc$ function can pick an unused address either by incrementing some external counter that bounds the set of currently allocated addresses, or by simply choosing an address that is not in the domain of the partial function which is the current store. The second choice requires less machinery, so it is the one we'll adopt:

$$alloc\colon \text{Store} \rightarrow \text{Addr}.$$

The critical property we require of $alloc$ is that it allocate an unused address:

$$alloc\,\sigma \notin Dom(\sigma).$$

As discussed in section 3.8.1, there are several possibilities for store abstractions. The one we'll use is the simplest: all exact addresses are abstracted to a single abstract address; an abstract store is just the set of values that have been placed into the store.

4.1.3 Restricting the Function Domains

The $\mathcal{C}$ function takes five (curried) arguments: $\mathcal{C}\,c\,\beta\,ve\,b\,\sigma$. These are the call expression, the contour environment, the variable environment, the contour counter, and the store. However, we cannot just apply $\mathcal{C}$ to any five arguments chosen arbitrarily from the corresponding sets. The five arguments are all delicately intertwined and interdependent.

For example, we cannot evaluate the call $c = [\![\texttt{(f 3)}]\!]$ in an empty environment $\beta = [\,], ve = [\,]$ — the variable lookup $\mathcal{A}\,[\![\texttt{f}]\!]\,\beta\,ve$ would not be defined. In order to successfully perform variable references while evaluating a call, the three arguments c, β, and ve must be consistent with each other.

Definition 1 *c, β, ve are* consistent *in program P if*

- *Dom(β) is c's lexical context: for every lambda or* `letrec` *ℓ that is lexically superior to c (*i.e.*, c appears in the lexical scope of ℓ), $\ell \in$ Dom(β).*
- *ve covers β: $\forall \ell \in \text{Dom}(\beta)\ \ \forall v \ni \textbf{binder}\,v = \ell\ \ \langle v,\, \beta\ell\rangle \in \text{Dom}(ve)$.*
  That is, all variable lookups that can be done through β are defined by ve.

This definition of *consistent* ensures that all variable lookups that could be performed while evaluating c's subexpressions will be handled by the environment β, ve. We can extend the definition of *consistent* to closures with the following:

Definition 2 *$\langle \ell,\, \beta\rangle$, ve are* consistent *in program P if*

- *Dom*(β) *is* ℓ*'s lexical context: for every lambda or* `letrec` ℓ' *that is lexically superior to* ℓ, $\ell' \in$ *Dom*(β).
- *ve covers* β*:* $\forall \ell' \in \mathit{Dom}(\beta)\ \ \forall v \ni \mathit{binder}\, v = l' \quad \langle v,\ \beta\ell'\rangle \in \mathit{Dom}(ve)$.

This guarantees us that free variable references occurring in the lambda's internal call expression must be defined by the current environment.

The idea of consistency should show that our semantic functions are only well-defined when applied to particular combinations of arguments. Our semantic functions, as defined in the previous chapter, have function domains that allow them to be applied to inconsistent arguments. As such, they are not well-defined functions.

Notice that this problem is just a superficial issue of structure. When a syntactically correct top-level program is evaluated in the initial environment, the chain of recursive $\mathcal{C}$ and $\mathcal{F}$ applications that arise always have arguments that are consistent. This is no accident, of course — the equations are carefully designed to do this. Environments are always constructed before dependent variables are referenced, addresses are always allocated before they are dereferenced, and so forth. In short, the illegal argument tuples are ones that never occur during evaluation of a syntactically legal CPS Scheme program. In this sense, these constraints on legal argument values to the semantic functions are a form of invariant on the semantics — constraints that are satisfied at all times during program evaluation.

There are a few ways to make the equations well-defined. First, we could take the semantic functions to be partial functions rather than total functions; the illegal argument combinations would be removed from the function domains. The problem with this approach is that we will be dealing with limits of sequences drawn from these spaces of partial functions. The particulars of defining the ordering relations and limits on these structures complicates the proofs unnecessarily.

Alternatively, we could add an extra "static error" value to the range of the semantic functions. When a semantic function is applied to a bad argument tuple, it could return the static error value. This would make the semantic functions well-defined over their complete domains, and we could then subsequently prove that the static error value is never returned from the top-level application of the $\mathcal{C}$ and $\mathcal{F}$ functions. Adding such a static error value, however, complicates the definition of the Ans and CCache domains. Proving that the resulting recursive domain equations have solutions is messy, which is unfortunate considering that the actual value we must work so hard to add to the set is never actually used by the equations at all. Furthermore, requiring the semantic functions to check their arguments for correctness further complicates their definitions.

The approach we will follow here is to restrict the domains of the semantic functions. We'll define the domain of the semantic functions to be only those combinations of values that are consistent. This will make the semantic functions well-defined total functions.

Consider the five arguments that $\mathcal{C}$ is applied to: c, β, ve, b, σ. We want to take the domain of $\mathcal{C}$ to be a subset of CALL $\times$ BEnv $\times$ VEnv $\times$ CN $\times$ Store — only those values that satisfy the consistency constraints will be included in the subset. To appeal to the

A tuple $\langle c,\ \beta,\ ve,\ b,\ \sigma\rangle \in$ CALL $\times$ BEnv $\times$ VEnv $\times$ CN $\times$ Store is in the set CState iff

1. All environment contexts in the system are consistent: **[C-ENV]**
   (a) c, β, ve are consistent
   (b) $\forall \langle \ell,\ \beta\rangle \in (\mathit{Im}(ve) \cup \mathit{Im}(\sigma)) \cap$ Clo $\langle \ell,\ \beta\rangle$, ve are consistent.
2. All addresses in the system are defined by the current store: **[C-ADDR]**
   $(\mathit{Im}(ve) \cup \mathit{Im}(\sigma)) \cap \text{Addr} \subset \mathit{Dom}(\sigma)$.

A tuple $\langle f,\ av,\ ve,\ b,\ \sigma\rangle \in$ Proc $\times$ D* $\times$ VEnv $\times$ CN $\times$ Store is in the set FState iff

1. All environment contexts in the system are consistent: **[F-ENV]**
   $\forall \langle \ell,\ \beta\rangle \in (\{f\} \cup av \cup \mathit{Im}(ve) \cup \mathit{Im}(\sigma)) \cap$ Clo $\langle \ell,\ \beta\rangle$, ve are consistent.
   (where the argument vector av is taken as a simple set).
2. All addresses in the system are defined by the current store: **[F-ADDR]**
   $(av \cup \mathit{Im}(ve) \cup \mathit{Im}(\sigma)) \cap \text{Addr} \subset \mathit{Dom}(\sigma)$
   (where the argument vector av is taken as a simple set).
3. Primops applications have correct arity: **[F-ARITY]**
   $f \in$ PRIM $\quad\Rightarrow\quad$ length $av =$ arity of f.

Figure 4.7: Function domains for $\mathcal{C}$ and $\mathcal{F}$.

notion of the semantic functions giving a functional interpreter for CPS Scheme, a 5-tuple $\langle c,\ \beta,\ ve,\ b,\ \sigma\rangle$ can be viewed as an interpreter state. Let's call the set of legal interpreter states for the $\mathcal{C}$ function CState. Similarly, the subset of Proc $\times$ D* $\times$ VEnv $\times$ CN $\times$ Store that are legitimate argument tuples for the $\mathcal{F}$ function will be called FState.

The relationships that these argument tuples must satisfy to be legitimate are given in figure 4.7. C-ENVa requires the call context $c/\beta/ve$ that $\mathcal{C}$ is evaluating to be consistent, so all variable references are well-defined. C-ENVb requires all the other evaluation contexts that exist in the system (in the environment $\mathit{Im}(ve)$, or the store $\mathit{Im}(\sigma)$) to be well-defined. C-ADDR requires every address in the system to be defined by the store. This means that any address dereference will be well-defined. F-ENV and F-ADDR are similar constraints for the argument tuples in the domain of $\mathcal{F}$. The third $\mathcal{F}$ constraint, F-ARITY, requires primop applications to be of the correct arity, so that the semantic equations won't have to perform this check themselves.

We could redefine the $\mathcal{A}$ function by similarly restricting its domain. Since $\mathcal{A}$ is not defined recursively, we do not need to invoke all the fixed-point machinery to prove its existence and properties. So we can take the simpler approach of declaring $\mathcal{A}$ to be a partial function, defined only when its arguments are consistent. Note that the domain restriction on $\mathcal{C}$ implies that applications of $\mathcal{A}$ in $\mathcal{C}$ are always well-defined.

The $\mathcal{C}$ and $\mathcal{F}$ functions are defined recursively, so we need to ensure that if one of these functions is applied to an argument in CState or FState, that the argument tuples used in the recursive applications are also in CState and FState. This is necessary to show that the recursive definitions are legitimate. We'll return to this issue when we prove the existence of the $\mathcal{C}$ and $\mathcal{F}$ functions (section 4.3).

The domains of abstract functions $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are much easier to restrict, since there are much fewer ways to go wrong in the abstract semantics. Since the abstract variable environment $\widehat{ve}$ is a total function to power sets, the operation of looking up a variable/abstract-contour pair $\langle v,\, \hat{b}\rangle$ is always well-defined. This removes half of the C-ENV and F-ENV constraints. Addresses have been eliminated from the abstract semantics entirely, so the issue of which addresses are defined by the current store doesn't arise (constraints C-ADDR and F-ADDR). The only items that must be checked for are (1) contour lookups in $\widehat{\beta}$ must be defined and (2) primop applications must have the correct arity. This is summarised in figure 4.8.

A tuple $\left\langle c,\, \widehat{\beta},\, \widehat{ve},\, \hat{b},\, \widehat{\sigma}\right\rangle \in \text{CALL} \times \widehat{\text{BEnv}} \times \widehat{\text{VEnv}} \times \widehat{\text{CN}} \times \widehat{\text{Store}}$ is in the set $\widehat{\text{CState}}$ iff

1. All contour environment contexts in the system are consistent: **[C-ENV']**
   (a) $Dom(\widehat{\beta})$ is c's lexical context.
   (b) $\forall \left\langle \ell,\, \widehat{\beta}\right\rangle \in (\bigcup Im(\widehat{ve}) \cup \widehat{\sigma}) \cap \widehat{\text{Clo}} \quad Dom(\widehat{\beta})$ is ℓ's lexical context.

A tuple $\left\langle \hat{f},\, \widehat{av},\, \widehat{ve},\, \hat{b},\, \widehat{\sigma}\right\rangle \in \widehat{\text{Proc}} \times \hat{\text{D}}^* \times \widehat{\text{VEnv}} \times \widehat{\text{CN}} \times \widehat{\text{Store}}$ is in the set $\widehat{\text{FState}}$ iff

1. All environment contexts in the system are consistent: **[F-ENV']**
   $\forall \left\langle \ell,\, \widehat{\beta}\right\rangle \in \left(\left\{\hat{f}\right\} \cup (\bigcup \widehat{av}) \cup (\bigcup Im(\widehat{ve})) \cup \widehat{\sigma}\right) \cap \widehat{\text{Clo}} \quad Dom(\widehat{\beta})$ is ℓ's lexical context.
   (where the argument vector $\widehat{av}$ is taken as a simple set).
2. Primops applications have correct arity: **[F-ARITY']**
   $\hat{f} \in \text{PRIM} \quad \Rightarrow \quad \text{length } \widehat{av} = \text{arity of } \hat{f}$.

Figure 4.8: Function domains for $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$.

4.1.4 Redefining CCache

In the previous chapter, we defined an exact call cache γ to be a partial function from a call context (that is, a call-site/contour-environment pair) to a procedure: CCtxt $\rightharpoonup$ Proc. In the old semantics, we updated the call cache with an expression of the form $\gamma[cc \mapsto f]$.

An update of this form has the potential to delete information from the final call cache: the update $[cc \mapsto f]$ could shadow a previous entry for cc in γ. What we want is an update that is *increasing* — that never shadows old information in the call cache.

In this chapter, we define a call cache to be a relation on CCtxt $\times$ Proc, that is, a set of $\langle cc,\, f\rangle$ pairs. So the update now has the form $\gamma \cup \{\langle cc,\, f\rangle\}$, which is obviously increasing.

With this definition, we can be confident that our semantics definition isn't accidentally deleting any entries after adding them to the call cache, and so the result call cache contains every call that was made during program execution.

It is, however, notationally convenient to manipulate call caches in functional ways. Furthermore, our chapter 3 intuitions are, in fact, correct: the call caches we construct with the relational update have the structure of a partial function. So, the development in this chapter will proceed as follows:

- First, we'll define the exact semantics using the relational update, clearly establishing correctness.
- Next, we'll show that the semantic functions defined this way exist.
- Finally, we'll show that the call cache produced by these functions is a partial function. That is,
$$\langle cc,\, f\rangle, \langle cc,\, f'\rangle \in \gamma \Rightarrow f = f'.$$

After we've done this, we can revert to treating exact call caches as partial functions whenever it is convenient to do so.

4.1.5 Order Relations and Join Operations

Table 4.9 collects the definitions of $\sqsubseteq$ and $\sqcup$ for the domains using these operations in the semantics equations. These definitions are all the obvious ones; they are collected together into a single table simply for reference purposes. Note that if a domain is not also a lattice,

CCache	$\subset$	$\cup$
$\widehat{\text{CCache}}$	pointwise $\subset$	pointwise $\cup$
$\widehat{\text{D}}$	$\subset$	$\cup$
$\widehat{\text{VEnv}}$	pointwise $\subset$	pointwise $\cup$
$\widehat{\text{Store}}$	$\subset$	$\cup$
$\widehat{\text{D}}^*$	elementwise $\subset$	elementwise $\cup$
CPO	$\sqsubseteq$	$\sqcup$

Table 4.9: Order and join definitions

then its $\sqcup$ operation is only defined on elements that are ordered. Consider $\widehat{\text{D}}^*$ for example. A two-element argument vector $\left\langle \hat{d},\, \hat{e}\right\rangle$ and a three-element argument vector $\left\langle \hat{f},\, \hat{g},\, \hat{h}\right\rangle$ are not ordered with respect to each other, and so $\left\langle \hat{d},\, \hat{e}\right\rangle \sqcup \left\langle \hat{f},\, \hat{g},\, \hat{h}\right\rangle$ is not defined.

$$
\begin{array}{lllll}
\text{Proc} \to \widehat{\text{Proc}} & |prim| = prim & |stop| = stop & |\langle \ell,\, \beta \rangle| = \langle \ell,\, |\beta| \rangle \\
\text{D} \to \widehat{\text{D}} & |b| = |a| = \emptyset & |f| = \{|f|_{\text{Proc}}\} & \\
\text{D}^* \to \widehat{\text{D}}^* & |\langle d_1, \ldots, d_n \rangle| = \langle |d_1|, \ldots, |d_n| \rangle & & \\
\text{BEnv} \to \widehat{\text{BEnv}} & |\beta| = |\cdot| \circ \beta & & \\
 & |\beta|\ \ell = |\beta\, \ell| & & \\
\text{CN} \to \widehat{\text{CN}} & |b| = 1 & \textit{(0CFA)} & \\
 & |c_0 \cdots c_n| = c_n & \textit{(1CFA)} & \\
\text{VEnv} \to \widehat{\text{VEnv}} & |ve| = \lambda \left\langle v,\, \hat{b} \right\rangle . \displaystyle\bigsqcup_{|b| = \hat{b}} |ve\ \langle v,\, b \rangle| & & \\
\text{Store} \to \widehat{\text{Store}} & |\sigma| = \bigsqcup |Im(\sigma)| & & \\
\text{CCtxt} \to \widehat{\text{CCtxt}} & |\langle c,\, \beta \rangle| = \langle c,\, |\beta| \rangle & & \\
\text{CCache} \to \widehat{\text{CCache}} & |\gamma| = \lambda \widehat{cc}. \displaystyle\bigsqcup_{|cc| = \widehat{cc}} |\gamma\ cc| & &
\end{array}
$$

Figure 4.10: Abstraction functions

4.1.6 Result Semantics

The new, detailed semantics we get after applying the changes of the previous two subsections is collected in figures 4.1–4.6. Figure 4.1 gives the semantic domains and functionalities for the exact control-flow semantics; figures 4.2 and 4.3 give the definitions of the $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{F}$ functions; figure 4.4 gives the domains and functionalities for the abstract semantics; figures 4.5 and 4.6 give the definitions of the $\widehat{\mathcal{A}}$, $\widehat{\mathcal{C}}$, and $\widehat{\mathcal{F}}$ functions.

4.2 Abstraction Functions

The connection between an exact and an abstract semantics is specified by an *abstraction function*. An abstraction function maps an exact value x to its abstract counterpart $|x|$. The abstraction function for a set S is written $|\cdot|_S$, and has functionality $|\cdot|_S\colon S \to \hat{S}$. We will drop the subscript and just write $|\cdot|$ whenever convenient. An abstraction function precisely specifies how we "throw away" information to arrive at our abstract semantics.

The abstraction functions for the control-flow semantics are given in figure 4.10.

Procedures Primops and the stop continuation are abstracted to themselves. A closure $\langle \ell,\, \beta \rangle$ is abstracted by forming an abstract closure $\langle \ell,\, |\beta| \rangle$, with lambda ℓ and abstract contour environment $|\beta|$.

Values The abstract semantics only tracks procedural values[2] — abstract values are sets of

[2]Certain analyses — for instance, constant propagation — might want to track non-procedural values, as

abstract procedures from $\widehat{\text{Proc}}$. So booleans and addresses, which obviously are not procedures, abstract to the empty set: $|b| = |a| = \emptyset$. A procedure f abstracts to the singleton set containing its abstract counterpart: $\{|f|_{\text{Proc}}\}$.

Argument Vectors An argument vector of values is simply abstracted elementwise: $|\langle d_1, \ldots, d_n \rangle| = \langle |d_1|, \ldots, |d_n| \rangle$.

Contour Environments An exact contour environment β is abstracted by composing it with the contour abstraction function.

Contours The contour abstraction is the critical one that determines most of the other abstractions.

0CFA In the 0CFA analysis (section 3.6), the abstract semantics has only one abstract contour, 1. So the 0CFA abstraction is trivial: all contours abstract to 1.

In 0CFA, the exact contour set can simply be taken to be integers, CN $= \mathcal{Z}$; and the contour allocation function just increments the current contour: $nb\ b\ c = b + 1$. Note that nb ignores its extra argument c — this is because the eventual abstraction doesn't need any information at all, so we don't need to build it in to the exact contour. The initial "seed" contour b_0 is 0. The abstract contour set is the unity set: $\widehat{\text{CN}} = \{1\}$. The contour allocation function always returns 1: $\widehat{nb}\ \hat{b}\ c = 1$. The abstraction function abstracts all contours to 1: $|b| = 1$.

If we use these definitions for CN, b_0, nb, $\widehat{\text{CN}}$, $\widehat{nb}$, and $|\cdot|_{\text{CN}}$ in the semantic equations given in figures 4.1–4.6, we get a 0CFA interpretation.

1CFA The 1CFA contour abstraction requires a little more machinery. The abstract contour allocated when entering a lambda is the call site from which the lambda was called. We need to have this information present in the exact contour so that the abstraction function can extract it. A straightforward way to do this is to use *call strings* for the exact contours. A call string is just the string of call sites through which control has passed during the execution of the program.[3] Each time a call is performed, the call site is appended to the end of the call string. Note that in CPS, calls never return, so the current call string is just a trace of the execution path through the program, continuously growing as the program executes. The current call string will suffice as a unique "gensym" for the purposes of the exact semantics, since each time a new call string is constructed, it is longer than all the previously created call strings and hence unique.

If the current contour $b = c_0 \ldots c_n$ when we enter some lambda is the current call string, then the call site we just branched from is clearly c_n. So the contour we want for the 1CFA abstraction is c_n. Thus our contour abstraction function just picks off the last call site in the call string: $|c_0 \ldots c_n| = c_n$.

---

well. This would be accomplished by specifying a new abstraction function and abstract value domain $\widehat{\text{D}}$.

[3] Call strings have been used in other interprocedural analysis work [Harrison 89, Sharir$^+$ 81]. This work usually uses a more complex structure than the simple CPS variant we need here.

So, in the 1CFA model the set of exact contours is just the set of finite strings of call sites: $\text{CN} = \text{LAB}^*$. The initial call string is the label of the top-level call: $b_0 = c_{\text{top}}$. Exact contours are allocated by tacking the current call site onto the end of the current call string: $nb\ b\ c = b\S c$. The abstract contour set is simply call sites: $\widehat{\text{CN}} = \text{LAB}$. When an abstract contour is allocated, it is just taken to be the current call site: $\widehat{nb}\ \hat{b}\ c = c$. Using these definitions in the equations of figures 4.1–4.6 will give us a 1CFA interpretation.

It's easy to change the details of this abstraction to come up with variants on a theme. For example, if we take $\widehat{\text{CN}} = \text{LAB} \times \text{LAB}$, and define $\widehat{nb}\ b\ c = \langle b{\downarrow}2,\ c\rangle$, then our abstract contours are the last *two* calls made during program execution. This would extend 1CFA along the control dimension to a 2CFA abstraction. Alternatively, when entering a closure we could allocate as contour the pair of call sites (1) the call from which we just branched to the closure and (2) the call from which we previously branched to the the closure's lexically superior lambda. This would extend 1CFA along the environment dimension to a different kind of of 2CFA abstraction. There are many ways to abstract the analysis — 0CFA and 1CFA are not intended to be the last word on this subject.

	CN	b_0	$nb\ b\ c$	$\widehat{\text{CN}}$	$\widehat{nb}\ \hat{b}\ c$	$\lvert b\rvert$
0CFA	$\mathcal{Z}$	0	$b+1$	$\{1\}$	1	1
1CFA	LAB^*	c_{top}	$b\S c$	LAB	c	c_n, where $b = c_0 \ldots c_n$

Table 4.11: 0CFA and 1CFA Contour Machinery

The details of these two contour abstractions are summarised in table 4.11. Note that with either of these choices, our domains, allocation policies, and abstractions obey the following properties:

- ***nb* functions as a gensym.**
  As discussed earlier (section 4.1.1), in order for the exact semantics to be correct, the exact contour set needs to be a partially ordered, and the allocation function nb must be an increasing function. This is also necessary to guarantee that the exact call cache produced by the exact control-flow semantics is a partial function (lemma 5).

  In 0CFA, the partial order is just arithmetic $\leq$; in 1CFA, $b \leq b'$ if call string b is a prefix of call string b'. The nb function is clearly increasing for each of these contour definitions.

- **Contour allocation is homomorphic under abstraction: $\lvert nb\ b\ x\rvert = \widehat{nb}\ \lvert b\rvert\ \lvert x\rvert$.**
  This ensures that the evolution of the abstract interpretation stays linked to the exact interpretation – as successive contours are allocated in the two interpre-

tations, they stay related due to this property. The proofs that the abstract semantics safely approximates the exact semantics (section 4.4) rely upon this.

In the particular contour allocation machinery we are using, x is a call label c. If we define $|c| = c$, then both the 0CFA and 1CFA nb functions given above have this property:

$$\text{0CFA:} \quad |nb\ b\ c| = |b+1| = 1 = \widehat{nb}\ |b|\ |c|$$
$$\text{1CFA:} \quad |nb\ b\ c| = |b\S c| = c = \widehat{nb}\ |b|\ c = \widehat{nb}\ |b|\ |c|$$

- **$\widehat{\text{CN}}$ is finite.**
  The computability of the abstract semantics relies upon the finiteness of the contour set (section 4.4.3).

These are the only properties we require of our contour machinery, so the proofs in following sections go through in both the 0CFA and 1CFA cases.

Variable Environments An abstract variable environment maps a variable/abstract-contour pair $\left\langle v,\ \hat{b}\right\rangle$ to an abstract value. When an abstracted variable environment $|ve|$ is applied to such a pair, it performs the lookup in the exact environment ve for each variable/exact-contour pair that could abstract to $\left\langle v,\ \hat{b}\right\rangle$. This produces all the exact values that the corresponding exact lookup might have produced: $\left\{\ ve\ \langle v,\ b\rangle\ \middle|\ |b| = \hat{b}\ \right\}$. Each value in this set is then converted to an abstract value, and all these abstract values are joined together[4] to form the result value $\bigsqcup_{|b|=\hat{b}} |ve\ \langle v,\ b\rangle|$.

This abstraction is a specific example of a general technique for abstracting functions whose domains and ranges have abstractions: $|f| = \lambda\hat{x}.\ \bigsqcup_{|x|=\hat{x}} |f\ x|$. Note that when f is a partial function (as in the variable environment case), it is assumed that the join operation's index variable x ranges over $Dom(f)$.

Stores In the single-address abstraction introduced in section 3.8.1, an abstract store is just the set of all values placed into the store. So an exact store abstracts to its contents: $|\sigma| = \bigsqcup\left\{\ |d|\ \middle|\ d \in Im(\sigma)\ \right\} = \bigsqcup |Im(\sigma)|$.

Call Contexts A call context $cc = \langle c,\ \beta\rangle$ is abstracted by abstracting its component contour environment: $|cc| = \langle c,\ |\beta|\rangle$.

Call Caches Call caches are abstracted following the same pattern for variable environments. When the abstracted call cache $|\gamma|$ is applied to a call context $\left\langle c,\ \widehat{\beta}\right\rangle$, we apply the exact call cache γ to all the exact call contexts that could abstract to $\left\langle c,\ \widehat{\beta}\right\rangle$. All the result procedures are abstracted and joined together[5] to produce the result value: $|\gamma|\ \left\langle c,\ \widehat{\beta}\right\rangle = \bigsqcup_{|cc|=\left\langle c,\widehat{\beta}\right\rangle} |\gamma\ cc| = \bigsqcup_{|\beta|=\widehat{\beta}} |\gamma\ \langle c,\ \beta\rangle|$.

[4] Since abstract values are sets of abstract procedures, and the join operation for $\widehat{\text{D}}$ is set union, this just means all the abstract procedures are collected together into one set.

[5] That is, collected into a set.

4.3 Existence

In the exact semantics, the only values whose existence is non-obvious are the $\mathcal{C}$ and $\mathcal{F}$ functions. The value sets, such as D, Proc, and VEnv, and the semantic functions $\mathcal{PR}$ and $\mathcal{A}$ are all defined with simple, constructive definitions, and clearly exist. $\mathcal{C}$ and $\mathcal{F}$, on the other hand, are defined with recursive equations, and so we must go to a little trouble to prove these equations are well-defined and admit a solution.

We will define $\mathcal{C}$ and $\mathcal{F}$ as the least fixed point of a functional H. That is, we'll define a series of approximations $\mathcal{C}_0, \mathcal{C}_1, \ldots$ and $\mathcal{F}_0, \mathcal{F}_1, \ldots$ to $\mathcal{C}$ and $\mathcal{F}$, such that $\langle \mathcal{C}_{i+1},\, \mathcal{F}_{i+1} \rangle = H\ \langle \mathcal{C}_i,\, \mathcal{F}_i \rangle$. If we can show that H is a continuous operator, then it has a least fixed point, which will be the limit of our approximation series:

$$\langle \mathcal{C},\, \mathcal{F} \rangle = H\ \langle \mathcal{C},\, \mathcal{F} \rangle = \bigsqcup_{i \geq 0} \langle \mathcal{C}_i,\, \mathcal{F}_i \rangle.$$

This is the standard method of constructing functions from recursive definitions in denotational semantics.

We take the definition of H from the recursive definitions of $\mathcal{C}$ and $\mathcal{F}$ given in figures 4.2 and 4.3. $\langle \mathcal{C}'',\, \mathcal{F}'' \rangle = H\ \langle \mathcal{C}',\, \mathcal{F}' \rangle$ is just the value we get if we define $\langle \mathcal{C}'',\, \mathcal{F}'' \rangle$ from the figure 4.2 and 4.3 definitions, using the functions $\mathcal{C}'$ and $\mathcal{F}'$ for the recursive calls inside the equations. Clearly, we are seeking a solution to the equation $\langle \mathcal{C},\, \mathcal{F} \rangle = H\ \langle \mathcal{C},\, \mathcal{F} \rangle$.

4.3.1 H is well-defined

First, we need to determine that H is a well-defined functional. This is guaranteed for us by the constraints we placed on values from CState and FState. Suppose we apply H to a pair $\langle \mathcal{C}',\, \mathcal{F}' \rangle$, getting result $\langle \mathcal{C}'',\, \mathcal{F}'' \rangle$.

Consider $\mathcal{C}''$ first. Suppose $\mathcal{C}''$ is being applied to an argument tuple $\langle c,\, \beta,\, ve,\, b,\, \sigma \rangle$. If the call expression c is a simple call $[\![\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!]$, then $\mathcal{C}''$ uses the partial function $\mathcal{A}$ to evaluate the procedure f and arguments a_i. The C-ENVa constraint (Figure 4.7) on c, β, and ve guarantee that the arguments passed to $\mathcal{A}$ are all within the partial function's domain. So all evaluations with $\mathcal{A}$ are well-defined within $\mathcal{C}''$. We also need to show that in the recursive call to $\mathcal{F}'$, the arguments that $\mathcal{C}''$ passes (f', av, ve, b, and σ) satisfy the constraints of FState so that the application is correct. $\mathcal{C}''$ does not alter the variable environment or store, so to show the environment constraint F-ENV, we need only show that the procedure f' and arguments av are all consistent with ve. If the procedure expression f or an argument a_i is a constant or primop, $\mathcal{A}$ will certainly evaluate it to a non-closure, so we can ignore it. If it is a variable, $\mathcal{A}$ will evaluate it to some already-existing value in $Im(ve)$, so the constraints on $\mathcal{C}''$ guarantee us the value is consistent with ve. On the other hand, if the expression is a lambda ℓ, then $\mathcal{A}$ will evaluate it to a new closure $\langle \ell,\, \beta \rangle$; we need to show that this new closure is consistent with the variable environment. This is straightforward. Suppose lambda ℓ is an argument of call c, and is being closed in environment β/ve. The CState constraint requires $Dom(\beta)$ to include c's lexical context, that is, every lambda lexically superior to c. This is ℓ's lexical context as well, so the F-ENVa half of the FState environment constraint is satisfied. The CState constraint C-ENVa also requires ve to cover β, so the F-ENVb half of FState's environment constraint is satisfied as well. We've syntactically restricted primop applications to be of correct arity, so the F-ARITY constraint is preserved if c happens to be a primop application. Clearly, since $\mathcal{C}''$ does not allocate any new addresses or alter the store, the address constraint F-ADDR is preserved in the arguments it passes to $\mathcal{F}'$. So $\mathcal{C}''$ is well-defined in the simple-call case.

Suppose that the call expression c that $\mathcal{C}''$ is evaluating is instead a `letrec` expression: $[\![$`(letrec ((`f_1 l_1`)...)` c'`)`$]\!]$. In this case, $\mathcal{C}''$ recursively calls $\mathcal{C}'$ to evaluate the internal call expression in an augmented environment β'/ve'. To show that this application of $\mathcal{C}'$ is well-defined, we need to verify that it is applied to arguments satisfying the CState constraints. Again, clearly the store constraint C-ADDR is preserved, since the store is unchanged, and no new addresses are introduced into the system. Verifying the environment constraint C-ENV means showing that the inner call expression c' is consistent with the environment structure that $\mathcal{C}''$ builds, and showing that each closure $\langle l_i,\ \beta'\rangle$ created by the `letrec` is consistent with the updated variable environment ve'.

To show that a `letrec` closure $\langle l_i,\ \beta'\rangle$ is consistent with ve', we need to show first that the domain of β' contains l_i's lexical context. l_i's lexical context is the lexical context of c plus the binding introduced by the `letrec` itself. This is just $Dom(\beta) \cup \{c\}$, which is precisely the domain of β'. Similarly, ve' covers β' by construction. The new variable environment ve' is the old ve plus an update $[\langle f_i,\ b'\rangle \mapsto \mathcal{A}\ l_i\ \beta'\ ve]$, where $b' = nb\, b$. If a binding construct ℓ' is in the domain of $\beta' = \beta[c \mapsto b']$, then either it is c or it is in $Dom(\beta)$. If it is the `letrec` c, then the update portion of ve' (*i.e.*, $[\langle f_i,\ b'\rangle \mapsto \mathcal{A}\ l_i\ \beta'\ ve]$) handles all of the `letrec`'s variables. If it is in $Dom(\beta)$, then the old portion of ve' (*i.e.*, ve) handles all of the variables bound by ℓ'.

Showing that $\mathcal{F}''$ is well-defined follows similar reasoning. When $\mathcal{F}''$ is applied to a closure $\langle \ell,\ \beta\rangle$, we need to show that the internal application of $\mathcal{C}'$ has arguments satisfying the CState constraints. This is straightforward. We know that the arguments of $\mathcal{F}''$ satisfy the FState constraints. This means that $\langle \ell,\ \beta\rangle$ is consistent with the variable environment ve (by F-ENV). After augmenting the environment structure to bind the lambda ℓ's variables, the resulting environment must be consistent with the lambda's internal call expression. Since the store remains unaltered, its constraints remain satisfied. No new addresses or closures are added to the system, so the other constraints remain satisfied.

The primop cases in the definition of $\mathcal{F}''$ generally involve recursive applications of $\mathcal{F}'$ to invoke the primop continuation. So we need to show that the arguments used in the recursive applications satisfy the FState constraints. Consider the addition operator. No new environment contexts or addresses are introduced into the system, and the continuation, being a first-class value, can't be a primop, so the constraints are all trivially satisfied. The `if`, `contents`, and `set` primops are similar. The `new` primop, however, does introduce a new address into the system. Since this address is defined by the result store passed along to the recursive $\mathcal{F}'$ application, however, we continue to satisfy the FState constraints.

4.3.2 H is continuous

Now that we've established H is a well-defined operator, we need to show that it is a continuous one, which will allow us to take its fixed point. Continuous operators operate on values taken from a cpo. H operates on values taken from the space

$$(\text{CState} \rightarrow \text{CCache}) \times (\text{FState} \rightarrow \text{CCache}).$$

This space is a cpo, under the pointwise ordering induced by the ordering on call caches. Call caches are ordered by set inclusion: $\gamma \sqsubseteq \gamma'$ iff $\gamma \subset \gamma'$. So the cpo upon which H acts has bottom value $\bot = \langle \lambda cs.\ \emptyset,\ \lambda fs.\ \emptyset \rangle$.

H is a continuous operator if

- it preserves chains:
  $\langle\, \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle \mid i \geq 0 \,\rangle$ a chain $\Rightarrow \langle\, H\ \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle \mid i \geq 0 \,\rangle$ a chain
- it preserves limits on chains:
  $H \bigsqcup\limits_{i \geq 0} \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle = \bigsqcup\limits_{i \geq 0} H\ \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle.$

Lemma 1 *H is monotonic*

This follows straightforwardly from the form of H.

Lemma 2 *H preserves limits on chains. That is, if $\langle\, \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle \mid i \geq 0 \,\rangle$ is a chain, then $H \bigsqcup_{i \geq 0} \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle = \bigsqcup_{i \geq 0} H\ \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle$.*

Proof: Let $\langle \mathcal{C}_\infty,\ \mathcal{F}_\infty \rangle$ be the limit of the chain: $\langle \mathcal{C}_\infty,\ \mathcal{F}_\infty \rangle = \bigsqcup_{i \geq 0} \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle$. Let $\langle \mathcal{C}',\ \mathcal{F}' \rangle = H\ \langle \mathcal{C}_\infty,\ \mathcal{F}_\infty \rangle$ and $\langle \mathcal{C}'',\ \mathcal{F}'' \rangle = \bigsqcup_{i \geq 0} H\ \langle \mathcal{C}_i,\ \mathcal{F}_i \rangle$. We want to show that $\langle \mathcal{C}',\ \mathcal{F}' \rangle = \langle \mathcal{C}'',\ \mathcal{F}'' \rangle$.

Part 1: $\mathcal{C}' = \mathcal{C}''$

Let cs be some argument drawn from CState — that is, cs is a tuple $\langle c,\ \beta,\ ve,\ b,\ \sigma \rangle$. The call expression c is either a simple call or a `letrec`. Suppose it is a simple call, and that evaluation of the call doesn't cause an immediate run-time error. Then

$$\mathcal{C}'\ cs = \{\langle cc,\ f \rangle\} \cup \mathcal{F}_\infty\, fs,$$

where $cc = \langle c,\ \beta \rangle$ and $\mathcal{F}_\infty\, fs$ is the recursive call to $\mathcal{F}_\infty$, with fs the appropriate argument tuple. On the other hand,

$$\begin{aligned} \mathcal{C}''\ cs &= \bigcup_i (\{\langle cc,\ f \rangle\} \cup \mathcal{F}_i\, fs) \\ &= \{\langle cc,\ f \rangle\} \cup \bigcup_i \mathcal{F}_i\, fs \\ &= \{\langle cc,\ f \rangle\} \cup \mathcal{F}_\infty\, fs. \end{aligned}$$

So $\mathcal{C}'\ cs = \mathcal{C}''\ cs$.

If evaluation of the call's subforms causes a run-time error, then the situation is even simpler, since all the functions abort and produce the empty cache:

$$\mathcal{C}''\ cs = \bigcup_i \emptyset = \emptyset = \mathcal{C}'\ cs.$$

If c is not a simple call, then it is a `letrec` expression: $[\![$`(letrec ((`$f_1\ \ l_1$`)...)` c'`)`$]\!]$. In this case,

$$\mathcal{C}'\ cs = \mathcal{C}_\infty\ cs',$$

where cs' is the argument value passed on the recursive call to evaluate the `letrec`'s inner call expression c'. However,

$$\mathcal{C}''\ cs = \bigcup_i \mathcal{C}_i\ cs' = \mathcal{C}_\infty\ cs'.$$

So $\mathcal{C}'\ cs = \mathcal{C}''\ cs$.

In either case, $\mathcal{C}'\ cs = \mathcal{C}''\ cs$, and so $\mathcal{C}' = \mathcal{C}''$.

Part 2: $F' = F''$

The main recursive case proceeds along the same lines as its $\mathcal{C}' = \mathcal{C}''$ counterpart. Let fs be some argument drawn from FState — that is, fs is a tuple $\langle f,\ av,\ ve,\ b,\ \sigma\rangle$. Suppose that the procedure f is a closure, and that processing the procedure application doesn't cause a run-time error.

$$\mathcal{F}'\ fs = \mathcal{C}_\infty\ cs,$$

where cs the appropriate argument tuple for the recursive call to $\mathcal{C}_\infty$. On the other hand,

$$\mathcal{F}''\ fs = \bigcup_i \mathcal{C}_i\ cs = \mathcal{C}_\infty\ cs.$$

So $\mathcal{F}'\ fs = \mathcal{F}''\ fs$.

Suppose instead that the procedure f is a primop, and, again, that no run-time errors are caused. We'll only consider the $f = [\![p\text{:+}]\!]$ case here; the others primops are similar. In this case,

$$\mathcal{F}'\ fs = \left\{\left\langle\left\langle ic_{p:+},\ b\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}_\infty\ fs',$$

where $\left\{\left\langle\left\langle ic_{p:+},\ b\right\rangle,\ k\right\rangle\right\}$ records the immediate call to the continuation, and $\mathcal{F}_\infty\ fs'$ is the recursive application that actually handles the continuation. Considering $\mathcal{F}''$, we have

$$\mathcal{F}''\ fs = \bigcup_i \left(\left\{\left\langle\left\langle ic_{p:+},\ b\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}_i\ fs'\right) = \left\{\left\langle\left\langle ic_{p:+},\ b\right\rangle,\ k\right\rangle\right\} \cup \mathcal{F}_\infty\ fs'.$$

So $\mathcal{F}'\ fs = \mathcal{F}''\ fs$.

If $f = stop$ or fs causes a run-time error, then, just as in the $\mathcal{C}$ error case, both $\mathcal{F}'$ and $\mathcal{F}''$ trivially return the empty call cache $\emptyset$.

In either event, $\mathcal{F}'\ fs = \mathcal{F}''\ fs$. So $\mathcal{F}' = \mathcal{F}''$.

Q.E.D.

Theorem 3 *H is continuous*

Proof: This follows from the two previous lemmas.

Corollary 4 *H has a least fixed point: $\langle \mathcal{C}, \mathcal{F} \rangle = \text{fix}\, H$.*

Proof: Continuous functions have least fixed points.

This suffices to show the main result of this section: the $\mathcal{C}$ and $\mathcal{F}$ functions exist. We also need to know that the abstract functions $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ exist. This involves showing that their defining functional $\widehat{H}$ is well-defined and continuous. I omit the actual proof of this fact, as it is almost identical to the proof for $\mathcal{C}$ and $\mathcal{F}$.

4.4 Properties of the Semantic Functions

Now that we know $\mathcal{C}$, $\mathcal{F}$, $\mathcal{A}$, $\mathcal{PR}$ and their abstract counterparts exist, we can proceed to proving properties about them. In this section we'll work out proofs to show

- The answer returned by $\mathcal{PR}$ is actually a partial function.
- $\widehat{\mathcal{PR}}$ safely approximates $\mathcal{PR}$.
- $\widehat{\mathcal{PR}}$ is computable.

The proof technique we'll use to show most of these theorems and their supporting lemmas is the technique of fixed-point induction.

Suppose that Φ is a predicate on some cpo D. Φ is an *admissible predicate* if Φ holds over a chain $\langle\, d_i \mid i \geq 0 \,\rangle$ implies that Φ holds on the limit: $\Phi \,\langle\, d_i \mid i \geq 0 \,\rangle \Rightarrow \Phi \bigsqcup_i d_i$. It is a simple theorem that if Φ is an admissible predicate on D, $H\colon D \to D$ is a continuous function, $\Phi \bot$, and $\Phi\, d \Rightarrow \Phi\, (H\, d)$, then $\Phi\, (\text{fix}\, f)$. This is a standard proof technique found in most introductory texts on denotational semantics [Brookes 89, Schmidt 86, Stoy 77].

4.4.1 Call caches are partial functions

Lemma 5 *The call cache returned by $\mathcal{C}$ or $\mathcal{F}$ is always a partial function.*

Informal proof: Each time $\mathcal{F}$ calls $\mathcal{C}$, it passes along a new contour environment β that has never been created before. The contour environment is guaranteed to be unique because the innermost contour was newly allocated by $\mathcal{F}$. So $\mathcal{C}$'s update $\langle\langle c, \beta\rangle, f'\rangle$ is guaranteed unique in β. Similar reasoning applies to the cache entries made by the primops: their cache entries always have a contour environment containing a newly allocated contour, so the environment is guaranteed unique. No collisions among the entries means the result cache is a partial function.

A careful proof would use an admissible predicate formalising the above argument. The predicate would be something like: if $\mathcal{C}$'s argument has (1) b dominating all contours in the system, and (2) β does not occur in the system then (3) the returned call cache is a partial function and (4) b lower bounds the innermost contour of every entry in the returned call cache … with similar statements about $\mathcal{F}$.

Establishing that our semantic functions produce partial functions allows us to do two useful things. First, we can now manipulate the result call cache γ in functional ways *e.g.*, we can apply it to values: $\gamma\ cc$. Second, we can take the call cache updates in the defining equations to be either a set update ($\gamma \cup \{\langle\langle c,\ \beta\rangle,\ f\rangle\}$) or a function update ($\gamma\,[\langle c,\ \beta\rangle \mapsto f]$) as convenient, since the result is the same.

4.4.2 Abstract semantics is a safe approximation

In this section, we will show that our abstract control-flow analysis gives a safe approximation to the exact analysis.

First, we need to define formally what we mean by "conservatively approximates." An exact call-cache γ can have an infinite domain and image — it is a table that gives infinitely precise answers to infinitely precise queries. At compile time, however, we are restricted to asking finitely precise queries, and receiving finitely precise answers. Thus, an abstract call-cache $\hat{\gamma}$ has finite domain and range.

Now, an abstract call context $\widehat{cc} \in \widehat{\text{CCtxt}}$ corresponds to a set of exact call contexts, that is, the set of all exact call contexts that abstract to $\widehat{cc}$:

$$\left\{\, cc \;\middle|\; |cc| = \widehat{cc} \,\right\}.$$

That is, the finitely precise abstract call context $\widehat{cc}$ describes a set of potential exact call contexts.

Similarly, an abstract procedure $\hat{f} \in \widehat{\text{D}}$ corresponds to a set of exact procedures, that is, the set of all exact procedures that abstract to values contained in $\hat{f}$:

$$\left\{\, f \;\middle|\; |f| \sqsubseteq \hat{f} \,\right\}.$$

(In this case $|f| = \{|f|_{\text{Proc}}\}$, so $|f| \sqsubseteq \hat{f}$ means $|f|_{\text{Proc}} \in \hat{f}$.)

So, when we query an abstract call cache $\hat{\gamma}$ with some abstract call context $\widehat{cc}$, we get back an abstract procedure $\hat{\gamma}\ \widehat{cc}$ that describes a set of exact procedures:

$$\left\{\, f \;\middle|\; |f| \sqsubseteq \hat{\gamma}\ \widehat{cc} \,\right\}.$$

If we query an exact call cache γ with the equivalent set of exact call contexts, we get

$$\left\{\, \gamma\ cc \;\middle|\; |cc| = \widehat{cc} \,\right\}.$$

Our abstract call cache is a safe approximation if the approximate answer to our approximate query $\widehat{cc}$ contains the exact answer to our approximate query:

$$\left\{ \gamma\ cc \;\middle|\; |cc| = \widehat{cc} \right\} \subset \left\{ f \;\middle|\; |f| \sqsubseteq \hat{\gamma}\ \widehat{cc} \right\}.$$

The left-hand side is the set of all procedures that were really called from some call context described by the abstraction $\widehat{cc}$. The inclusion guarantees us that all of these procedures are included in the answer the abstract call cache produces. This inclusion is equivalent to stating that

$$|cc| = \widehat{cc} \quad \Rightarrow \quad |\gamma\ cc| \sqsubseteq \hat{\gamma}\ \widehat{cc}.$$

So, stating that "$\hat{\gamma}$ conservatively approximates γ" can be formally stated as "$|\gamma\ cc| \sqsubseteq \hat{\gamma}\ |cc|$." We can extend this concept to any exact/abstract pair of functions:

Definition 3

$\hat{f}$ conservatively approximates *(c.a.)* f *if* $\hat{f}\ |x| \sqsupseteq |f\ x|$ (i.e., $\hat{f} \circ |\cdot| \sqsupseteq |\cdot| \circ f$).

$\hat{f}$ strongly conservatively approximates *(s.c.a.)* f *if* $\hat{x} \sqsupseteq |x| \Rightarrow \hat{f}\,\hat{x} \sqsupseteq |f\ x|$.

If f is a partial function, it is understood that x is restricted to Dom(f) *in the above comparisons.*

"Strongly conservatively approximates," is an even stronger condition than simple "conservatively approximates," and will be of use in our upcoming proofs of correctness.

Note the extra qualification in this definition for the partial function case. We glossed over this detail in the above discussion: γ, a partial function, is not defined over all the cc to which we might apply it in the expression $\left\{ \gamma\ cc \;\middle|\; |cc| = \widehat{cc} \right\}$. When $cc \notin Dom(\gamma)$, then there is no call recorded in the cache for that call context, so the correct expression is $\left\{ \gamma\ cc \;\middle|\; |cc| = \widehat{cc} \wedge cc \in Dom(cc) \right\}$. With this qualification, our definition correctly expresses the notion of safe approximation in the partial function case as well as the total function one.

In this section we will prove as lemmas that (roughly speaking) $\widehat{\mathcal{A}}$, $\widehat{\mathcal{C}}$, and $\widehat{\mathcal{F}}$ conservatively approximate $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{F}$, respectively. With these lemmas, we can show the main result of this section:

Theorem 6 *$\widehat{\mathcal{PR}}\ \ell$ conservatively approximates $\mathcal{PR}\ \ell$.*
That is, $\hat{\gamma} \circ |\cdot| \sqsupseteq |\cdot| \circ \gamma$, where $\hat{\gamma} = \widehat{\mathcal{PR}}\ \ell$, $\gamma = \mathcal{PR}\ \ell$.

Proof: The theorem follows from the fact that $\widehat{\mathcal{F}}$ conservatively approximates $\mathcal{F}$.
Let $\gamma = \mathcal{PR}\ \ell$ and $\hat{\gamma} = \widehat{\mathcal{PR}}\ \ell$.

$$\hat{\gamma} = \widehat{\mathcal{PR}}\ \ell = \widehat{\mathcal{F}}\ \langle\langle \ell,\, [\,]\rangle,\, \langle\{stop\}\rangle,\, [\,],\, |b_0|,\, [\,]\rangle \sqcup [\langle c_{\text{top}},\, [\,]\rangle \mapsto \{\langle \ell,\, [\,]\rangle\}]$$
$$\gamma = \widehat{\mathcal{PR}}\ \ell = \mathcal{F}\ \langle\langle \ell,\, [\,]\rangle,\, \langle stop\rangle,\, [\,],\, b_0,\, [\,]\rangle\ [\langle c_{\text{top}},\, [\,]\rangle \mapsto \langle \ell,\, [\,]\rangle]$$

There are two cases to consider.

- $cc = \langle c_{\text{top}},\, [\,]\rangle$:

$$\hat{\gamma}\ \ |cc| = \hat{\gamma}\ \ |\langle c_{\text{top}},\, [\,]\rangle| = \hat{\gamma}\ \ \langle c_{\text{top}},\, [\,]\rangle \sqsupseteq \{\langle \ell,\, [\,]\rangle\}\,,$$
$$|\gamma\ cc| = |\gamma\ \ \langle c_{\text{top}},\, [\,]\rangle| = |\langle \ell,\, [\,]\rangle| = \{\langle \ell,\, [\,]\rangle\}\,.$$

  So, $\hat{\gamma}\ \ |cc| \sqsupseteq |\gamma\ cc|$.

- $cc \neq \langle c_{\text{top}},\, [\,]\rangle$:

$$\hat{\gamma}\ \ |cc| = \widehat{\mathcal{F}}\ \ \langle\langle \ell,\, [\,]\rangle,\, \langle\{\mathit{stop}\}\rangle,\, [\,],\, |b_0|,\, [\,]\rangle\ \ |cc|$$
$$|\gamma\ cc| = |\mathcal{F}\ \ \langle\langle \ell,\, [\,]\rangle,\, \langle \mathit{stop}\rangle,\, [\,],\, b_0,\, [\,]\rangle\ \ cc|$$

  Now, $|\langle \ell,\, [\,]\rangle| = \langle \ell,\, [\,]\rangle$, $|\langle \mathit{stop}\rangle| = \langle\{\mathit{stop}\}\rangle$, and $|[\,]| = [\,]$, so

$$\hat{\gamma}\ \ |cc| \sqsupseteq |\gamma\ cc|$$

  by the conservative approximation property of $\mathcal{F}$ and $\widehat{\mathcal{F}}$ (lemma 9).

In either case, $\hat{\gamma}\ \ |cc| \sqsupseteq |\gamma\ cc|$, and the theorem is shown. Q.E.D.

Now, let's prove the supporting lemmas of the above theorem. Because $\widehat{\mathcal{A}}$ and $\mathcal{A}$ are defined non-recursively, the proof of their approximation lemma is correspondingly direct.

Lemma 7 *$\widehat{\mathcal{A}}\, a\, \widehat{ve}\ \ |\beta| \sqsupseteq |\mathcal{A}\ a\ ve\ \beta|$ where $\widehat{ve} \sqsupseteq |ve|$.* *(s.c.a. in ve and c.a. in β.)*

Proof by cases:

1. $a = [\![k]\!]$ or $a = [\![\mathit{prim}]\!]$ (constant or primop)
   Immediate.

2. $a = [\![\ell{:}(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]$ (lambda)

$$\widehat{\mathcal{A}}\,\ell\ ve\ \ |\beta| = \{\langle \ell,\, |\beta|\rangle\}$$
$$|\mathcal{A}\ \ell\ ve\ \beta| = |\langle \ell,\, \beta\rangle| = \{\langle \ell,\, |\beta|\rangle\}$$

3. $a = [\![r{:}v]\!]$ (variable reference)
   Let $\ell = \mathit{binder}\ v$.

$$\begin{aligned}\widehat{\mathcal{A}}\, r\ \widehat{ve}\ \ |\beta| &= \widehat{ve}\ \ \langle v,\, |\beta|\ \ \ell\rangle \sqsupseteq |ve|\ \ \langle v,\, |\beta|\ \ \ell\rangle \\ &= \bigsqcup_{|b|=|\beta|\ \ell} |ve\ \ \langle v,\, b\rangle| \sqsupseteq |ve\ \ \langle v,\, \beta\ \ell\rangle| \\ &= |\mathcal{A}\ r\ ve\ \beta|\end{aligned}$$

Q.E.D.

The next two lemmas use the elementwise ordering on $\widehat{\text{CState}}$ and $\widehat{\text{FState}}$. Note that the c, $\widehat{\beta}$, and $\hat{b}$ components of a $\widehat{\text{CState}}$ tuple, and the $\hat{f}$, $\hat{b}$ components of a $\widehat{\text{FState}}$ tuple are ordered discretely. This means, for example, that if $\left\langle c_1,\, \widehat{\beta}_1,\, \widehat{ve}_1,\, \hat{b}_1,\, \widehat{\sigma}_1\right\rangle \sqsubseteq \left\langle c_2,\, \widehat{\beta}_2,\, \widehat{ve}_2,\, \hat{b}_2,\, \widehat{\sigma}_2\right\rangle$, it must be the case that $c_1 = c_2$, $\widehat{\beta}_1 = \widehat{\beta}_2$, and $\hat{b}_1 = \hat{b}_2$.

Lemma 8 $\widehat{\mathcal{C}}\ \widehat{cs}\ |cc| \sqsupseteq |\mathcal{C}\ cs\ cc|$, *where* $\widehat{cs} \sqsupseteq |cs|$. *(s.c.a. in cs, and c.a. in cc)*
Equivalently: Suppose $cs = \langle c,\ \beta,\ ve,\ b,\ \sigma\rangle \in \text{CState}$ *and* $\widehat{cs} = \langle c,\ |\beta|,\ \widehat{ve},\ |b|,\ \hat{\sigma}\rangle \in \widehat{\text{CState}}$ *such that* $\widehat{cs} \sqsupseteq |cs|$. *Let* $\gamma = \mathcal{C}\ \langle c,\ \beta,\ ve,\ b,\ \sigma\rangle$ *and* $\hat{\gamma} = \widehat{\mathcal{C}}\ \langle c,\ |\beta|,\ \widehat{ve},\ |b|,\ \hat{\sigma}\rangle$. *Then* $cc \in Dom(\gamma) \Rightarrow \hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

Lemma 9 $\widehat{\mathcal{F}}\ \widehat{fs}\ |cc| \sqsupseteq |\mathcal{F}\ fs\ cc|$, *where* $\widehat{fs} \sqsupseteq |fs|$. *(s.c.a. in fs, and c.a. in cc)*
Equivalently: Suppose $fs = \langle f,\ av,\ ve,\ b,\ \sigma\rangle \in \text{FState}$ *and* $\widehat{fs} = \langle |f|,\ \widehat{av},\ \widehat{ve},\ |b|,\ \hat{\sigma}\rangle \in \widehat{\text{FState}}$ *such that* $\widehat{fs} \sqsupseteq |fs|$. *Let* $\gamma = \mathcal{F}\ \langle f,\ av,\ ve,\ b,\ \sigma\rangle$ *and* $\hat{\gamma} = \widehat{\mathcal{F}}\ \langle |f|,\ \widehat{av},\ \widehat{ve},\ |b|,\ \hat{\sigma}\rangle$. *Then* $cc \in Dom(\gamma) \Rightarrow \hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

The "equivalent" restatements of these lemmas take advantage of the observation that $\hat{\gamma} \sqsupseteq |\gamma|$ iff $\hat{\gamma}$ c.a. γ. This is due to the definition of $|\cdot|_{\text{CCache}}$ (figure 4.10).

Proofs of the Inductive Case

Because of their recursive definitions, the approximation lemmas for $\mathcal{C}$, $\widehat{\mathcal{C}}$, $\mathcal{F}$ and $\widehat{\mathcal{F}}$ are proved with a fixed-point induction. The following proofs will proceed in two distinct steps. First, we'll "prove" the lemmas in a recursive fashion: the proofs of the lemmas will actually assume the lemmas already hold for the applications of the semantic functions in the recursive expansions of the function definitions. This does not constitute a sound proof. These "recursive proofs" do, however, establish the inductive case of the related admissible predicate. This is the part of the proof that provides the most insight, and so we'll handle it separately. The extra issues of showing the base case and the limit case are uninteresting machinery, and will be addressed in the following text.

As we discussed in section 4.2, our contour abstraction and allocation functions are related by the identity $|nb\ b\ c| = \widehat{nb}\ |b|\ c$. We'll be using these two expressions interchangeably in the proofs of the following two lemmas.

Lemma 8 Proof of Inductive Case ($\widehat{\mathcal{C}}$ conservatively approximates $\mathcal{C}$)
Recursion assumption: $\mathcal{C}$ and $\widehat{\mathcal{C}}$ are defined in terms of $\mathcal{C}$ and $\widehat{\mathcal{C}}$ functions for which lemma 8 holds, and $\mathcal{F}$ and $\widehat{\mathcal{F}}$ functions for which lemma 9 holds.

Suppose the call expression being evaluated is a simple call, $[\![c{:}(f\ \ a_1 \ldots a_n)]\!]$. Let $\hat{f} = \widehat{\mathcal{A}}\, f\ |\beta|\ \widehat{ve}$, $\widehat{av}{\downarrow}i = \widehat{\mathcal{A}}\, a_i\ |\beta|\ \widehat{ve}$, and $f' = \mathcal{A}\ f\ \beta\ ve$. If $f' \notin \text{Proc}$, then $\mathcal{C}$ encounters a run-time error, and aborts with the empty cache [ ]. In this case, the lemma holds trivially. Suppose, instead, that $f' \in \text{Proc}$.

$$\begin{aligned}
&\widehat{\mathcal{C}}\ \langle c,\ |\beta|,\ \widehat{ve},\ |b|,\ \hat{\sigma}\rangle \\
&\quad = \left[\langle c,\ |\beta|\rangle \mapsto \hat{f}\right] \sqcup \bigsqcup_{\hat{f}' \in \hat{f}} \widehat{\mathcal{F}}\ \left\langle \hat{f}',\ \widehat{av},\ \widehat{ve},\ \widehat{nb}\ |b|\ c,\ \hat{\sigma}\right\rangle && \text{(4.1a)}\\
&\quad \sqsupseteq [\langle c,\ |\beta|\rangle \mapsto |f'|] \sqcup \bigsqcup_{\hat{f}' \in |f'|} \widehat{\mathcal{F}}\ \left\langle \hat{f}',\ \widehat{av},\ \widehat{ve},\ \widehat{nb}\ |b|\ c,\ \hat{\sigma}\right\rangle && \text{(4.1b)}\\
&\quad = [\langle c,\ |\beta|\rangle \mapsto |f'|] \sqcup \widehat{\mathcal{F}}\ \left\langle |f'|_{\text{Proc}},\ \widehat{av},\ \widehat{ve},\ \widehat{nb}\ |b|\ c,\ \hat{\sigma}\right\rangle && \text{(4.1c)}
\end{aligned}$$

Equation 4.1a is the definition of $\widehat{\mathcal{C}}$. Equation 4.1b holds because $\widehat{\mathcal{A}}$ approximates $\mathcal{A}$ (lemma 7), so that $|f'| \sqsubseteq \hat{f}$. The join over $\hat{f}' \in |f'|$ can be discarded, because $|f'| = \{|f'|_{\text{Proc}}\}$ is a singleton set, giving us equation 4.1c.

a. $cc = \langle c,\, \beta\rangle$

$$\begin{aligned}
&\widehat{\mathcal{C}}\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \hat{\sigma}\rangle\ |cc| \\
&\quad \sqsupseteq \left([\langle c,\, |\beta|\rangle \mapsto |f'|] \sqcup \widehat{\mathcal{F}}\ \left\langle |f'|_{\text{Proc}},\, \widehat{av},\, \widehat{ve},\, \widehat{nb}\ |b|\ c,\, \hat{\sigma}\right\rangle\right)\ \langle c,\, |\beta|\rangle \qquad \text{(by 4.1)} \\
&\quad \sqsupseteq |f'|
\end{aligned}$$

$$\begin{aligned}
&|\mathcal{C}\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc| \\
&\quad = |(\mathcal{F}\ \langle f',\, av,\, ve,\, nb\, b\, c,\, \sigma\rangle)\,[\langle c,\, \beta\rangle \mapsto f']\ \langle c,\, \beta\rangle| = |f'| \qquad \text{(by def'n } \mathcal{C}\text{)}
\end{aligned}$$

Therefore, $\widehat{\mathcal{C}}\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \hat{\sigma}\rangle\ |cc| \sqsupseteq |\mathcal{C}\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc|$.

b. $cc \neq \langle c,\, \beta\rangle$

$$\begin{aligned}
&\widehat{\mathcal{C}}\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \hat{\sigma}\rangle\ |cc| \\
&\quad \sqsupseteq \left([\langle c,\, |\beta|\rangle \mapsto |f'|] \sqcup \widehat{\mathcal{F}}\ \left\langle |f'|_{\text{Proc}},\, \widehat{av},\, \widehat{ve},\, \widehat{nb}\ |b|\ c,\, \hat{\sigma}\right\rangle\right)\ |cc| \qquad \text{(by 4.1)} \\
&\quad = \widehat{\mathcal{F}}\ \left\langle |f'|_{\text{Proc}},\, \widehat{av},\, \widehat{ve},\, \widehat{nb}\ |b|\ c,\, \hat{\sigma}\right\rangle\ |cc| \\
&\quad = \widehat{\mathcal{F}}\ \langle |f'|_{\text{Proc}},\, \widehat{av},\, \widehat{ve},\, |nb\, b\, c|,\, \hat{\sigma}\rangle\ |cc|
\end{aligned}$$

$$\begin{aligned}
|\mathcal{C}\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc| &= |(\mathcal{F}\ \langle f',\, av,\, ve,\, nb\, b\, c,\, \sigma\rangle)\,[\langle c,\, \beta\rangle \mapsto f']\ cc| \\
&= |\mathcal{F}\ \langle f',\, av,\, ve,\, nb\, b\, c,\, \sigma\rangle\ cc|
\end{aligned}$$

Therefore, $\widehat{\mathcal{C}}\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \hat{\sigma}\rangle\ |cc| \sqsupseteq |\mathcal{C}\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc|$ by c.a. on $\mathcal{F}$.

Suppose, instead, that $\mathcal{C}$ is evaluating a `letrec` form, $[\![c\texttt{:(letrec ((}f_1\ l_1\texttt{)...)}\ c'\texttt{)}]\!]$. Let $b' = nb\ b\ c$ (so that $|b'| = \widehat{nb}\ |b|\ c$), and $\beta' = \beta[c \mapsto b']$ (so that $|\beta'| = |\beta|\ [c \mapsto |b'|]$).

$$\begin{aligned}
\widehat{\mathcal{C}}\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \hat{\sigma}\rangle\ |cc| &= \widehat{\mathcal{C}}\ \left\langle c',\, |\beta'|,\, \widehat{ve} \sqcup \left[\langle f_i,\, |b'|\rangle \mapsto \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve}\right],\, |b'|,\, \hat{\sigma}\right\rangle\ |cc| \\
|\mathcal{C}\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc| &= |\mathcal{C}\ \langle c',\, \beta',\, ve[\langle f_i,\, b'\rangle \mapsto \mathcal{A}\, l_i\, \beta'\ ve],\, b',\, \sigma\rangle\ cc|
\end{aligned}$$

The result follows from (1) $\widehat{ve} \sqcup \left[\langle f_i,\, |b'|\rangle \mapsto \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve}\right] \sqsupseteq |ve[\langle f_i,\, b'\rangle \mapsto \mathcal{A}\, l_i\, \beta'\ ve]|$ (shown below), and (2) the recursion assumption on $\mathcal{C}$ and $\widehat{\mathcal{C}}$.

To see that

$$\widehat{ve} \sqcup \left[\langle f_i,\, |b'|\rangle \mapsto \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve}\right] \sqsupseteq |ve[\langle f_i,\, b'\rangle \mapsto \mathcal{A}\, l_i\, \beta'\ ve]|\,,$$

let $\hat{g}$ be the expression on the left, and $\hat{h}$ be the expression on the right.

$\underline{\hat{g}\ \langle f_i,\ |b'|\rangle \sqsupseteq \hat{h}\ \langle f_i,\ |b'|\rangle}$

$$\begin{aligned}
\hat{g}\ \langle f_i,\ |b'|\rangle &= \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve} \sqcup \widehat{ve}\ \langle f_i,\ |b'|\rangle \sqsupseteq \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve} \sqcup |ve|\ \langle f_i,\ |b'|\rangle \\
&= \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve} \sqcup \bigsqcup_{|b''|=|b'|} |ve\ \langle f_i,\ b''\rangle| \\
&\sqsupseteq \widehat{\mathcal{A}}\, l_i\ |\beta'|\ \widehat{ve} \sqcup \bigsqcup_{\substack{|b''|=|b'| \\ b''\neq b'}} |ve\ \langle f_i,\ b''\rangle| \\
&\sqsupseteq |\mathcal{A}\, l_i\ \beta'\ ve| \sqcup \bigsqcup_{\substack{|b''|=|b'| \\ b''\neq b'}} |ve\ \langle f_i,\ b''\rangle| \\
\hat{h}\ \langle f_i,\ |b'|\rangle &= \bigsqcup_{|b''|=|b'|} |ve[\langle f_i,\ b'\rangle \mapsto \mathcal{A}\, l_i\ \beta'\ ve]\ \langle f_i,\ b''\rangle| \\
&= |\mathcal{A}\, l_i\ \beta'\ ve| \sqcup \bigsqcup_{\substack{|b''|=|b'| \\ b''\neq b'}} |ve\ \langle f_i,\ b''\rangle|
\end{aligned}$$

$\underline{\hat{g}\ \left\langle v,\ \hat{b}\right\rangle \sqsupseteq \hat{h}\ \left\langle v,\ \hat{b}\right\rangle \textbf{ for } \left\langle v,\ \hat{b}\right\rangle \neq \langle f_i,\ |b'|\rangle}$

$$\begin{aligned}
\hat{g}\ \left\langle v,\ \hat{b}\right\rangle &= \widehat{ve}\ \left\langle v,\ \hat{b}\right\rangle \\
\hat{h}\ \left\langle v,\ \hat{b}\right\rangle &= \bigsqcup_{|b''|=\hat{b}} |ve[\langle f_i,\ b'\rangle \mapsto \mathcal{A}\, l_i\ \beta'\ ve]\ \langle v,\ b''\rangle| \\
&= \bigsqcup_{|b''|=\hat{b}} |ve\ \langle v,\ b''\rangle| = |ve|\ \left\langle v,\ \hat{b}\right\rangle
\end{aligned}$$

The $\langle v,\ b''\rangle$ lookup skips past the $[\langle f_i,\ b'\rangle \mapsto \mathcal{A}\ l_i\ \beta'\ ve]$ environment update because either $v \neq f_i$ or $\hat{b} \neq |b'|$. In the first case, the lookup obviously skips the update. In the second case, $|b''| = \hat{b}$, so $|b''| \neq |b'|$, and therefore $b'' \neq b'$ — also causing the lookup to skip the update.

This completes the recursive proof that $\widehat{\mathcal{C}}$ conservatively approximates $\mathcal{C}$.

Q.E.D.

Lemma 9 Proof of Inductive Case ($\widehat{\mathcal{F}}$ conservatively approximates $\mathcal{F}$)
Recursion assumption: $\mathcal{F}$ and $\widehat{\mathcal{F}}$ are defined in terms of $\mathcal{C}$ and $\widehat{\mathcal{C}}$ functions for which lemma 8 holds, and $\mathcal{F}$ and $\widehat{\mathcal{F}}$ functions for which lemma 9 holds.

The argument f is either a closure, the stop continuation, or a primop. We will show each case in turn. We may assume for all the cases that none of the run-time error checks apply. Otherwise, $\mathcal{F}$ aborts with the empty cache [ ], and the lemma holds trivially.

Assume the procedure is a closure: $f = \langle [\![\ell\!:\!(\lambda\ (v_1 \ldots v_n)\ c)]\!],\ \beta\rangle$.

$$\begin{aligned}
\mathcal{F}\ \langle\langle \ell,\ \beta\rangle,\ av,\ ve,\ b,\ \sigma\rangle\ \ cc &= \mathcal{C}\ \langle c,\ \beta[\ell \mapsto b],\ ve[\langle v_i,\ b\rangle \mapsto av{\downarrow}i],\ b,\ \sigma\rangle\ \ cc \\
\widehat{\mathcal{F}}\ \langle\langle \ell,\ |\beta|\rangle,\ \widehat{av},\ \widehat{ve},\ |b|,\ \widehat{\sigma}\rangle\ \ |cc| &= \widehat{\mathcal{C}}\ \langle c,\ |\beta|\ [\ell \mapsto |b|],\ \widehat{ve} \sqcup [\langle v_i,\ |b|\rangle \mapsto \widehat{av}{\downarrow}i],\ |b|,\ \widehat{\sigma}\rangle\ \ |cc|
\end{aligned}$$

The result follows from (1) $|\beta[\ell \mapsto b]| = |\beta|\,[\ell \mapsto |b|]$, (2) $\widehat{ve} \sqcup [\langle v_i,\, |b|\rangle \mapsto \widehat{av}{\downarrow}i] \sqsupseteq |ve[\langle v_i,\, b\rangle \mapsto av{\downarrow}i]|$ (shown below), and (3) the recursion assumption on $\mathcal{C}$ and $\widehat{\mathcal{C}}$.

To see that

$$\widehat{ve} \sqcup [\langle v_i,\, |b|\rangle \mapsto \widehat{av}{\downarrow}i] \sqsupseteq |ve[\langle v_i,\, b\rangle \mapsto av{\downarrow}i]|\,,$$

let $\hat{g}$ be the expression on the left, and $\hat{h}$ be the expression on the right.

<u>**a.** $\hat{g}\ \langle v_i,\, |b|\rangle \sqsupseteq \hat{h}\ \langle v_i,\, |b|\rangle$</u>

$$\begin{aligned}\hat{g}\ \langle v_i,\, |b|\rangle &= \widehat{av}{\downarrow}i \sqcup \widehat{ve}\ \langle v_i,\, |b|\rangle \sqsupseteq |av{\downarrow}i| \sqcup |ve|\ \langle v_i,\, |b|\rangle \\ &= |av{\downarrow}i| \sqcup \bigsqcup_{|b'|=|b|} |ve\ \langle v_i,\, b'\rangle| \sqsupseteq |av{\downarrow}i| \sqcup \bigsqcup_{\substack{|b'|=|b| \\ b'\neq b}} |ve\ \langle v_i,\, b'\rangle|\end{aligned}$$

$$\hat{h}\ \langle v_i,\, |b|\rangle = \bigsqcup_{|b'|=|b|} |ve[\langle v_i,\, b\rangle \mapsto av{\downarrow}i]\ \langle v_i,\, b'\rangle| = |av{\downarrow}i| \sqcup \bigsqcup_{\substack{|b'|=|b| \\ b'\neq b}} |ve\ \langle v_i,\, b'\rangle|$$

Therefore, $\hat{g}\ \langle v_i,\, |b|\rangle \sqsupseteq \hat{h}\ \langle v_i,\, |b|\rangle$.

<u>**b.** $\hat{g}\ \left\langle v,\, \hat{b}\right\rangle \sqsupseteq \hat{h}\ \left\langle v,\, \hat{b}\right\rangle$ **for** $\left\langle v,\, \hat{b}\right\rangle \neq \langle v_i,\, |b|\rangle$</u>

$$\hat{g}\ \left\langle v,\, \hat{b}\right\rangle = (\widehat{ve} \sqcup [\langle v_i,\, |b|\rangle \mapsto \widehat{av}{\downarrow}i])\ \left\langle v,\, \hat{b}\right\rangle = \widehat{ve}\ \left\langle v,\, \hat{b}\right\rangle \sqsupseteq |ve|\ \left\langle v,\, \hat{b}\right\rangle$$

$$\hat{h}\ \left\langle v,\, \hat{b}\right\rangle = \bigsqcup_{|b'|=\hat{b}} |ve[\langle v_i,\, b\rangle \mapsto av{\downarrow}i]\ \langle v,\, b'\rangle| = \bigsqcup_{|b'|=\hat{b}} |ve\ \langle v,\, b'\rangle| = |ve|\ \left\langle v,\, \hat{b}\right\rangle$$

Therefore, $\hat{g}\ \left\langle v,\, \hat{b}\right\rangle \sqsupseteq \hat{h}\ \left\langle v,\, \hat{b}\right\rangle$.

Alternatively, the procedure could be the *stop* continuation. This is a trivial case, since $\mathcal{F}$ produces the empty call cache which abstracts to the empty abstract call cache that $\widehat{\mathcal{F}}$ returns.

Finally, the procedure could be one of the primops: `+`, `if`, `new`, `contents`, and `set`. Suppose that $f = [\![p{:}\texttt{+}]\!]$. Let

$$\begin{aligned}\hat{\gamma} &= \widehat{\mathcal{F}}\ \left\langle [\![p{:}\texttt{+}]\!],\, \left\langle \hat{x},\, \hat{y},\, \hat{k}\right\rangle,\, \widehat{ve},\, |b|,\, \widehat{\sigma}\right\rangle \\ &= \left[\left\langle ic_{p:\texttt{+}},\, |\beta|\right\rangle \mapsto \hat{k}\right] \sqcup \bigsqcup_{k'\in\hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\, \langle\bot\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic_{p:\texttt{+}},\, \widehat{\sigma}\right\rangle \\ &\sqsupseteq \left[\left\langle ic_{p:\texttt{+}},\, |\beta|\right\rangle \mapsto |k|\right] \sqcup \bigsqcup_{k'\in|k|} \widehat{\mathcal{F}}\ \left\langle k',\, \langle\bot\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic_{p:\texttt{+}},\, \widehat{\sigma}\right\rangle \\ &= \left[\left\langle ic_{p:\texttt{+}},\, |\beta|\right\rangle \mapsto |k|\right] \sqcup \widehat{\mathcal{F}}\ \left\langle |k|_{\text{Proc}},\, \langle\bot\rangle,\, \widehat{ve},\, \left|nb\, b\, ic_{p:\texttt{+}}\right|,\, \widehat{\sigma}\right\rangle\end{aligned}$$

$$\begin{aligned}\gamma &= \mathcal{F}\ \langle [\![p{:}\texttt{+}]\!],\, \langle x,\, y,\, k\rangle,\, ve,\, b,\, \sigma\rangle \\ &= (\mathcal{F}\ \left\langle k,\, \langle x+y\rangle,\, ve,\, nb\, b\, ic_{p:\texttt{+}},\, \sigma\right\rangle)\left[\left\langle ic_{p:\texttt{+}},\, \beta\right\rangle \mapsto k\right],\end{aligned}$$

where $\beta = [[\![p{:}\texttt{+}]\!] \mapsto b]$, so that $|\beta| = [[\![p{:}\texttt{+}]\!] \mapsto |b|]$. We want to show that $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

$\underline{cc = \left\langle ic_{p:\texttt{+}},\, \beta \right\rangle}$

$\hat{\gamma}\ |cc| \sqsupseteq |k|$ and $|\gamma\ cc| = |k|$.

$\underline{cc \neq \left\langle ic_{p:\texttt{+}},\, \beta \right\rangle}$

$$\hat{\gamma}\ |cc| \sqsupseteq \widehat{\mathcal{F}}\ \left\langle |k|_{\text{Proc}},\, \langle\bot\rangle,\, \widehat{ve},\, \left|nb\ b\ ic_{p:\texttt{+}}\right|,\, \widehat{\sigma} \right\rangle\ |cc|$$
$$|\gamma\ cc| = \left|\mathcal{F}\ \left\langle k,\, \langle x+y\rangle,\, ve,\, nb\ b\ ic_{p:\texttt{+}},\, \sigma \right\rangle\ cc\right|$$

By $\widehat{\mathcal{F}}$ c.a. $\mathcal{F}$, $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

Showing the lemma for the rest of the primops involves minor variations on the + case. Suppose that $f = [\![p{:}\texttt{if}]\!]$. Let

$$\begin{aligned}
\hat{\gamma} &= \widehat{\mathcal{F}}\ \left\langle [\![p{:}\texttt{if}]\!],\, \left\langle \hat{x},\, \hat{k}_0,\, \hat{k}_1 \right\rangle,\, \widehat{ve},\, |b|,\, \widehat{\sigma} \right\rangle \\
&= \left(\bigsqcup_{f \in \hat{k}_0} \widehat{\mathcal{F}}\ \left\langle f,\, \langle\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic^0_{p:\texttt{if}},\, \widehat{\sigma} \right\rangle \right) \sqcup \left(\bigsqcup_{f \in \hat{k}_1} \widehat{\mathcal{F}}\ \left\langle f,\, \langle\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic^1_{p:\texttt{if}},\, \widehat{\sigma} \right\rangle \right) \\
&\qquad \sqcup \left[\left\langle ic^0_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto \hat{k}_0,\, \left\langle ic^1_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto \hat{k}_1 \right] \\
&\sqsupseteq \left(\bigsqcup_{f \in |k_0|} \widehat{\mathcal{F}}\ \left\langle f,\, \langle\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic^0_{p:\texttt{if}},\, \widehat{\sigma} \right\rangle \right) \sqcup \left(\bigsqcup_{f \in |k_1|} \widehat{\mathcal{F}}\ \left\langle f,\, \langle\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic^1_{p:\texttt{if}},\, \widehat{\sigma} \right\rangle \right) \\
&\qquad \sqcup \left[\left\langle ic^0_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto |k_0|,\, \left\langle ic^1_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto |k_1| \right] \\
&= \widehat{\mathcal{F}}\ \left\langle |k_0|_{\text{Proc}},\, \langle\rangle,\, \widehat{ve},\, \left|nb\ b\ ic^0_{p:\texttt{if}}\right|,\, \widehat{\sigma} \right\rangle \sqcup \widehat{\mathcal{F}}\ \left\langle |k_1|_{\text{Proc}},\, \langle\rangle,\, \widehat{ve},\, \left|nb\ b\ ic^1_{p:\texttt{if}}\right|,\, \widehat{\sigma} \right\rangle \\
&\qquad \sqcup \left[\left\langle ic^0_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto |k_0|,\, \left\langle ic^1_{p:\texttt{if}},\, |\beta| \right\rangle \mapsto |k_1| \right]
\end{aligned}$$

$$\begin{aligned}
\gamma &= \mathcal{F}\ \langle [\![p{:}\texttt{if}]\!],\, \langle x,\, k_0,\, k_1\rangle,\, ve,\, b,\, \sigma\rangle \\
&= x \neq \text{false} \ \longrightarrow\ \mathcal{F}\ \left\langle k_0,\, \langle\rangle,\, ve,\, nb\ b\ ic^0_{p:\texttt{if}},\, \sigma \right\rangle \left[\left\langle ic^0_{p:\texttt{if}},\, \beta \right\rangle \mapsto k_0 \right] \\
&\quad\ \text{otherwise}\ \mathcal{F}\ \left\langle k_1,\, \langle\rangle,\, ve,\, nb\ b\ ic^1_{p:\texttt{if}},\, \sigma \right\rangle \left[\left\langle ic^1_{p:\texttt{if}},\, \beta \right\rangle \mapsto k_1 \right],
\end{aligned}$$

where $\beta = [[\![p{:}\texttt{if}]\!] \mapsto b]$, so that $|\beta| = [[\![p{:}\texttt{if}]\!] \mapsto |b|]$. We want to show that $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$. We have to be a little bit careful in manipulating the various subexpressions of the exact cache returned for `if`. Exact caches are partial functions, and although cc is assumed to lie within $Dom(\gamma)$, that does not imply that we can apply the caches for both arms of the `if` to cc — it is possibly in the domain of only one of these arms. The following development is careful to handle this small complexity.

$\underline{cc = \left\langle ic^0_{p:\texttt{if}},\, \beta \right\rangle}$

In this case, we are looking up the call to the true continuation k_0.

$$\hat{\gamma}\ |cc| = \hat{\gamma}\ \left\langle ic^0_{p:\texttt{if}},\, |\beta| \right\rangle \sqsupseteq |k_0| \sqcup \widehat{\mathcal{F}}\ \left\langle |k_1|_{\text{Proc}},\, \langle\rangle,\, \widehat{ve},\, \left|nb\ b\ ic^0_{p:\texttt{if}}\right|,\, \widehat{\sigma} \right\rangle\ |cc|$$

Suppose that $x \neq$ false. Then $|\gamma\ cc| = |k_0|$, and the conservative inequality holds. Alternatively, suppose that $x =$ false. Then

$$\begin{aligned}|\gamma\ cc| &= \left|\left(\mathcal{F}\ \left\langle k_1,\ \langle\rangle,\ ve,\ nb\, b\, ic^0_{p:\texttt{if}},\ \sigma\right\rangle \left[\left\langle ic^1_{p:\texttt{if}},\ \beta\right\rangle \mapsto k_1\right]\right)\ cc\right| \\ &= \left|\mathcal{F}\ \left\langle k_1,\ \langle\rangle,\ ve,\ nb\, b\, ic^0_{p:\texttt{if}},\ \sigma\right\rangle\ cc\right|,\end{aligned}$$

and the result follows from the recursion assumption on $\mathcal{F}$ and $\widehat{\mathcal{F}}$.

$cc = \left\langle ic^1_{p:\texttt{if}},\ \beta\right\rangle$

In this case, we are looking up the call to the false continuation k_1. This case is analagous to the previous case for the true continuation.

$cc \neq \left\langle ic^0_{p:\texttt{if}},\ \beta\right\rangle, \left\langle ic^1_{p:\texttt{if}},\ \beta\right\rangle$

The final case involves looking up some call other than the two immediate potential calls to the `if` primop's continuations.

$$\begin{aligned}\hat{\gamma}\ |cc| \sqsupseteq\ & \widehat{\mathcal{F}}\ \left\langle |k_0|_{\text{Proc}},\ \langle\rangle,\ \widehat{ve},\ \left|nb\, b\, ic^0_{p:\texttt{if}}\right|,\ \widehat{\sigma}\right\rangle\ |cc| \\ & \sqcup \widehat{\mathcal{F}}\ \left\langle |k_1|_{\text{Proc}},\ \langle\rangle,\ \widehat{ve},\ \left|nb\, b\, ic^1_{p:\texttt{if}}\right|,\ \widehat{\sigma}\right\rangle |cc|\end{aligned}$$

Suppose that $x \neq$ false. Then

$$|\gamma\ cc| = \left|\mathcal{F}\ \left\langle k_0,\ \langle\rangle,\ ve,\ nb\, b\, ic^0_{p:\texttt{if}},\ \sigma\right\rangle\ cc\right|,$$

and the result follows from the recursion assumption on $\mathcal{F}$ and $\widehat{\mathcal{F}}$. Alternatively, suppose that $x =$ false. Then

$$|\gamma\ cc| = \left|\mathcal{F}\ \left\langle k_1,\ \langle\rangle,\ ve,\ nb\, b\, ic^1_{p:\texttt{if}},\ \sigma\right\rangle\ cc\right|.$$

Again, the result follows from the recursion assumption on $\mathcal{F}$ and $\widehat{\mathcal{F}}$.

Suppose that $f = [\![p\texttt{:new}]\!]$. Let

$$\begin{aligned}\hat{\gamma} = \widehat{\mathcal{F}}\ &\left\langle [\![p\texttt{:new}]\!],\ \left\langle \hat{d},\ \hat{k}\right\rangle,\ \widehat{ve},\ |b|,\ \widehat{\sigma}\right\rangle = \\ &\left[\left\langle ic_{p:\texttt{new}},\ |\beta|\right\rangle \mapsto \hat{k}\right] \sqcup \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\ \langle\bot\rangle,\ \widehat{ve},\ \widehat{nb}\ |b|\ ic_{p:\texttt{new}},\ (\widehat{\sigma} \sqcup \hat{d})\right\rangle\end{aligned}$$

$$\begin{aligned}\gamma = \mathcal{F}\ &\langle [\![p\texttt{:new}]\!],\ \langle d,\ k\rangle,\ ve,\ b,\ \sigma\rangle = \\ &\mathcal{F}\ \left\langle k,\ \langle a\rangle,\ ve,\ nb\, b\, ic_{p:\texttt{new}},\ \sigma[a \mapsto d]\right\rangle \left[\left\langle ic_{p:\texttt{new}},\ \beta\right\rangle \mapsto k\right],\end{aligned}$$

where $\beta = [[\![p\texttt{:new}]\!] \mapsto b]$ (so that $|\beta| = [[\![p\texttt{:new}]\!] \mapsto |b|]$) and $a = alloc\, \sigma$. We want to show that $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

$cc = \left\langle ic_{p:\texttt{new}},\ \beta\right\rangle$

$\hat{\gamma}\ |cc| \sqsupseteq \hat{k} \sqsupseteq |k|$ and $\left|\hat{h}\ cc\right| = |k|$.

$\underline{cc \neq \left\langle ic_{p:\texttt{new}},\, \beta \right\rangle}$

$$\begin{aligned} \hat{\gamma}\ |cc| &\sqsupseteq \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\, \langle \bot \rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic_{p:\texttt{new}},\, (\widehat{\sigma} \sqcup \hat{d}) \right\rangle\ |cc| \\ &\sqsupseteq \widehat{\mathcal{F}}\ \left\langle |k|_{\mathrm{Proc}},\, \langle \bot \rangle,\, \widehat{ve},\, \left| nb\, b\, ic_{p:\texttt{new}} \right|,\, (\widehat{\sigma} \sqcup \hat{d}) \right\rangle\ |cc| \\ |\gamma\ cc| &= \left| \mathcal{F}\ \left\langle k,\, \langle a \rangle,\, ve,\, nb\, b\, ic_{p:\texttt{new}},\, \sigma[a \mapsto d] \right\rangle\ cc \right| \end{aligned}$$

The result follows from $\widehat{\mathcal{F}}$ c.a. $\mathcal{F}$ and $\widehat{\sigma} \sqcup \hat{d} \sqsupseteq |\sigma[a \mapsto d]|$.

To see that $\widehat{\sigma} \sqcup \hat{d} \sqsupseteq |\sigma[a \mapsto d]|$:

$$\begin{aligned} |\sigma[a \mapsto d]| &= \bigsqcup |\mathit{Im}(\sigma[a \mapsto d])| \sqsubseteq \bigsqcup |\mathit{Im}(\sigma) \cup \{d\}| \\ &= \bigsqcup |\mathit{Im}(\sigma)| \sqcup |d| = |\sigma| \sqcup |d| \sqsubseteq \widehat{\sigma} \sqcup \hat{d} \end{aligned}$$

Suppose that $f = [\![p{:}\texttt{set}]\!]$. Let

$$\begin{aligned} \hat{\gamma} = \widehat{\mathcal{F}}\ &\left\langle [\![p{:}\texttt{set}]\!],\, \left\langle \hat{a},\, \hat{d},\, \hat{k} \right\rangle,\, \widehat{ve},\, |b|,\, \widehat{\sigma} \right\rangle = \\ &\left[\left\langle ic_{p:\texttt{set}},\, |\beta| \right\rangle \mapsto \hat{k} \right] \sqcup \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\, \langle\rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic_{p:\texttt{set}},\, (\widehat{\sigma} \sqcup \hat{d}) \right\rangle \end{aligned}$$

$$\begin{aligned} \gamma = \mathcal{F}\ &\langle [\![p{:}\texttt{set}]\!],\, \langle a,\, d,\, k \rangle,\, ve,\, b,\, \sigma \rangle = \\ &(\mathcal{F}\ \left\langle k,\, \langle\rangle,\, ve,\, nb\, b\, ic_{p:\texttt{set}},\, \sigma[a \mapsto d] \right\rangle) \left[\left\langle ic_{p:\texttt{set}},\, \beta \right\rangle \mapsto k \right] \end{aligned}$$

where $\beta = [[\![p{:}\texttt{set}]\!] \mapsto b]$, so that $|\beta| = [[\![p{:}\texttt{set}]\!] \mapsto |b|]$. We want to show that $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

$\underline{cc = \left\langle ic_{p:\texttt{set}},\, \beta \right\rangle}$

$\hat{\gamma}\ |cc| \sqsupseteq \hat{k} \sqsupseteq |k|$ and $|\gamma\ cc| = |k|$.

$\underline{cc \neq \left\langle ic_{p:\texttt{set}},\, \beta \right\rangle}$

$$\begin{aligned} \hat{\gamma}\ |cc| &\sqsupseteq \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\, \langle\rangle,\, \widehat{ve},\, \left| nb\, b\, ic_{p:\texttt{set}} \right|,\, (\widehat{\sigma} \sqcup \hat{d}) \right\rangle\ |cc| \\ &\sqsupseteq \widehat{\mathcal{F}}\ \left\langle |k|_{\mathrm{Proc}},\, \langle\rangle,\, \widehat{ve},\, \left| nb\, b\, ic_{p:\texttt{set}} \right|,\, (\widehat{\sigma} \sqcup \hat{d}) \right\rangle\ |cc| \\ |\gamma\ cc| &= \left| \mathcal{F}\ \left\langle k,\, \langle\rangle,\, ve,\, nb\, b\, ic_{p:\texttt{set}},\, \sigma[a \mapsto d] \right\rangle\ cc \right| \end{aligned}$$

The result follows from $\widehat{\mathcal{F}}$ c.a. $\mathcal{F}$ and (as in the case of `new`) $\widehat{\sigma} \sqcup \hat{d} \sqsupseteq |\sigma[a \mapsto d]|$.

Suppose that $f = [\![p{:}\texttt{contents}]\!]$. Let

$$\begin{aligned} \hat{\gamma} = \widehat{\mathcal{F}}\ &\left\langle [\![p{:}\texttt{contents}]\!],\, \left\langle \hat{a},\, \hat{k} \right\rangle,\, \widehat{ve},\, |b|,\, \widehat{\sigma} \right\rangle = \\ &\left[\left\langle ic_{p:\texttt{contents}},\, |\beta| \right\rangle \mapsto \hat{k} \right] \sqcup \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\, \langle \widehat{\sigma} \rangle,\, \widehat{ve},\, \widehat{nb}\ |b|\ ic_{p:\texttt{contents}},\, \widehat{\sigma} \right\rangle \end{aligned}$$

$$\gamma = \mathcal{F}\ \langle [\![p\text{:}\mathtt{contents}]\!],\ \langle a,\ k\rangle,\ ve,\ b,\ \sigma\rangle =$$
$$(\mathcal{F}\ \left\langle k,\ \langle \sigma\ a\rangle,\ ve,\ nb\ b\ ic_{p:\mathtt{contents}},\ \sigma\right\rangle)\left[\left\langle ic_{p:\mathtt{contents}},\ \beta\right\rangle \mapsto k\right],$$

where $\beta = [[\![p\text{:}\mathtt{contents}]\!] \mapsto b]$, so that $|\beta| = [[\![p\text{:}\mathtt{contents}]\!] \mapsto |b|]$. We want to show that $\hat{\gamma}\ |cc| \sqsupseteq |\gamma\ cc|$.

$\underline{cc = \left\langle ic_{p:\mathtt{contents}},\ \beta\right\rangle}$

$\hat{\gamma}\ |cc| \sqsupseteq \hat{k} \sqsupseteq |k|$ and $|\gamma\ cc| = |k|$.

$\underline{cc \neq \left\langle ic_{p:\mathtt{contents}},\ \beta\right\rangle}$

$$\begin{aligned}
\hat{\gamma}\ |cc| &\sqsupseteq \bigsqcup_{k' \in \hat{k}} \widehat{\mathcal{F}}\ \left\langle k',\ \langle\hat{\sigma}\rangle,\ \widehat{ve},\ \left|nb\ b\ ic_{p:\mathtt{contents}}\right|,\ \hat{\sigma}\right\rangle\ |cc| \\
&\sqsupseteq \bigsqcup_{k' \in |k|} \widehat{\mathcal{F}}\ \left\langle k',\ \langle\hat{\sigma}\rangle,\ \widehat{ve},\ \left|nb\ b\ ic_{p:\mathtt{contents}}\right|,\ \hat{\sigma}\right\rangle\ |cc| \\
&= \widehat{\mathcal{F}}\ \left\langle |k|_{\text{Proc}},\ \langle\hat{\sigma}\rangle,\ \widehat{ve},\ \left|nb\ b\ ic_{p:\mathtt{contents}}\right|,\ \hat{\sigma}\right\rangle\ |cc| \\
|\gamma\ cc| &= \left|\mathcal{F}\ \left\langle k,\ \langle\sigma\ a\rangle,\ ve,\ nb\ b\ ic_{p:\mathtt{contents}},\ \sigma\right\rangle\ cc\right|
\end{aligned}$$

$|\sigma\ a| \sqsubseteq |\sigma| \sqsubseteq \hat{\sigma}$, and the result follows from $\widehat{\mathcal{F}}$ c.a. $\mathcal{F}$.

This completes the recursive proof that $\widehat{\mathcal{F}}$ conservatively approximates $\mathcal{F}$.

Q.E.D.

Proofs of the Base and Limit Cases

Recall from section 4.3 that the $\mathcal{C}$ and $\mathcal{F}$ functions are defined as the limit of a sequence of approximations $\langle\ \langle \mathcal{C}_i,\ \mathcal{F}_i\rangle \mid i \geq 0\ \rangle$ generated by the H functional, such that $\langle \mathcal{C}_i,\ \mathcal{F}_i\rangle = H^i\ \perp$. Similarly, $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are the limit of an approximation sequence $\left\langle\ \left\langle \widehat{\mathcal{C}}_i,\ \widehat{\mathcal{F}}_i\right\rangle \mid i \geq 0\ \right\rangle$, generated by some $\widehat{H}$ functional. We prove properties of $\mathcal{C}$, $\mathcal{F}$, $\widehat{\mathcal{C}}$, and $\widehat{\mathcal{F}}$ with the technique of inductive predicates. With this technique, a complete proof of the two previous lemmas involves showing the lemma holds in three cases:

- The base case.
  That is, showing that the lemmas hold on $\left\langle \mathcal{C}_0,\ \mathcal{F}_0,\ \widehat{\mathcal{C}}_0,\ \widehat{\mathcal{F}}_0\right\rangle$.
- The inductive case.
  That is, showing that if the lemmas hold on $\left\langle \mathcal{C}_i,\ \mathcal{F}_i,\ \widehat{\mathcal{C}}_i,\ \widehat{\mathcal{F}}_i\right\rangle$, then they hold on $\left\langle \mathcal{C}_{i+1},\ \mathcal{F}_{i+1},\ \widehat{\mathcal{C}}_{i+1},\ \widehat{\mathcal{F}}_{i+1}\right\rangle$.
- The limit case.
  That is, showing that if the lemmas hold over a chain $\left\langle\ \left\langle \mathcal{C}_i,\ \mathcal{F}_i,\ \widehat{\mathcal{C}}_i,\ \widehat{\mathcal{F}}_i\right\rangle \mid i \geq 0\ \right\rangle$, then they hold on the limit of the chain $\bigsqcup_{i \geq 0} \left\langle \mathcal{C}_i,\ \mathcal{F}_i,\ \widehat{\mathcal{C}}_i,\ \widehat{\mathcal{F}}_i\right\rangle$.

The recursive proofs we've just completed establish the inductive case. We still need to show the lemmas hold in the base and limit cases.

Proof of Base Case
Our quadruples of functions $\left\langle \mathcal{C}_i,\, \mathcal{F}_i,\, \widehat{\mathcal{C}}_i,\, \widehat{\mathcal{F}}_i \right\rangle$ are taken from the cpo

$$(\text{CState} \to \text{CCache}) \times (\text{FState} \to \text{CCache}) \times (\widehat{\text{CState}} \to \widehat{\text{CCache}}) \times (\widehat{\text{FState}} \to \widehat{\text{CCache}}).$$

The ordering on this cpo is pointwise over each element of the quadruple. So the bottom element of the cpo is

$$\left\langle \mathcal{C}_0,\, \mathcal{F}_0,\, \widehat{\mathcal{C}}_0,\, \widehat{\mathcal{F}}_0 \right\rangle = \left\langle \lambda cs.\ [\,],\, \lambda fs.\ [\,],\, \lambda \widehat{cs}.\ [\,],\, \lambda \widehat{fs}.\ [\,] \right\rangle,$$

where the exact empty call caches from CCache are empty partial functions (*i.e.*, $[\,] = \emptyset$), and the approximate empty call caches from $\widehat{\text{CCache}}$ are total functions mapping all inputs to the empty set (*i.e.*, $[\,] = \lambda cc.\ \emptyset$).

To establish the base case for lemmas 8 and 9, we must show they hold for the bottom tuple of functions $\left\langle \mathcal{C}_0,\, \mathcal{F}_0,\, \widehat{\mathcal{C}}_0,\, \widehat{\mathcal{F}}_0 \right\rangle$. This is trivial. The empty exact call cache $[\,]$ that $\mathcal{C}_0$ and $\mathcal{F}_0$ return in all cases is the empty partial function, with domain $Dom([\,]) = \emptyset$. The conservative approximation definitions (definition 3) restrict the comparison of the call caches to the domain of the exact call cache, which is empty. So the lemma trivially holds.

Q.E.D.

Proof of the Limit Case
Suppose that $\left\langle \left\langle \mathcal{C}_i,\, \mathcal{F}_i,\, \widehat{\mathcal{C}}_i,\, \widehat{\mathcal{F}}_i \right\rangle \mid i \geq 0 \right\rangle$ is a chain from our cpo of function quadruples such that lemmas 8 and 9 hold for each element of the chain. We need to show that the lemmas hold on the limit $\left\langle \mathcal{C}_\infty,\, \mathcal{F}_\infty,\, \widehat{\mathcal{C}}_\infty,\, \widehat{\mathcal{F}}_\infty \right\rangle = \bigsqcup_i \left\langle \mathcal{C}_i,\, \mathcal{F}_i,\, \widehat{\mathcal{C}}_i,\, \widehat{\mathcal{F}}_i \right\rangle$.

- **$\widehat{\mathcal{C}}_\infty$ conservatively approximates $\mathcal{C}_\infty$ (Lemma 8):**
  Suppose that $cc \in Dom(\mathcal{C}_\infty\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle)$.

$$\begin{aligned}
\widehat{\mathcal{C}}_\infty\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \widehat{\sigma}\rangle\ |cc| &= \left(\bigsqcup_i \widehat{\mathcal{C}}_i\ \langle c,\, |\beta|,\, \widehat{ve},\, |b|,\, \widehat{\sigma}\rangle\right)\ |cc| \\
&\sqsupseteq \left(\bigsqcup_i |\mathcal{C}_i\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle|\right)\ |cc| \\
&= \left(\bigsqcup_i |\gamma_i|\right)\ |cc|,
\end{aligned}$$

  where $\gamma_i = \mathcal{C}_i\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle$.

$$\begin{aligned}
|\mathcal{C}_\infty\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\ cc| &= \left|\left(\bigsqcup_i \mathcal{C}_i\ \langle c,\, \beta,\, ve,\, b,\, \sigma\rangle\right)\ cc\right| \\
&= \left|\left(\bigsqcup_i \gamma_i\right)\ cc\right|
\end{aligned}$$

  Now,

$$\left(\bigsqcup_i |\gamma_i|\right)\ |cc| \sqsupseteq \left|\left(\bigsqcup_i \gamma_i\right)\ cc\right|,$$

so the lemma holds. To see the above inequality, first note that since $\langle\, \mathcal{C}_i \mid i \geq 0 \,\rangle$ is a chain, applying the $\mathcal{C}_i$ functions to a common argument produces a chain of results. So $\langle\, \gamma_i \mid i \geq 0 \,\rangle$ is also a chain. Now, the γ_i are partial functions, ordered by inclusion. If $cc \in Dom(\mathcal{C}_\infty\ \langle c,\ \beta,\ ve,\ b,\ \sigma\rangle)$, and $\mathcal{C}_\infty\ \langle c,\ \beta,\ ve,\ b,\ \sigma\rangle = \bigsqcup_{i\geq 0} \gamma_i$, then $cc \in \bigcup_{i\geq 0} Dom(\gamma_i)$. Let n be the smallest integer such that $cc \in Dom(\gamma_n)$. Since the γ_i are a chain, $\forall j \geq n$, $cc \in Dom(\gamma_j)$ and $\gamma_j\ cc = \gamma_n\ cc$. So $\gamma_n\ cc$ is also the value produced by applying the chain's limit, $(\bigsqcup_{i\geq 0} \gamma_i)\ cc$. The result follows from the chain of inequalities:

$$\left(\bigsqcup_{i\geq 0} |\gamma_i|\right)\ |cc| = \bigsqcup_{i\geq 0}(|\gamma_i|\ \ |cc|) = \bigsqcup_{i\geq 0}\ \bigsqcup_{|cc'|=|cc|} |\gamma_i\ cc'|$$
$$\sqsupseteq \bigsqcup_{i\geq n}\ \bigsqcup_{|cc'|=|cc|} |\gamma_i\ cc'| \sqsupseteq \bigsqcup_{i\geq n} |\gamma_i\ cc| = \left|\left(\bigsqcup_{i\geq 0} \gamma_i\right)\ cc\right|.$$

- $\widehat{\mathcal{F}}_\infty$ **conservatively approximates** $\mathcal{F}_\infty$ **(Lemma 9):**

$$\begin{aligned}
\widehat{\mathcal{F}}_\infty\ \langle |f|,\ \widehat{av},\ \widehat{ve},\ |b|,\ \widehat{\sigma}\rangle\ \ |cc| &= \left(\bigsqcup_i \widehat{\mathcal{F}}_i\ \langle |f|,\ \widehat{av},\ \widehat{ve},\ |b|,\ \widehat{\sigma}\rangle\right)\ |cc| \\
&\sqsupseteq \left(\bigsqcup_i |\mathcal{F}_i\ \langle f,\ av,\ ve,\ b,\ \sigma\rangle|\right)\ |cc| \\
&= \left(\bigsqcup_i |\gamma_i|\right)\ |cc|,
\end{aligned}$$

where $\gamma_i = \mathcal{F}_i\ \langle f,\ av,\ ve,\ b,\ \sigma\rangle$.

$$\begin{aligned}
|\mathcal{F}_\infty\ \langle f,\ av,\ ve,\ b,\ \sigma\rangle\ \ cc| &= \left|\left(\bigsqcup_i \mathcal{F}_i\ \langle f,\ av,\ ve,\ b,\ \sigma\rangle\right)\ cc\right| \\
&= \left|\left(\bigsqcup_i \gamma_i\right)\ cc\right|
\end{aligned}$$

With reasoning identical to the $\mathcal{C}$ case above,

$$\left(\bigsqcup_i |\gamma_i|\right)\ |cc| \sqsupseteq \left|\left(\bigsqcup_i \gamma_i\right)\ cc\right|,$$

and thus the limit case holds for lemma 9.

This completes the proofs of lemmas 8 and 9. Q.E.D.

4.4.3 Computability

The whole point of the abstract semantics is that it should be computable. In this subsection, we'll see that the abstract functions $\widehat{\mathcal{PR}}$, $\widehat{\mathcal{C}}$, $\widehat{\mathcal{F}}$, and $\widehat{\mathcal{A}}$ are computable.

$\widehat{\mathcal{A}}$ and $\widehat{\mathcal{PR}}$ are no problem. $\widehat{\mathcal{A}}$ is a function from a finite domain to a finite range; it is defined in terms of computable primitives: a type-case conditional, a pair constructor, and some table lookups. It can be directly implemented as a program. $\widehat{\mathcal{PR}}$ consists of a single call to $\widehat{\mathcal{F}}$; it's computability depends on the computability of $\widehat{\mathcal{F}}$.

Recursive functions $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ require a little more thought to show there are effective procedures for computing them. $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ have similar structure that we can exploit:

- They are functions from finite domains to finite ranges.
- When called on a value, each function makes a local contribution to its result, and recurses on some new argument for another term.

Let's consider the simple form for this class of functions. Suppose we have a function $f\colon X \to A$, for some finite set X and some lattice A, where f is defined by the following recursive equation:

$$f\ x = g\ x \sqcup f\ (r\ x). \tag{4.6}$$

That is, f's action on x is defined by a function g that makes a local contribution $g\ x$ to the answer, and a recursive component, which is specified by the recursion function r. If g, r, and the join operation $\sqcup$ are computable, then there is a simple computable algorithm for computing $f x$.

The key to f's computability is the finiteness of its domain. Consider the set

$$\left\{x, r\ x, r^2\ x, r^3\ x, \ldots\right\}.$$

This is the set of all elements making a contribution to $f\ x$. Because it is a finite set (being a subset of the finite set X), the naive expansion of f's definition:

$$f\ x = g\ x \sqcup g\ (r\ x) \sqcup g\ (r^2\ x) \sqcup g\ (r^3\ x) \sqcup \cdots$$

actually has only a finite set of terms:

$$f\ x = \bigsqcup \left\{g\ \left(r^i\ x\right) \mid i \geq 0\right\}.$$

A simple algorithm to compute these finite terms is:

```
f(x) =
  let S   = ∅,         /* elements already processed */
      ans = ⊥,        /* accumulated answer -- ⨆g(S) */
      loop(y) = if y ∉ S then
                   S := S ∪ {y};
                   ans := ans ⊔ g(y);
                   loop(r(y));
  in loop(x); ans
```

That concludes the intuitive discussion of f. Let's formally prove our claims. We would like to prove that the function

$$f\ x = \bigsqcup_{i \geq 0} g\ \left(r^i\ x\right)$$

is a minimal solution to the recursive definition of f.

Theorem 10 *$f\ x = \bigsqcup_{i \geq 0} g\ \ (r^i\ x)$ is the least solution to equation 4.6.*

Proof:
First, let's show that f satisfies the recursive definition:

$$\begin{aligned} f\ x &= \bigsqcup \left\{ g\ \left(r^i\ x \right) \mid i \geq 0 \right\} \\ &= g\ x \sqcup \bigsqcup \left\{ g\ \left(r^i\ x \right) \mid i \geq 1 \right\} \\ &= g\ x \sqcup f\ \ (r\ x)\,. \end{aligned}$$

Now, we want to show that f is the *least* solution:

$$h\ x = g\ x \sqcup h\ (r\ x) \quad \Rightarrow \quad h \sqsupseteq f.$$

That is, any solution to equation 4.6 must be greater than or equal to f. Suppose that h is a solution to equation 4.6. It is an easy induction to show that for all $j \geq 0$,

$$\begin{aligned} h\ x &= g\ (r^0\ x) \sqcup \cdots \sqcup g\ (r^j\ x) \sqcup h\ (r^j\ x) \\ &\sqsupseteq g\ (r^j\ x). \end{aligned}$$

That is, $h\ x$ is an upper bound for each term of f's expansion, and so

$$h\ x \sqsupseteq f\ x.$$

Q.E.D.

Why do we want a *least* solution? By our ordering on cache functions, a least solution is the one producing the tightest, most accurate caches, subject to the constraints of the equations.

Now that we have a computable solution for the class of finite functions given by equation 4.6, let's return to the specifics of $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$. As given in figures 4.5, 4.6, $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ don't have quite the right form. Instead of being self-recursive, as required by equation 4.6, they are mutually recursive: $\widehat{\mathcal{C}}$ calls $\widehat{\mathcal{F}}$ and *vice versa*. However, it is fairly straightforward to mechanically transform a system of recursively defined functions into a single self-recursive function. As long as the individual recursions and local contributions are monotonic, the results go through. We'll pass over this complexity to simplify the presentation.

We still need to handle one final detail: the recursions in $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ have a branch factor greater than one. For example, when $\widehat{\mathcal{C}}$ is applied to a simple call $[\![(f\ \ a_1 \ldots a_n)]\!]$, it performs a recursive call to $\widehat{\mathcal{F}}$ for each f' in $\widehat{\mathcal{A}}\ f\ \ \widehat{\beta}\ \ \widehat{ve}$. We can generalise the form of equation 4.6 slightly to allow for this:

$$f\ x = g\ x \sqcup \bigsqcup f\ (R\ x), \tag{4.7}$$

where $R{:}X \to \mathcal{P}(X)$ is a function mapping an input x to the set of elements on which f recurses (here we use $f\ (R\ x)$ as shorthand for $\{f\ y \mid y \in R\ x\}$).

Allowing for branching does not change the essence of the proofs we've worked out; we simply move to the power-set relatives of f, g, and R:

$$\underline{g}\,Y = \bigsqcup g\,Y, \qquad \underline{R}\,Y = \bigcup R\,Y,$$

and

$$\underline{f}\,Y = \underline{g}\,Y \sqcup \underline{f}\,(\underline{R}\,Y). \tag{4.8}$$

These related power-set functions have functionality $\underline{g}\colon \mathcal{P}(X) \to A$, $\underline{R}\colon \mathcal{P}(X) \to \mathcal{P}(X)$, and $\underline{f}\colon \mathcal{P}(X) \to A$.

The relations between g, R and $\underline{g}$, $\underline{R}$ cause f and $\underline{f}$ to be related:

$$f\,x = \underline{f}\,\{x\}, \qquad \underline{f}\,Y = \bigsqcup f\,Y. \tag{4.9}$$

Furthermore, $\underline{f}$ satisfies the simple recursion of equation 4.6, which we know (theorem 10) has the least solution

$$\underline{f}\,Y = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,Y).$$

Putting this together with $f\,x = \underline{f}\,\{x\}$ gives us a solution for f:

$$f\,x = \bigsqcup_{i} \underline{g}\,(\underline{R}^i\,\{x\}).$$

It only remains to prove equations 4.9.

Lemma 11 *If f satisfies recursive equation 4.7, then the function $\underline{f}\,Y = \bigsqcup f\,Y$ satisfies recursive equation 4.8.*

Proof:

$$\begin{aligned}\underline{f}\,Y = \bigsqcup f\,Y &= \bigsqcup_{y \in Y} \left[g\,y \sqcup \bigsqcup f\,(R\,y)\right] \\ &= \underline{g}\,Y \sqcup \bigsqcup_{y \in Y} \bigsqcup f\,(R\,y) = \underline{g}\,Y \sqcup \underline{f}\,(\underline{R}\,Y)\end{aligned}$$

Q.E.D.

It should be clear that $\underline{g}$ and $\underline{R}$ distribute over union: $\underline{g}\,(A \cup B) = \underline{g}\,A \sqcup \underline{g}\,B$, and $\underline{R}\,(A \cup B) = \underline{R}\,A \cup \underline{R}\,B$. This implies that powers of $\underline{R}$ also distribute over union: $\underline{R}^i\,(A \cup B) = \underline{R}^i\,A \cup \underline{R}^i\,B$. This all implies, in turn, that the least solution to equation 4.8, $\underline{f}\,Y = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,Y)$, distributes over union.

We can now derive the least solution to equation 4.7:

Theorem 12 *$f\,x = \underline{f}\,\{x\}$ is the least solution to recursive equation 4.7, where $\underline{f}$ is the least solution to recursive equation 4.8, $\underline{f}\,Y = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,Y)$.*

Proof:
(1) f satisfies 4.7:

$$\begin{array}{lll} f\;x = \underline{f}\;\{x\} & = \underline{g}\;\{x\} \sqcup \underline{f}\,(\underline{R}\;\{x\}) & \text{(by eqn. 4.8)} \\ & = g\;x \sqcup \underline{f}\;(R\;x) & \\ & = g\;x \sqcup \bigsqcup_{x' \in R\,x} \underline{f}\;\{x'\} & \text{(by distributivity of } \underline{f}) \\ & = g\;x \sqcup \bigsqcup_{x' \in R\,x} f\;x' & \\ & = g\;x \sqcup \bigsqcup f\;(R\;x) & \end{array}$$

(2) f is least solution to 4.7:
Suppose f' is a solution to 4.7. We want $f \sqsubseteq f'$. $\underline{f'}\;Y = \bigsqcup f'\;Y$ is a solution to 4.8 by lemma 11. So $f\;x = \underline{f}\;\{x\} \sqsubseteq \underline{f'}\;\{x\} = f'\;x$, where the inequality follows from the leastness of $\underline{f}$. Q.E.D.

We have just proved that the least solution for recursive functions with branching is

$$f\;x = \bigsqcup_{i \geq 0} \underline{g}\;\left(\underline{R}^i\;\{x\}\right) = \underline{g}\;\bigcup_{i \geq 0} \underline{R}^i\;\{x\}\,.$$

The finiteness of X again guarantees us a limited number of terms to join together to find the value of the function for a given argument. This expansion can be computed with the following algorithm:

```
f(x) =
  let S   = ∅,        /* elements already processed */
      ans = ⊥,       /* accumulated answer -- ⊔g(S) */
      loop(y) = if y ∉ S then
                   ans := ans ⊔ g(y);
                   S := S ∪ {y};
                   for each z ∈ R(y)
                       loop(z);
  in loop(x); ans
```

For an intuitive explanation of why this procedure computes $\underline{f}$, consider that $\bigcup_{i \geq 0} \underline{R}_i\;\{x\}$ is essentially a breadth-first exploration of the tree rooted at x, where R is the "sons-of" function. $\underline{R}^i\;\{x\}$ is the set of children at depth i in the tree. The procedure `f(x)` is just the equivalent depth-first search.

We will return to a detailed analysis of algorithms for computing our semantic functions in chapter 6.

Chapter 5

Implementation I

That this manages to work is truly remarkable.
— Sussman & Steele

The equations of chapters 3 and 4 define mathematical functions. We need to define algorithms to compute these functions. In this chapter, we'll consider how to do this. I'll start with the basic algorithm for computing the abstract control-flow semantics, and then discuss some tricks that can be played to increase the speed of the algorithms. I won't attempt to rigorously prove that the algorithms are correct; I'll rely mostly on intuitive appeals. Formal rigour will be deferred to the next chapter.

5.1 The Basic Algorithm

When the $\widehat{\mathcal{C}}$ or $\widehat{\mathcal{F}}$ function is applied to an argument, it adds some local contribution to the final answer, and then recurses on some new arguments (actually, it's a mutual recursion: $\widehat{\mathcal{C}}$ calls $\widehat{\mathcal{F}}$, and *vice versa*). The basic algorithm, then, for computing one of these functions is to keep a set of all the arguments to which the function has been applied during the recursive evaluation of the top-level argument. When the function recurses on an argument, it checks to see if it has already been applied to that argument. If so, we have looped: all the contributions arising from this argument and all its recursive successors have already been made, so the function returns immediately. If not, the argument is added to the set of already-seen arguments, the local contribution is made, and the function recurses.

So, ignoring the `letrec` case for simplicity, the $\widehat{\mathcal{C}}$ function can be realised with the following procedure for the 1CFA abstraction:

```
Chat(call,benv,venv,store)
  if <call,benv,venv,store> ∈ S then return; /* Quit if repeat. */
  S := S ∪ {<call,benv,venv,store>}; /* New arg. Memoise it. */
  let ⟦c:(f . args)⟧ = call,          /* Destructure call form. */
      F = Ahat(f,benv,venv),          /* Eval proc subexp. */
      argv = mapAhat(args,benv,venv) /* Ahat(a,benv,venv)
  in                                     for a in args.    */
      ans := ans ⊔ [<c,benv> ↦ F];
      /* Argument c to Fhat is the 1CFA contour: */
      for each f in F do Fhat(f,argv,venv,store,c);
```

With a similar memoising definition of $\widehat{\mathcal{F}}$, the $\widehat{\mathcal{PR}}$ procedure is just:

```
PRhat(lam)
  S := ∅;
  ans := [];
  Fhat(Ahat(lam,[],[]),<{stop}>,[],[],topcall);
  return(S)
```

This algorithm is similar to the "memoised pending analysis" treated by Young and Hudak [Young$^+$ 86].

5.2 Exploiting Monotonicity

We can optimise this procedure by taking advantage of its monotonicity to use a more aggressive cutoff. $\widehat{\mathcal{C}}$'s arguments have a simple elementwise ordering:

$$\left\langle c,\, \widehat{\beta},\, \widehat{ve},\, \widehat{\sigma}\right\rangle \sqsubseteq \left\langle c',\, \widehat{\beta}',\, \widehat{ve}',\, \widehat{\sigma}'\right\rangle \qquad \text{if} \qquad c = c',\, \widehat{\beta} = \widehat{\beta}',\, \widehat{ve} \sqsubseteq \widehat{ve}',\, \text{and } \widehat{\sigma} \sqsubseteq \widehat{\sigma}'.$$

The ordering on $\widehat{\mathcal{F}}$'s arguments is similar.

It's fairly straightforward to see that given these orderings the $\widehat{\mathcal{C}}$, $\widehat{\mathcal{F}}$, and $\widehat{\mathcal{A}}$ functions are all monotonic. Suppose we have a call subform a, contour environment $\widehat{\beta}$, and two related variable environments $\widehat{ve} \sqsubseteq \widehat{ve}'$. It follows that $\widehat{\mathcal{A}}\ a\ \widehat{\beta}\ \widehat{ve}$ must be a subset of $\widehat{\mathcal{A}}\ a\ \widehat{\beta}\ \widehat{ve}'$.

Similarly, suppose that we apply $\widehat{\mathcal{C}}$ to a pair of related arguments: $\left\langle c,\, \widehat{\beta},\, \widehat{ve},\, \widehat{\sigma}\right\rangle$ and $\left\langle c,\, \widehat{\beta},\, \widehat{ve}',\, \widehat{\sigma}'\right\rangle$, where $\widehat{ve} \sqsubseteq \widehat{ve}'$ and $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$. Then it is clear, from the monotonicity of $\widehat{\mathcal{A}}$, that $\widehat{\mathcal{C}}$'s local contribution to the final answer cache for the first, precise argument ($\left[\left\langle c,\, \widehat{\beta}\right\rangle \mapsto \widehat{\mathcal{A}}\ f\ \widehat{\beta}\ \widehat{ve}\right]$) is contained by its contribution for the second, approximate argument ($\left[\left\langle c,\, \widehat{\beta}\right\rangle \mapsto \widehat{\mathcal{A}}\ f\ \widehat{\beta}\ \widehat{ve}'\right]$). Further, for each recursive call to $\widehat{\mathcal{F}}$ made by $\widehat{\mathcal{C}}$ in the first case, there is a recursive call to $\widehat{\mathcal{F}}$ in the second case, with an argument that approximates the argument used in the first case. So all the contributions to the answer cache proceeding from the chains of recursive calls in the first case are covered by the contributions made by the chains of calls in the second case.

We can take advantage of this fact to speed up the algorithm. If we call Chat on some argument tuple <call,benv,venv,store>, and we notice that we've already called Chat on some other argument tuple that approximates <call,benv,venv,store>, then we can cut off the search immediately, since all the contributions produced by the earlier, approximate tuple will cover all the contributions made by the current one.

We can implement this optimisation by changing the single line

```
if <call,benv,venv,store> ∈ S then return;
```

to

```
if <call,benv,venv,store> ⊑ cs for some cs ∈ S then return;
```

in the Chat procedure. Note that the new cutoff is strictly more aggressive than the basic one: since <call,benv,venv,store> ⊑ <call,benv,venv,store>, the aggressive cutoff will trigger whenever the current argument is already in S.

5.3 Time-Stamp Approximations

/ too painful to do right */*
— <stdio.h>, 5.3 (Berkeley)

Even with the aggressive cutoff of the previous section, the function $\widehat{\mathcal{C}}$ is painful to compute. It requires us to maintain and search sets of argument tuples. Since each argument tuple includes a complete abstract variable environment and abstract store, these tuples are sizeable structures. So the space and time expense of computing $\widehat{\mathcal{C}}$ is fairly high.

An alternative is to compute a related function that approximates $\widehat{\mathcal{C}}$ but is much cheaper to compute. This is the function that uses single-threaded variable environment and store data structures.

Suppose we keep the variable environment and store in the global variables venv and store. The variable environment and store are passed to the Chat, Fhat, and Ahat procedures implicitly. For instance, Chat now only takes arguments call and benv, with the store and variable environment taken from their global variables. Whenever a procedure adds information to the variable environment or store, it simply alters the global variables. When a recursive procedure has completed, it makes no effort to restore the values of venv and store before returning to its caller. Thus, all the additions that were made to venv and store will be visible to the procedure's caller.

The reason we can do this is that our error is always going to be on the safe side. Suppose Chat is evaluating some call expression (f 2 5), and f evaluates to a set of three abstract procedures. Then Chat is going to perform three recursive calls to Fhat, one for each of the abstract procedures to which f could possibly be bound. Let's assume that before the first call to Fhat, venv has the correct value. During the first call to Fhat, venv is going to be augmented, so that when Fhat returns, venv will no longer have its original, pre-call value. So when the *second* call to Fhat is made, the variable environment and store won't have their correct values. Instead they'll have values that are approximations to their correct values, because they'll include extra information.

Using the approximations instead of the correct values is allowable because $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are monotonic. This means that if we apply them to approximate arguments, they'll produce an answer cache that is a safe approximation of what we'd have gotten with the exact arguments.

So if we proceed without restoring the `venv` or `store` values around recursive calls, we'll get a result that is a little weaker than the result provided by the basic algorithm of section 5.1, but consistent with it.

What is the advantage of this approximate algorithm? By making the variable environment and store single-threaded, and only changing them monotonically, we can be assured that the temporal sequence of `venv` and `store` values are strictly ordered during the search procedure. This means we can use time stamps to describe the different variable environments and stores. We can compare the variable environments that existed at two different calls to `Chat`, say, simply by comparing their corresponding time stamps. The one with the larger number approximates the one with the smaller number.

All we need to do is introduce global variables `venv-ts` and `store-ts` that are integer counters serving as modification time stamps for `venv` and `store`. Every time `venv` is altered, we increment the `venv-ts` counter; and similarly for `store` and `store-ts`. If an operation on one of these tables doesn't add any new information, we leave its time stamp unchanged.

Now we can memoise the variable environment and store by their time stamps. The argument tuples we store away in our memo table need only contain the time stamps for the variable environment and store: `<call,benv,venv-ts,store-ts>`. This means we don't need to keep around large numbers of variable environments and stores in our memo table, and we can also search the table and compare tuples much faster.

Note further that for any two memoised argument tuples

$$cs_1 = \texttt{<call,benv,venv-ts}_1\texttt{,store-ts}_1\texttt{>},$$
$$cs_2 = \texttt{<call,benv,venv-ts}_2\texttt{,store-ts}_2\texttt{>}$$

with identical call and contour environment components, the tuples must be ordered. That is, either

$$\texttt{venv-ts}_1 \leq \texttt{venv-ts}_2 \quad \wedge \quad \texttt{store-ts}_1 \leq \texttt{store-ts}_2$$

or

$$\texttt{venv-ts}_2 \leq \texttt{venv-ts}_1 \quad \wedge \quad \texttt{store-ts}_2 \leq \texttt{store-ts}_1$$

(*i.e.*, either $cs_1 \sqsubseteq cs_2$ or $cs_2 \sqsubseteq cs_1$). This is because the tuples are generated sequentially, and the time-stamp pairs are thus always ordered. So, for a given `call` and `benv` value, we only need store the most recent time stamps for their corresponding `venv-ts` and `store-ts` values in the memo table. This will still implement the aggressive cutoff of section 5.2, but with still lower storage overhead for the memo table `S`.

If we put all of this together, we get the following approximate algorithm:

```
Cts(call,benv)
  let <venv-memo,store-memo> = fetch-memo(call,benv)
  in
    if venv-ts ≤ venv-memo and store-ts ≤ store-memo then
        return;
    memoise(call,benv,venv-ts,store-ts);
    let ⟦c:(f . args)⟧ = call,    /* Destructure call form. */
        F = Ats(f,benv),           /* Eval proc subexp. */
        argv = mapAts(args,benv)   /* Ats(a,benv)
    in                                for a in args. */
      ans := ans ⊔ [<c,benv> ↦ F];
      /* Argument c to Fts is the 1CFA contour: */
      for each f in F do Fts(f, argv, c);


PRts(lam)
  S := [];
  venv  := []; venv-ts  := 0;
  store := []; store-ts := 0;
  ans := [];

  Fts(Ats(lam,[]),<{stop}>,topcall);
  return(S)


fetch-memo(call,benv)
  if <call,benv> in table S then
      return(table-entry(S,<call,benv>));
  /* Never seen <call,benv> before. Initialise memo entry. */
  memoise(call,benv,-1,-1);
  return(<-1,-1>);


memoise(call,benv,v-ts,s-ts)
  set-table-entry(S, <call,benv>, <v-ts,s-ts>);
```

This algorithm is the one that I have actually implemented. All of the applications I discuss in the following chapters have been built on top of this implementation. The running time of the package, when applied to small examples, is effectively immediate (about a second, in the interpreter), and the loss of precision has not been noticeable for the examples I have tried.

5.4 Particulars of My Implementation

I have an implementation of 1CFA written in the Scheme dialect T [Rees$^+$ 82, Rees$^+$ 84]. It uses a modified version of the ORBIT compiler's front end to produce CPS Scheme code from programs written in full Scheme. The features of the particular algorithm are:

- 1CFA (section 3.7)
- simple side-effects analysis (section 3.8.1)
- external procedures and calls (section 3.8.2)
- user procedures and continuations partitioned (section 3.8.3)
- single-threaded variable environment and store with time stamps (section 5.3).

The front end and its related utilities are about 2100 lines of commented code. The actual 1CFA code is about 450 lines of commented code.

Appendix A is a code listing of my 1CFA implementation. Appendix B gives a detailed dump of the call cache produced by applying this code to two small Scheme procedures, iterative factorial and the small "puzzle" example from chapter 9 (page 120).

On a DECstation 3100, running uncompiled in the T interpreter, the 1CFA analysis can analyse an iterative factorial procedure in .58 seconds, and a `delq` procedure in 1.8 seconds. I have not made a serious effort to carefully evaluate the running time of the analysis; these running times are only given to suggest that the execution times are manageable. A detailed study of compile-time costs would be premature at this point. The efficiency of the simple implementation given in appendix A could be improved a great deal by tuning the code and data structures for speed. These issues are discussed in greater detail in chapter 11.

Chapter 6

Implementation II

You wouldn't want us to give you a Ph.D.
for a buggy optimiser, would you?
— J. Reynolds

Do the algorithms of the previous chapter correctly implement the abstract control-flow analysis semantics functions? In this chapter, we'll develop proofs to show:

- The basic algorithm computes the abstract CFA function.
- The aggressive-cutoff algorithm is equivalent to the basic algorithm.
- The time-stamp algorithm safely approximates the basic algorithm.

This will formally establish the correctness of our algorithms.

6.1 Preliminaries

First, we need to briefly summarise the framework of section 4.4.3. The functions $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ have the same, simple structure. When one of these functions is applied to an argument, it makes a local contribution to the final answer, and then recurses on some set of child arguments. If the local contribution for argument x is $g\ x$, and the set of children to recurse on is $R\ x$, then we have the general form[1]

$$f\ x = g\ x \sqcup \bigsqcup f\ (R\ x). \tag{6.1}$$

We assume these functions map a finite set X to a lattice A, *i.e.* $f\colon X \to A$. (In the case of $\widehat{\mathcal{C}}$, for instance, X is $\widehat{\text{CState}}$ and A is $\widehat{\text{CCache}}$.) The functionalities of the parameterising functions are $g\colon X \to A$ and $R\colon X \to \mathcal{P}(X)$.

[1]Actually, $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ are *mutually* recursive — one calls the other. See section 4.4.3 for further discussion.

It simplifies our descriptions of these functions to introduce their power-set relatives:

$$\underline{g}\,Y = \bigsqcup g\,Y, \qquad \underline{R}\,Y = \bigcup R\,Y,$$

with functionality $\underline{g}\colon \mathcal{P}(X) \to A$, and $\underline{R}\colon \mathcal{P}(X) \to \mathcal{P}(X)$. The power-set analog of equation 6.1 is

$$\underline{f}\,Y = \underline{g}\,Y \sqcup \underline{f}\,(\underline{R}\,Y), \tag{6.2}$$

where $\underline{f}\colon \mathcal{P}(X) \to A$. We saw in section 4.4.3 that the least solution of equation 6.2 is

$$\underline{f}\,Y = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,Y),$$

and the least solution of equation 6.1 is

$$f\,x = \underline{f}\,\{x\} = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,\{x\}).$$

6.2 The Basic Algorithm

We need to show that the basic algorithm of section 5.1 computes

$$f\,x = \bigsqcup_{i \geq 0} \underline{g}\,(\underline{R}^i\,\{x\}).$$

For the remainder of this chapter, we will concentrate on computing the set $\bigcup_i \underline{R}^i\,\{x\}$ instead of its image under $\underline{g}$. This will simplify the discussion. Let $\underline{h}$ be the function computing $\underline{R}$'s closure on Y:

$$\underline{h}\,Y = \bigcup_{i \geq 0} \underline{R}^i\,Y.$$

It's clear that $\underline{f}\,x = \underline{g}\,(\underline{h}\,\{x\})$.

Altering the basic algorithm of section 5.1 to simply compute the closure set S gives us

```
h(x) =
    let S    = ∅,
        loop(y) = if y ∉ S then
                      S := S ∪ {y};
                      for each z ∈ R(y)
                          loop(z);
    in loop(x); S
```

We would like to show that h(x) correctly computes the set $\underline{h}\,x$.

First, it's clear that h(x) terminates. Each recursive call either terminates immediately (if y ∈ S), or increases the size of S, making progress towards termination.

We can prove that h computes the desired set by considering the associated loop function L. Suppose we call the interior procedure loop on value y', with S initially having the value S'. Define $L\,S'\,y'$ to be the value of S after loop(y') returns.

The procedure `loop(y)` recursively explores the subtree rooted at `y` in a depth-first order, where R can be viewed as the "sons-of" function. However, `loop` cuts off the search when it arrives at a node in the set `S`. So, we can intuitively see that $L\ S\ y$ returns the tree explored from y, *stopping at elements in* S. The $\underline{h}$ function, on the other hand, can be viewed as a complete breadth-first search from its argument. This leads us to the following theorem:

Theorem 13 $L\ S\ y \cup \underline{h}\ S = \underline{h}\ \{y\} \cup \underline{h}\ S.$

Proof:

1. $L\ S\ y \cup \underline{h}S \subset \underline{h}\ \{y\} \cup \underline{h}S$:
   During the execution of `loop(y)`, all elements added to `S` are produced by chains of R applications, starting with `y`, hence each such element must be in y's closure, $\underline{h}\ \{y\}$. So,
   $$L\ S\ y \subset \underline{h}\ \{y\} \cup S \subset \underline{h}\ \{y\} \cup \underline{h}\ S.$$
   The inclusion follows.

2. $\underline{h}\ \{y\} \cup \underline{h}S \subset L\ S\ y \cup \underline{h}S$:
   Proof by negation. Suppose $z \in \underline{h}\ \{y\} - (L\ S\ y \cup \underline{h}\ S)$. Now $z \in \underline{h}\ \{y\}$ means there is a sequence $s_0, \ldots, s_n$ such that $s_0 = y$, $s_n = z$, and $s_{i+1} \in R\, s_i$ (that is, there is a sequence of R applications leading from y to z). Since $z \notin \underline{h}\ S$, we know that none of the s_i are in S — if one was, then its descendant z would have to be in the closure. Since $z \notin L\ S\ y$, we know $s_{n-1} \notin L\ S\ y$. However, s_{n-1} isn't in the initial `S` set, so if it were ever added to `S`, `loop` would recurse over its children $R\ s_{n-1}$, which includes z, and so z would have been added to `S`. In a similar fashion, we can derive
   $$z \notin L\ S\ y \quad\Rightarrow\quad s_{n-1} \notin L\ S\ y \quad\Rightarrow\quad \cdots \quad\Rightarrow\quad s_1 \notin L\ S\ y \quad\Rightarrow\quad y \notin L\ S\ y.$$
   But, clearly $y \in L\ S\ y$. Contradiction.

Q.E.D.

We can finish off `h` with the following corollary:

Corollary 14 `h(x)` $= \underline{h}\ \{x\}$

Proof: This follows from the previous theorem, `h(x)` $= L\ \emptyset\ x$, and $\underline{h}\ \emptyset = \emptyset$.

6.3 The Aggressive-Cutoff Algorithm

If we similarly abstract the basic structure of the aggressive-cutoff algorithm of section 5.2, we get the procedure:

```
h'(x) =
    let S     = ∅,
        loop'(y) = if not (y ⊑ z for some z ∈ S) then
                       S := S ∪ {y};
                       for each z ∈ R(y)
                           loop'(z);
    in loop'(x); S
```

Assume that the domain of our functions, X, is a partial order with order relation $\sqsubseteq$. This order relation induces an preorder on the power set $\mathcal{P}(X)$:

$$Y \sqsubseteq Y' \quad \text{iff} \quad \forall y \in Y \; \exists y' \in Y' \; y \sqsubseteq y'.$$

That is, $Y \sqsubseteq Y'$ if every element in Y is $\sqsubseteq$ some element in Y'. (We will extend this notation to scalar elements: $y \sqsubseteq Y'$ is shorthand for $\{y\} \sqsubseteq Y'$.) We will call two sets equivalent if they are ordered in both senses:

$$Y \equiv Y' \quad \text{iff} \quad Y \sqsubseteq Y' \wedge Y' \sqsubseteq Y.$$

Let's further assume that the functions g and R are monotonic with respect to these orders. Then $\underline{g}$, $\underline{R}$, and $\underline{R}^i$ are monotonic. It follows that $f\,x = \bigsqcup_i \underline{g}\,(\underline{R}^i\;\{x\})$ is monotonic.

What do the R and g functions correspond to in the control-flow semantics? Consider the simple-call case of the $\hat{\mathcal{C}}$ function. R gives the recursion set $\left\{\left\langle f',\, \widehat{av},\, \widehat{ve},\, \hat{b}',\, \hat{\sigma}\right\rangle \mid f' \in F\right\}$ calculated from the input argument tuple, and g is the local contribution made by the function towards the answer cache: $\left[\left\langle c,\, \hat{\beta}\right\rangle \mapsto F\right]$ (figure 4.5). All of the $\hat{\mathcal{C}}$ and $\hat{\mathcal{F}}$ cases have definitions with similar structure, so R and g functions can be defined for them. It should be a simple matter to convince yourself that, given the above definitions of set ordering and monotonicity, the g and R components of $\hat{\mathcal{C}}$ and $\hat{\mathcal{F}}$ are monotone. Passing $\hat{\mathcal{C}}$ more approximate arguments causes it both to make a larger contribution to the call cache and to recurse over a larger set of argument tuples; similarly for $\hat{\mathcal{F}}$. So our assumptions of monotonicity apply to the abstract semantic functions.

Now, let's return to the two sets `h(x)` and `h'(x)` — the former computed by the basic algorithm, and the latter computed by the aggressive algorithm. If we know that `h(x)` $\sqsubseteq$ `h'(x)`, then the monotonicity of g gives us that

$$\bigsqcup \underline{g}\; \texttt{h(x)} \sqsubseteq \bigsqcup \underline{g}\; \texttt{h'(x)}.$$

Similarly, `h'(x)` $\sqsubseteq$ `h(x)` implies the reverse ordering. So, if `h(x)` $\equiv$ `h'(x)`, then the joins of their images under g are equal:

$$\texttt{h(x)} \equiv \texttt{h'(x)} \quad \Rightarrow \quad \bigsqcup \underline{g}\; \texttt{h(x)} = \bigsqcup \underline{g}\; \texttt{h'(x)}.$$

Now, the left-hand side of the equality $\bigsqcup \underline{g}$ `h(x)` is just $f\;x$. So, if we can show the equivalence of `h` and `h'`, we can use the aggressive `h'` in place of the basic `h` to compute f.

In order to show this equivalence, we need to use the analog of the L function for `h'`. Define L' to be the corresponding function: $L'\;S'\;y'$ is the value of `S` after initialising it to S' and calling `loop'`(y').

Theorem 15 $L'\ S\ y \cup \underline{h}\ S \equiv \underline{h}\ \{y\} \cup \underline{h}\ S$.

Proof:
The proof is similar to the previous proof for the basic algorithm, using $\sqsubseteq$ in place of $\in$.

1. $L'\ S\ y \cup \underline{h}\ S \sqsubseteq \underline{h}\ \{y\} \cup \underline{h}\ S$:
   During the execution of `loop'(y)`, all elements added to `S` are of produced by chains of R applications, starting with `y`, hence each such element must be in y's closure, $\underline{h}\ \{y\}$. So,
   $$L'\ S\ y \subset \underline{h}\ \{y\} \cup S \subset \underline{h}\ \{y\} \cup \underline{h}\ S.$$
   Clearly, $\underline{h}\ S \subset \underline{h}\ \{y\} \cup \underline{h}\ S$ as well, so
   $$L'\ S\ y \cup \underline{h}\ S \subset \underline{h}\ \{y\} \cup \underline{h}\ S,$$
   which in turn implies
   $$L'\ S\ y \cup \underline{h}\ S \sqsubseteq \underline{h}\ \{y\} \cup \underline{h}\ S.$$

2. $\underline{h}\ \{y\} \cup \underline{h}\ S \sqsubseteq L'\ S\ y \cup \underline{h}\ S$:
   Proof by negation. Suppose $\exists z \in \underline{h}\ \{y\} - \underline{h}\ S$ such that $z \not\sqsubseteq L'\ S\ y \cup \underline{h}\ S$. Since $z \in \underline{h}\ \{y\}$, there is a sequence $s_0, \ldots, s_n$ such that $s_0 = y$, $s_n = z$, and $s_{i+1} \in R\ s_i$ (that is, there is a sequence of R applications leading from y to z). Now, since $z \not\sqsubseteq \underline{h}\ S$, we know that $\forall i \quad s_i \not\sqsubseteq S$ — if some s_i were $\sqsubseteq S$, then by the monotonicity of R, its descendant z would have to be $\sqsubseteq$ the closure $\underline{h}\ S$.

   Since $z \not\sqsubseteq L'\,S\,y$, we know $s_{n-1} \not\sqsubseteq L'\,S\,y$. To see this, suppose that $s_{n-1} \sqsubseteq x \in L'\,S\,y$, for some x. Since $s_{n-1} \not\sqsubseteq S$, we know that $x \notin S$, so x must have been added to $L'\,S\,y$ during the execution of the algorithm. However, when `loop'` adds some element x to `S`, it also recurses over x's children $R\ x$. R is monotonic, so the recursion set $R\ x$ includes some element $w \sqsupseteq z$. Since `loop'` is called on w, the final value is guaranteed to be $\sqsupseteq w$ (either w is $\sqsubseteq$ some element already in `S`, or it is added to `S` itself; either way, the result set is $\sqsupseteq w$), hence $\sqsupseteq z$. So $z \sqsubseteq L'\ S\ y$, which is a contradiction. Hence $s_{n-1} \not\sqsubseteq L'\ S\ y$.

   We can apply this reasoning backwards along the s_i chain, giving us the series of implications:
   $$z \not\sqsubseteq L'\,S\,y \quad\Rightarrow\quad s_{n-1} \not\sqsubseteq L'\,S\,y \quad\Rightarrow\quad \cdots \quad\Rightarrow\quad s_1 \not\sqsubseteq L'\,S\,y \quad\Rightarrow\quad y \not\sqsubseteq L'\,S\,y.$$
   But, clearly $y \sqsubseteq L'\ S\ y$. Contradiction.

Q.E.D.

This theorem allows us to show that the aggressive-cutoff algorithm is equivalent to the basic algorithm, with the following corollary:

Corollary 16 $\forall y,$ `h'`$(y) \equiv \underline{h}\ \{y\}$.

Proof: The previous theorem gives us

$$L'\,\emptyset\,y \cup \underline{h}\,\emptyset \equiv \underline{h}\,\{y\} \cup \underline{h}\,\emptyset.$$

$\underline{h}\,\emptyset = \emptyset$, and `h'(y)` $= L'\,\emptyset\,y$, and the equivalence follows. Q.E.D.

Now that we've established that the basic and aggressive-cutoff algorithms are equivalent, we can implement the more efficient aggressive-cutoff algorithm and know that it will compute the desired function.

6.4 The Time-Stamp Algorithm

It is fairly simple to show that the time-stamp algorithm is a conservative approximation to the basic algorithm.

Suppose that the $\widehat{\mathcal{C}}$ function is being applied to some simple-call tuple cs. Consider the recursion set $R\ cs$ of FState tuples on which $\widehat{\mathcal{C}}$ recursively applies $\widehat{\mathcal{F}}$. This is the set

$$\left\{\left\langle \hat{f}_i,\ \widehat{av},\ \widehat{ve},\ \widehat{nb}\ \hat{b}\,c,\ \hat{\sigma}\right\rangle \mid 1 \le i \le n\right\},$$

where $\left\{\hat{f}_1, \ldots, \hat{f}_n\right\} = \widehat{\mathcal{A}}\ f\ \hat{\beta}\ \widehat{ve}$. Changing to an algorithm that uses a single-threaded implementation of the variable environment and store means that between the $\hat{f}_1$ call and the $\hat{f}_2$ call, the store and variable environment could increase. So the actual values of the global variable environment and store present when the $\hat{f}_2$ call is made aren't $\widehat{ve}$ and $\hat{\sigma}$, but some values $\widehat{ve}_2 \sqsupseteq \widehat{ve}$ and $\hat{\sigma}_2 \sqsupseteq \hat{\sigma}$. This means we are actually calling $\mathcal{F}$ with the tuple $\left\langle \hat{f}_2,\ \widehat{av},\ \widehat{ve}_2,\ \widehat{nb}\ \hat{b}\,c,\ \hat{\sigma}_2\right\rangle \sqsupseteq \left\langle \hat{f}_2,\ \widehat{av},\ \widehat{ve},\ \widehat{nb}\ \hat{b}\,c,\ \hat{\sigma}\right\rangle$. Between the $\hat{f}_2$ call and the $\hat{f}_3$ call, the globally modified $\widehat{ve}$ and $\hat{\sigma}$ variables can again be increased. So the actual values of the variable environment and store present when the $\hat{f}_3$ is made are some values $\widehat{ve}_3 \sqsupseteq \widehat{ve}_2$ and $\hat{\sigma}_3 \sqsupseteq \hat{\sigma}_2$, and so forth for the successive calls to the rest of the $\hat{f}_i$. So the effective recursion set for the single-threaded implementation is

$$\left\{\left\langle \hat{f}_i,\ \widehat{av},\ \widehat{ve}_i,\ \widehat{nb}\ \hat{b}\,c,\ \hat{\sigma}_i\right\rangle \mid 1 \le i \le n\right\}.$$

Call this alternate recursion set $R'\ cs$. It's clear that the $\widehat{ve}_i$ are all $\sqsupseteq \widehat{ve}$ and that the $\hat{\sigma}_i$ are all $\sqsupseteq \hat{\sigma}$, so each tuple in $R'\ cs$ is $\sqsupseteq$ its counterpart in $R\ cs$. Thus we know $R' \sqsupseteq R$ using the ordering of the previous section.

This is pervasively true throughout the $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{F}}$ functions: the only change of making the variable environment and store be global, single-threaded data structures is to shift to recursion sets that are $\sqsupseteq$ their counterparts in the basic algorithm. So, returning to our simple $f\ x = g\ x \sqcup \bigsqcup f\ (R\ x)$ model of the semantics functions, the single-threaded algorithm computes the function defined by

$$f'\ x = g\ x \sqcup \bigsqcup f'\ (R'\ x),$$

which is just

$$f'\ x = \bigsqcup_i \underline{g}\ (R'^i\ x).$$

Now, the basic recursion is computing the function

$$f\ x = \bigsqcup_i \underline{g}\,(R^i\ x).$$

As we discussed in the previous section, if $R' \sqsupseteq R$, then $R'^i \sqsupseteq R^i$. This plus the monotonicity of $\underline{g}$ gives us that

$$f' \sqsupseteq f,$$

or that the single-threaded recursion safely approximates the basic recursion.

Chapter 7

Applications I

The term "collection of hacks" is not meant to be pejorative. Some of my best friends are collections of hacks.
— S. Fahlman

Once we've performed control-flow analysis on a program, we can use the information gained to perform other data-flow analyses. Control-flow analysis by itself does not provide enough information to solve all the data-flow problems we might desire (we'll discuss these limitations in the following chapter), but we can still solve some useful problems.

In this chapter, we'll consider three applications we can perform using control-flow information: induction-variable elimination, useless-variable elimination, and constant propagation.

7.1 Induction-Variable Elimination

Induction-variable elimination is a technique used to transform array references within loops into pointer incrementing operations. For example, suppose we have the following C routine:

```
int a[50][30];

example() {
    integer r, c;
    for(r=0; r<50; r++)
        for(c=0; c<30; c++)
            a[r][c] = 4;
    }
```

The `example` function assigns 4 to all the elements of array `a`. The array reference `a[r][c]` on a byte addressed machine is equivalent to the address computation `a+4*(c+30*r)`. This calculation requires two multiplications and two additions. However, since the address value is linear in the `c` and `r` terms, we can cheaply compute the address computation incrementally.

```
example() {
    integer *ptr;
    for(ptr=a; ptr<a+1500; ptr++)
        *ptr = 4;
    }
```

The `ptr` variable tracks the dependent value `a+4*(c+30*r)`. Induction-variable elimination (IVE) is a technique for automatically performing the above transformation. Classical IVE is treated in detail in the "Dragon" book [Aho+ 86].

Classical IVE for imperative languages such as C or Pascal begins by searching for *basic induction variables* (BIVs). A BIV is a variable that is only altered in a loop by incrementing or decrementing it by constants. The `r` and `c` variables are BIVs in the above example. In Scheme, the role of a single C BIV is played by a *set* of related Scheme variables, called a *basic induction variable family*. We use a set of Scheme variables in the place of a single C variable because of the different ways that loop variables are updated in the two languages. In C, a loop variable is updated multiple times within a loop by assigning to that single variable:

```
j=0;
while(test) {
    j = j+1;
    ...
    j = j-4;
    ...
    j = j+2;
    }
```

In Scheme or ML, an iteration variable is updated by rebinding it in a tail-recursive loop:

```
(letrec ((loop (λ (j)
                 (if (not test)
                   (let ((j' (+ j 1)))
                     ...
                     (let ((j% (- j' 4)))
                       ...
                       (loop (+ j% 2))))))))
  (loop 0))
```

In the Scheme code, the variables {`j`, `j'`, `j%`} correspond to the C variable `j`.[1] So, whereas IVE analysis in C searches for the basic induction variable `j`, the Scheme variant searches for the family {`j`, `j'`, `j%`}.

A *basic induction variable* (BIV) family is a set of variables $B = \{v_1, \ldots, v_n\}$ obeying the following constraints:

1. Every call site that calls one of the v_i's lambda expression may only call one lambda expression.

2. Each v_i may only have definitions of the form b (constant b), and $a + v_j$ ($v_j \in B$, constant a). This corresponds to stating that v_i's lambda must be one of:

   - Called from a call site that simply binds v_i to a constant:
     `((λ (x vi z) ...) foo 3 bar)`.
   - The continuation of a `+` call, with v_j and a constant for addends:
     `(+ vj a (λ (vi) ...))`.
   - The continuation of a `-` call, with v_j and a constant for subtrahend and minuend:
     `(- vj a (λ (vi) ...))`.
   - Called from a call site that simply binds v_i to some v_j, *e.g.*:
     `((λ (x vi z) ...) foo vj bar)` (This is a special case of $a = 0$).

Given the control-flow information summarised by an abstract call cache function $\hat{\gamma}$, it is fairly simple to compute sets of variables satisfying these constraints.

After finding BIVs, IVE analysis searches for *dependent induction variables* (DIVs) — variables whose values are affine functions of some other induction variable. The computations calculating these variables are then replaced with cheaper incremental updates. Again, the Scheme variant looks for families of variables dependent on BIV families, and the emphasis is on binding updates instead of assignment updates.

A *dependent induction variable* (DIV) family is a set of variables $D = \{w_i\}$ together with a function $f(n) = a + b * n$ (a,b constant) and a BIV family $B = \{v_i\}$ such that:

- For each w_i, every definition of w_i is $w_i = f(v_j)$ for some $v_j \in B$.

We call the value $f(v_i)$ the *dependent value.*

We can introduce three sets of new variables, $\{z_i\}$, $\{x_i\}$, and $\{y_i\}$. The z_i and x_i track the dependent value, and the y_i are introduced as temporaries to help update the basic variable. In the following description, we use for illustration the following piece of Scheme code:

[1] In fact, all three variables in the set would probably be named `j` by the programmer — there is no point in distinguishing the names if they represent the same conceptual identifier at different points in the code. So the binding operations would look more like a real update: `(let ((j (- j 4))) ...)`. However, by the time the compiler begins to analyse the code, the compiler front-end has "alphatised" the three variables to have distinct names.

```
(+ v1 4 (λ (v2) ...(* 3 v2 (λ (w1) ...))))
```

BIVs `v1` & `v2`

DIV `w1` defined to be 3 $*$ `v2`

with associated function $f(n) = 3 * n$.

We perform the following transformations on the code:

- **Introduce z_i into the v_i lambda expression:**
  Modify each lambda that binds one of the v_i to simultaneously bind the corresponding value $f(v_i)$ to z_i.
  Binding site `(λ (v2) ...)` becomes:
  `(λ (v2 z2) ...)`

- **Introduce code to compute $z_j = f(v_j)$ from $z_i = f(v_i)$ when v_j is computed from v_i**
  All call sites who have continuations binding one of the v_i (*i.e.*, call sites that step the BIV family) are modified to update the corresponding z_i value. Temporary variables x_i and y_i are used to serially compute the new values.
  Call site `(+ v1 4 k)` becomes:

```
(+ z1 12 (λ (x2)
           (+ v1 4 (λ (y2)
                     (k y2 x2)))))
```

- **Replace computation of DIV w_i from v_j with simple binding:**
  Since $z_j = f(v_j)$, we can remove the computation of w_j from v_j, and simply bind w_j to z_j. Dependent variable computation `(* v2 3 (λ (w1) ...))` becomes:
  `((λ (w1) ...) z2)` and z_2 can be β-substituted for w_1.

Notice that $\{w_i\} \cup \{z_i\} \cup \{x_i\}$ now form a new BIV family, and may trigger off new applications.

For example, consider the loop of figure 7.1(a). It is converted to the partial CPS of (b). Now, {`n`} is a BIV family, and {`m`} is a DIV family dependent on {`n`}, with $f(n) = 3n$. A wave of IVE transformation gives us the optimised result of (c). Analysis of (c) reveals that {`3n`, `3n'`, `3n%`} is a BIV family, and {`p`} is a DIV family, with $f(n) = 4 + n$. Another wave of IVE gives us (d). If we examine (d), we notice that `3n`, `3n'`, and `3n%` are never used, except to compute values for each other. Another analysis using control-flow information, useless-variable elimination, spots these useless variables; they can be removed, leaving the optimised result of (e). (We'll discuss useless-variable elimination in the next section.)

This example of figure 7.1 was produced by an IVE implementation that I wrote in T using the results produced by a 0CFA analysis.

```
(letrec ((f (λ (n) (if (< n 50)
                       (if (= (aref a n) n) (f (+ n 1))
                           (block (print (+ 4 (* 3 n)))
                                  (f (+ n 2))))))))
  (f 0))
```

(a) Before

```
(letrec ((f (λ (n)
              (if (< n 50)
                  (if (= (aref a n) n) (+ n 1 f)
                      (* 3 n (λ (m) (+ 4 m (λ (p) (print p)
                                                (+ n 2 f))))))))))
  (f 0))
```

(b) Partial CPS conversion

```
(letrec ((f (λ (n 3n)
              (if (< n 50)
                  (if (= (aref a n) n)
                      (+ 3n 3 (λ (3n')
                                (+ n 1 (λ (n') (f n' 3n')))))
                      (+ 4 3n (λ (p) (print p)
                                     (+ 3n 6 (λ (3n%)
                                               (+ n 2 (λ (n%)
                                                        (f n% 3n%)))))))))))
  (f 0 0))
```

(c) IVE wave 1

```
(letrec ((f (λ (n 3n 3n+4)
  (if (< n 50)
      (if (= (aref a n) n)
          (+ 3n+4 3 (λ (3n+4')
                      (+ 3n 3 (λ (3n') (+ n 1 (λ (n')
                                                (f n' 3n' 3n+4')))))))
          (block (print 3n+4)
                 (+ 3n+4 6 (λ (3n+4%)
                             (+ 3n 6 (λ (3n%)
                                       (+ n 2 (λ (n%)
                                                (f n% 3n% 3n+4%)))))))))))))
  (f 0 0 4))
```

(d) IVE wave 2

```
(letrec ((f (λ (n 3n+4)
              (if (< n 50)
                  (if (= (aref a n) n)
                      (+ 3n+4 3 (λ (3n+4') (+ n 1 (λ (n') (f n' 3n+4')))))
                      (block (print 3n+4)
                             (+ 3n+4 6 (λ (3n+4%)
                                         (+ n 2 (λ (n%)
                                                  (f n% 3n+4%))))))))))
  (f 0 4))
```

(e) After

Figure 7.1: Example IVE application

7.2 Useless-Variable Elimination

A *useless variable* is one whose value contributes nothing to the final outcome of the computation. We can remove a useless variable and the computation producing its value from our program. For example, consider the following code fragment:

```
(let ((sum  (+ a b))                  (let ((sum (+ a b)))
      (prod (* a b)))        ⇒          (f sum a))
  (f sum a))
```

Prod is never used in the program, so we can remove it and its binding computation (* a b). This example is fairly easy to detect; most Scheme compilers would find and optimise it.

On the other hand, some useless variables involve circular dependencies or multiple (join) dependencies. The simple lexical analysis that suffices for the above example won't spot these cases. For example, consider the factorial loop:

```
(letrec ((lp (λ (ans j bogus)
               (if (= 0 j) ans
                   (lp (* ans j)
                       (- j 1)
                       (sqrt bogus))))))
  (lp 1 n n))
```

Although the variable bogus doesn't contribute anything at all to the final result, it appears to be used in the loop. Useless-variable elimination is used to spot these cases.

UVE is often useful to clean up after applying other code transformations, such as copy propagation or induction-variable elimination. For example, when we introduce a new variable to track an induction function on some basic induction variable, the basic variable frequently becomes useless. (For an example produced by a working UVE implementation, compare parts (d) and (e) of figure 7.1.)

7.2.1 Finding Useless Variables

Detecting these cases requires a simple backwards flow analysis on the CPS intermediate form. We actually compute a conservative approximation to the inverse problem — finding the set of all useful variables. We start with variables that must be assumed useful (*e.g.*, a variable whose value is returned as the value of the procedure being analysed). Then we trace backwards through the control-flow structure of the program. If a variable's value contributes to the computation of a useful variable, then it, too, is marked useful. When we're done, all unmarked variables are useless. This gives us a mark-and-sweep algorithm for a sort of "computational gc."

To be specific, in the implementation of UVE that I have implemented, a variable is *useful* if it appears

- in the function position of a call:
  `(f 5 0)`
- as the predicate in a conditional primop:
  `(if p (λ () ...) (λ () ...))`
- as the continuation of a primop:
  `(+ 3 5 c)`
- as an argument in a call to
  - a side-effecting operation (output or store):
    `(print a)`, `(set-car! x y)`
  - an external procedure, or a primop whose continuation is an external procedure.
  - a primop whose continuation binds a useful variable:
    `(+ metoo 3 (λ (used) ...))`
  - a lambda whose corresponding parameter is useful:
    `((λ (x used y) ...) 3 metoo 7)`

The first three conditions spot variables used for control flow. The next two mark variables whose value escapes, and must therefore be assumed useful. The final two recursive conditions are the ones that cause the analysis to chain backwards through the control-flow graph: if a variable is useful, then all the variables used to compute its value are useful.

7.2.2 Optimising Useless Variables

Once we have found the useless variables in a program, we can optimise the program in two steps. In the first step, we remove all references to useless variables and eliminate all useless computations (primop calls). In the second step, we remove useless variables from their lambda lists where possible.

Removing useless variables from calls

In this phase, we globally remove all references to useless variables in our program. If a useless variable appears as an argument to a primop call, we remove the entire primop computation. For instance, suppose the useless variable `x` appears as an argument in the primop call `(+ x y k)`. By the definition of a useless variable, we know that the continuation `k` must bind the result of the addition to a useless variable, as well (if it didn't, we'd have marked `x` as useful). This renders the addition operation useless, so we can remove it, replacing `(+ x y k)` with `(k #f)`. The actual value passed to the continuation (we used `#f` here) is not important — remember that we are globally deleting all references to useless variables. Since the continuation `k` must bind the value `#f` to a useless variable, the value is guaranteed never to be referenced.

If a reference to a useless variable appears in a non-primop call, we simply replace it with some constant. If `x` is useless, we convert `(f x y)` to `(f #f y)`. Similar reasoning applies in this case: if `x` is useless, it must be the case that `f`'s corresponding parameter is useless as well. All references to this parameter will be deleted, so we can pass any value we like to it.

Removing useless variables from lambda lists

Suppose we have determined that `x` is useless in lambda ℓ = `(`λ `(x y) ...)`. After applying the transformation of the previous subsection, we can be sure that ℓ contains no references to `x`. The only remaining appearance of `x` is in ℓ's parameter list. Consider the places this lambda is called from, *e.g.*, `(f a 7)`. We can delete both the formal parameter `x` from its lambda list and the corresponding argument `a` from its call:

`(`λ `(x y) ...)` $\Rightarrow$ `(`λ `(y) ...)` and `(f a 7)` $\Rightarrow$ `(f 7)`.

However, we can't apply this optimisation in all cases. Suppose ℓ is called from two places, the external call and `(f a 7)`. We can't simply delete `x` from ℓ's parameter list, because the external call, which we have no control over, is going to pass a value to ℓ for `x`. Or, suppose that our lambda is only called from one place, `(f a 7)`, but that call site calls two possible lambdas. Again, we can't delete the argument `a` from the call, because the other procedure is expecting it (unless it, as well, binds a useless variable).

We have a circular set of dependencies determining when it is safe to remove a useless variable from its lambda list and its corresponding arguments from the lambda's call sites:

- We can delete a variable from a lambda list only if we can delete its corresponding argument in all the calls that could branch to that lambda.
- We can delete an argument from a call only if we can delete the corresponding formal parameter from every lambda we could branch to from that call.

It is not hard to compute a maximal solution to these constraints given control-flow information. We use a simple algorithm that iterates to a fixed point. We compute two sets: the set RV of removable useless variables, and the set RA of removable call arguments. Initialise RV to be the set of all useless variables. For each useless variable v , find all the call sites that could branch to v's lambda, and put the call's corresponding argument into RA. Then iterate over these sets until we converge:

```
ITERATE until no change
    FOR each argument a in RA
        IF a's call could branch to a lambda whose
           corresponding parameter is not in RV
        THEN remove a from RA
    FOR each variable v in RV
        IF v's lambda can be called from a call site whose
           corresponding argument is not in RA
        THEN remove v from RV
```

For the purposes of the first loop, the external lambda counts as a disqualifying branch target; for the purposes of the second loop, the external call counts as a disqualifying branch source. When we're done, we're left with RV and RA sets that satisfy the circular criteria above. We can safely remove all the arguments in RA from their calls and all the variables in RV from their lambda lists.

Results

Applying these two phases of optimisation will eliminate useless variables and their associated computations wherever possible. For example, we can remove the useless `bogus` variable and its square-root calculation from the factorial loop we considered at the beginning of this section, leaving only the necessary computations:

```
(letrec ((lp (λ (ans j)
               (if (= 0 j) ans
                   (lp (* ans j)
                       (- j 1))))))
  (lp 1 n))
```

Note that as a special case of UVE, all unreferenced variables in a program are useless. The second phase of the UVE optimisation will remove these variables when possible.

7.3 Constant Propagation

Constant propagation is another standard code improvement based on data-flow analysis. In brief, if the only def of `x` that reaches `sin(x)` is `x:=3.14`, then we can replace `sin(x)` with `sin(3.14)` (and, subsequently, `-1.0`).

In the Scheme and ML world, where we bind variables instead of assigning them, this corresponds to finding variables that are always bound to constants, and replacing their references with the constant. β-substitution spots the easy cases, *e.g.*:

```
(let ((x 3.14))
  (sin x))          ⇒     (sin 3.14)
```

However, just as for useless-variable elimination, we need something more powerful to spot circular and join dependencies.

A simple variant of control-flow analysis will handle constant propagation. Simple control-flow analysis uses an abstract environment that binds variables to sets of abstract procedures. We just augment the variable environment (and the store) to include constants as well as procedures. When we perform a call with a constant argument a, we just pass the singleton set $\{a\}$ as the corresponding value set. As the abstract interpretation runs forward, it builds up a table of which closures and constants are bound to which variables. When the analysis converges, we look up the data that was bound to a given variable v. If there's only one such datum, and it's a constant, not a procedure, we can go ahead and replace all references to v with the constant.

Chapter 8

The Environment Problem and Reflow Analysis

In this chapter, we'll examine the limits of simple control-flow analysis. Then we'll develop an extension, called reflow analysis, to move beyond these limits. In the following two chapters, we'll look at some optimisations that can be performed with reflow analysis.

The work discussed in these next three chapters is not supported by the kind of careful formal development and proofs of correctness that I've shown for the simple control-flow analysis in the previous chapters. This work must be taken simply as interesting ideas that have not yet been backed up with rigorous mathematics. With reflow analysis, we've arrived at my research frontier as of the writing of this dissertation.

Future work could include a better theoretical characterisation of reflow analysis, and proofs of conservative abstraction and computability for reflow-based optimisations. I suspect that this sort of work would be carried out in basically the same fashion as the work on simple control-flow analysis in chapter 4.

8.1 Limits of Simple Control-Flow Analysis

At the start of this dissertation, I told the following story: Data-flow analyses require a control-flow graph. Scheme programs don't have static control-flow graphs. But, if we could somehow recover a good approximation of the control-flow graph, we could use it to perform all the classical data-flow analyses.

It turns out that this is somewhat optimistic. While control-flow analysis does enable some classical data-flow analyses — such as the ones we just developed in the last chapter — another class of applications cannot be handled this way.

This is the class of analyses that require detailed environment information. Classical data-flow analyses assume a single, flat environment over the program text. Assignments and references to global variables are the conduits through which computed values flow through the program. Thus any two references to the same variable refer to the same binding of that variable. This is the model provided by assembly code.

The intermediate representation we are using, CPS applicative-order λ-calculus, is different. Here, we can have simultaneous multiple bindings of a single variable. For instance, consider the following puzzle:

```
(let ((f (λ (x h) (if (zero? x) (h) (λ () x)))))
  (f 0 (f 3 '#f)))
```

The first call to `f`, `(f 3 '#f)`, returns a closure over `(λ () x)` in environment $[\llbracket \mathtt{x} \rrbracket \mapsto 3]$. This closure is then used as an argument in the second, outer call to `f`, where it is bound to `h`. Suppose we naively picked up information from the conditional `(zero? x)` test, and flowed the information $\mathtt{x} = \mathtt{0}$ and $\mathtt{x} \neq \mathtt{0}$ down the two arms of the `if` to be picked up by references to `x`. This would work in the single flat environment model of assembler. In this Scheme puzzle, however, we are undone by the power of lambda. Suppose that our contour abstraction maps the contours allocated by the two calls to `f` to the same abstract contour. After ensuring that $\mathtt{x} = \mathtt{0}$ in the then arm of the `if`, we jump off to $\mathtt{h} =$ `(λ () x)`, and evaluate a reference to `x`, which inconveniently persists in evaluating to 3, not 0. The one variable has two simultaneous bindings. We tested one binding and then referenced the other. Confusing the two bindings of the same variable can lead us to draw incorrect conclusions.

This problem prevents us from using plain control-flow analysis to perform important flow analysis optimisations for Scheme, such as type recovery. In type recovery, we would like to perform deduction of the form:

```
;;; references to x in f are int refs
(if (integer? x) (f) (g))
```

Unfortunately, as demonstrated above, we can't make such inferences.

The problem arises because of lambda's ability to create multiple environments over a given context. Analyses such as type recovery depend upon keeping these environments distinct; computable procedural approximations depend upon folding these environments together. This basic conundrum is what I call the "environment problem."

Only certain data-flow analyses are affected by the environment problem. The key property determining this is the direction in which the iterative analysis moves through the approximation lattice. In control-flow analysis or useless-variable elimination the analysis starts with an overly precise value and incrementally weakens it until it converges; all motion in the approximate lattice is in the direction of more approximate values. So, identifying two contours together simply causes further approximation, which is safe.

With an analysis like type recovery, however, our analysis moves in both directions along the type lattice as the analysis proceeds, and this is the source of the environment problem. In type recovery, we are passing around statements about the types of the values bound to specific variables. When two different calls to a procedure, each passing arguments of different types, bind a variable in the same abstract contour, the types are joined together — moving through the type lattice in the direction of increasing approximation. However, passing through a conditional test causes type information to be narrowed down — moving in the direction of increasing precision. Unfortunately, while it is legitimate to narrow down a variable's type in a single exact contour, it is not legitimate to narrow down its type in the corresponding abstract contour — other bindings that are identified together in the same abstract contour are not constrained by the type test. This is the heart of the problem with the above puzzle.

In general, then, the simple abstraction techniques of chapters 3–7 yield correct, conservative, safe analyses only when the analysis moves through its answer lattice solely in the direction of increasing approximation. Moving beyond this requires something new.

8.2 Reflow Analysis and Temporal Distinctions

Our central problem is that we are identifying together different contours over the same variable. We are forced to this measure by our desire to reduce an infinite number of contours down to a finite set.

For example, suppose we know that abstract procedure $f = \left\langle [\![\ell\text{:}(\lambda\ \texttt{(x)}\ \ldots)]\!],\ \widehat{\beta}\right\rangle$ is called from 3 different call sites, c_1, c_2 and c_3. In 1CFA, all the contours allocated when entering ℓ will be folded down to 3 abstract contours, one of $\{c_1, c_2, c_3\}$. Now, suppose we are in the middle of an abstract interpretation, and we have a call to f from c_2:(g z). We desire to track information associated with this binding of x. First, notice that the contour abstraction already distinguishes this binding of x from all the bindings that abstract to contour c_1 or c_3. We get into trouble from two sources: (1) previous bindings of x in the same abstract contour c_2, and (2) future bindings of x in the same abstract contour c_2. In other words, we won't confuse this binding with any bindings made in abstract contours c_1 or c_3, but we could confuse it with any made by other calls from c_2 — either calls previously made during program execution, or calls to be made in the future.

So, if we want to track the information associated with a general abstract binding of a variable, we need a way to distinguish it from past and future bindings made under the same abstract contour. This is the key to an iterative technique called *reflow analysis.*

We will start the program up in the middle of its execution, running it forward from the c_2 call to ℓ. We'll call this re-interpretation of the program from some mid-point a single "reflow." On this particular entry to ℓ, we'll allocate a special contour, c_2', that will distinguish the c_2 contour being tracked from the other c_2 calls, past or future, that would allocate the same abstract contour. It's the c_2' binding of x that we will track.

Now, it is quite likely that there is already a $\langle [\![\texttt{x}]\!],\ c_2\rangle$ binding in $\widehat{ve}$ when we start the program up in mid-stride, but this abstract binding is distinct from the special c_2' binding we are uniquely allocating at the beginning of this reflow. Further, any future entries to ℓ from c_2 during this reflow will allocate the ordinary c_2 contour, merging them in with the previously-recorded c_2 binding, but *remaining distinct* from the special c_2' binding we are tracking during this reflow. In this manner, we are distinguishing the reflow's initial binding from other bindings, past and future, that would otherwise abstract to the same binding.

This allows us to track the special c_2' contour's variable bindings during the program reflow. Suppose we are running the program forward, and we come to a reference to x, bound in contour c'_2. We know this isn't a binding of `x` established by any call to ℓ from either c_1 or c_3, else the contour would be c_1 or c_3. We also know that this isn't a binding of `x` caused by a call to ℓ from c_2 that occurred either before or after the call we're tracking, because those calls get allocated contour c_2 — the special contour c'_2 is allocated only once. So we've managed to keep this binding of `x` distinct from its other bindings.

This means that as we run the program forward, whenever we test `x`, if the test occurs in contour c'_2, we can track the information produced by the test. If the test is to some other binding of `x`, we ignore it. Whenever we arrive at a reference to `x`, if the reference occurs in contour c'_2, we can record the information. If the reference is to some other binding of `x`, we ignore it. Tracking the c'_2 binding of `x` is safe, because we've isolated it from any of its other possible bindings.

Since we've only tracked one of ℓ's possible contours, we must repeat the analysis for the two other abstract contours allocated to ℓ. So we restart the program from the call c_1, allocating special contour c'_1, and track the binding of `x` in that special contour. Finally, we restart the analysis from the c_3 call.

One detail we must handle is how to restart the analysis in mid-stream. What values do we use for $\widehat{ve}$ and $\hat{\sigma}$, for example? A simple solution is to take values for $\widehat{ve}$ and $\hat{\sigma}$ that safely approximate any value they could have had at that point of the interpretation. We could, for example, use the final values of $\widehat{ve}$ and $\hat{\sigma}$ after performing a control-flow analysis of our program using the time-stamp approximate algorithm discussed in section 5.3. These values are guaranteed to approximate any $\widehat{ve}$ or $\hat{\sigma}$ that could arise during interpretation of the program. So, we can safely use them to restart the program from any call context $\left\langle c,\ \hat{\beta}\right\rangle$ that appears in the domain of the call cache γ produced by the analysis.

Chapter 9

Applications II: Type Recovery

Strong types are for weak minds.
— anon.

In this chapter we'll examine in detail an optimisation that requires reflow analysis: type recovery. The full-blown semantics machinery for type recovery is fairly complex, and I recommend browsing or skipping these sections on a first reading.

9.1 Type Recovery

Scheme is a *latently typed* language [Rees$^+$ 86]. This means that unlike statically typed languages such as ML or Pascal, types are associated only with run-time values, not with variables. A variable can potentially be bound to values of any type, and type tests can be performed at run time to determine the execution path through a program. This gives the programmer an extremely powerful form of polymorphism without requiring a verbose system of type declarations.

This power and convenience is not without substantial cost, however. Since run-time behavior can determine which values are bound to a given variable, precisely determining the type of a given reference to a variable is undecidable at compile time. For example, consider the following fragment of Scheme code:

```
(let ((x (if (oracle) 3 'three)))
  (f x))
```

The reference to `x` in the subexpression `(f x)` could have any one of the types integer, symbol, integer+symbol, or bottom, depending on whether the oracle always returns true, always returns false, sometimes returns both, or never returns at all, respectively. Of course, determining the oracle's behavior at compile time is, in general, undecidable.

We can use data-flow analysis to recover a conservative approximation of the type information implicit in a program's structure. This analysis performs a kind of type inference. Since the term "type inference" is commonly used to refer to the generation of static type assignments for variables, I will instead refer to the type analysis performed by this technique as *type recovery*: the recovery of type information from the control and environment structure of a program.

9.1.1 Sources of Type Information

Type information in Scheme programs can be statically recovered from three sources: conditional branches, applications of primitive operations, and user declarations.

Conditional Branches

Consider a simple version of the Scheme `equal?` predicate:

```
(define (equal? a b)
  (cond ((number? a)
         (and (number? b) (= a b)))
        ((pair? a)
         (and (pair? b)
              (equal? (car a) (car b))
              (equal? (cdr a) (cdr b))))
        (else (eq? a b))))
```

There are three arms in the `cond` form. References to `a` down the first arm are guaranteed to have type number. Furthermore, in the numeric comparison form, `(= a b)`, an assumption can be made that `b` is also a number, since the `and` guarantees that control reaches the comparison only if `(number? b)` is true. Similarly, we can infer that references to `a` in the second arm of the `cond` have the pair type and, less usefully, that references to `a` in the third arm of the `cond` do not have either type pair or number. It is important to realise from this example that Scheme type recovery assigns types not to variables, but to variable *references*. The references to `a` in the different arms of the `cond` all have different types.

Type recovery of this sort can be helpful to a Scheme compiler. If the compiler can determine that the `(= a b)` form is guaranteed to compare only numbers, it can compile the comparison without a run-time type check, gaining speed without sacrificing safety. Similarly, determining that the `(car a)`, `(car b)`, `(cdr a)`, and `(cdr b)` forms are all guaranteed to operate on pairs allows the compiler to safely compile the `car` and `cdr` applications without run-time type checks.

Primop Application

An obvious source of type information is the application of primops such as `+` and `cons`. Clearly the result of `(cons a b)` is a pair. In addition, we can recover information about the type of subsequent references to primop arguments. For example, after the primop application `(cdr p)`, references to `p` along the same control path can be assumed to have the pair type.

As a simple example, consider the following Scheme expression:

```
(let* ((a (cdr b))
       (q (char->integer (vector-ref a i))))
  ... (car b) ... (+ q 2))
```

References to `b` occurring after the `(cdr b)` form are guaranteed to have the pair type — otherwise, the `cdr` application would have rendered the effect of the computation to be undefined. Hence, the subsequent `(car b)` application does not need to do a type check. Similarly, after evaluating the second clause of the `let*`, references to `a` have type vector and references to `i` and `q` are small integers, because `vector-ref` requires an integer index, and `char->integer` generates an integer result. Hence the compiler can omit all type checks in the object code for `(+ q 2)`, and simply open-code the addition.

In recovering information about the arguments to primops, we are essentially using information from "hidden conditional tests" inside the primop. The semantics of a (dangerous) CPS `car` primop is:

```
(define (car p cont)
  (if (pair? p) (cont (%car p))
      ($)))
```

where the subprimitive operation `%car` is only defined over cons cells, and `($)` is a special primop denoting a computation with undefined effect. Computation proceeds through the continuation `cont` only in the then arm of the type test, so we may assume that `p`'s value is a cons cell while executing the continuation. Recovering the type information implied by `car` reduces to recovering type information from conditional branches.

Of course, the compiler does not need to emit code for a conditional test if one arm is `($)`. It can simply take the undefined effect to be whatever happens when the code compiled for the other arm is executed. This reduces the entire `car` application to the machine operation `%car`.

A safe implementation of Scheme is one that guarantees to halt the computation as soon as a type constraint is violated. This can be modeled by replacing the `($)` form in the else arm of the `car` definition with a call to the run-time error handler:

```
(define (car p cont)
  (if (pair? p) (cont (%car p))
      (error)))
```

Of course, type information recovered about the arguments to a particular application of `car` may allow the conditional test to be determined at compile time, again allowing the compiler to fold out the conditional test and error call, still preserving type-safety without the run-time overhead of the type check.

User Declarations

Type recovery can also pick up information from judiciously placed declarations inserted by the programmer. There are essentially two types of user declarations, one requesting a run-time type check to enforce the verity of the declaration, and one simply asserting a type at compile time, in effect telling the compiler, "Trust me. This quantity will always have this type. Assume it; don't check for it."

The first kind of declaration is just T's `enforce` procedure [Rees+ 82], which can be defined as:

```
(define (enforce pred val) (if (pred val) val (error)))
```

`Enforce` simply forces a run-time type check at a point the user believes might be beneficial to the compile-time analysis, halting the program if the check is violated. For example,

(block (enforce number? x) ...*forms*...)

allows the compiler to assume that references to `x` in the forms following the `enforce` have the number type. `Enforce` allows execution to proceed into the body of

(let ((y (enforce integer? (foo 3)))) *body*)

only if `(foo 3)` returns an integer. Clearly, `enforce`-based type recovery is simply conditional-branch based type recovery.

The "trust me" sort of user declaration is expressed in Scheme via the `proclaim` procedure, which asserts that its second argument has the type specified by its first argument. For example, `(proclaim symbol? y)` tells the compiler to believe that `y` has type symbol. Like `enforce`, `proclaim` can also be viewed as a kind of conditional expression:

```
(define (proclaim pred val) (if (pred val) val ($)))
```

with `($)` the "undefined effect" primop. Since an undefined effect means that "anything goes," the compiler is permitted to elide the conditional expression altogether and simply take note of the programmer's assertion that `val` has the declared type. Incorrect assertions will still result in undefined effects.

9.1.2 Type Recovery from Multiple Sources

All three sources of type information — conditional branches, primop applications, and user declarations — can be used together in recovering type information from programs, thereby enabling many optimisations. Consider the `delq` procedure of figure 9.1. Because `ans` is bound only to the constant `'()`, itself, and the result of a `cons` application, it must always be a list. So all references to `ans` are completely typeable at compile time. Because of the conditional type test `(pair? rest)`, `car` and `cdr` are guaranteed to be applied to legitimate pair values. Thus compile-time type recovery can guarantee the full type safety of `delq` with no extra run-time type checks.

For a numerical example, consider the factorial procedure in figure 9.1. Note the use of an explicit run-time type check, `(enforce integer? n)`, to force the subsequent reference to `n` to be of integer type. With the help of this user declaration, the analysis

```
(define (delq elt lis)
  (letrec ((lp (λ (ans rest)
                 (if (pair? rest)
                     (let ((head (car rest)))
                       (lp (if (eq? head elt) ans
                               (cons head ans))
                           (cdr rest)))
                     (reverse! ans)))))
    (lp '() lis)))


(define (fact n)
  (letrec ((lp (λ (ans m)
                 (if (= m 0) ans
                     (lp (* ans m) (- m 1))))))
    (enforce integer? n)
    (lp 1 n)))
```

Figure 9.1: Scheme `delq` and factorial

can determine that `m` is always bound to an integer, and therefore, that `ans` must also be bound to an integer. Thus, the factorial function written with generic arithmetic operators can have guaranteed type safety for the primop applications and also specialise the generic arithmetic operations to integer operations, at the expense of a single run-time type check per `fact` call.

If we eliminate the `enforce` form, then the type recovery can do less, because `fact` could be called on bogus, non-integer values. However, if the equality primop `(= m 0)` requires its arguments to have the same type, we can infer that references to `m` after the equality test must be integer references, and so the multiplication and subtraction operations are guaranteed to be on integer values. Hence, even in the naive, declaration-free case, type-recovery analysis is able to pick up enough information from the code to guarantee the type safety of `fact` with only a single type check per iteration, as opposed to the four type checks required in the absence of any analysis.

The implementation of the type recovery algorithm discussed in section 9.5 can, in fact, recover enough information from the `delq` and `fact` procedures of figure 9.1 to completely assign precise types to all variable references, as discussed above. For these examples, at least, compile-time type analysis can provide run-time safety with no execution penalty.

9.2 Quantity-based Analysis

Type recovery is an example of what I call a *quantity-based analysis*. Consider the Scheme expression

```
(if (< j 0) (- j) j)
```

Whatever number `j` names, we know that it is negative in the then-arm of the `if` expression, and non-negative in the else-arm. In short, we are associating an abstract property of `j`'s value (its sign) with control points in the program. It is important to realize that we are tracking information about *quantities* (`j`'s value), not *variables* (`j` itself). For example, consider the following expression:

```
(let ((i j)) (if (< j 0) (foo i)))
```

Clearly, the test involving `j` determines information about the value of `i` since they *both name the same quantity*. In a quantity-based analysis, information is tracked for quantities, and quantities can be named by multiple variables. This information can then be associated with the variable references appearing in the program. For the purposes of keeping track of this information, we need names for quantities; variables can then be bound to these quantity names (which are called *qnames*), which in turn have associated abstract properties.

In principle, calls to primops such as `+` or `cons` create new qnames since these operations involve the creation of new computational quantities. On the other hand, lambda binding simply involves the binding of a variable to an already existing qname. In practice, extra qnames often must be introduced since it can be difficult to determine at compile-time which qname a variable is bound to. Consider, for example, the following procedure:

```
(define (foo x y) (if (integer? x) y 3))
```

It might be the case that all calls to `foo` are of the form `(foo a a)`, in which case `x` and `y` can refer to the same qname. But if the analysis cannot determine this fact at compile time, `x` and `y` must be allocated two distinct qnames; hence determining information about `x`'s value will not shed light on `y`'s value.

As another example, consider the vector reference in the expression:

```
(let ((y (vector-ref vec i))) ...)
```

Now, `y` is certainly bound to an existing quantity, but it is unlikely that a compile-time analysis will be able to determine which one. So, a new qname must be assigned to `y`'s binding.

In general, then, a conservative, computable, quantity-based analysis approximates the tracking of information on run-time values by introducing new qnames whenever it is unable to determine to which existing qname a variable is bound. These extra qnames limit the precision of the analysis, but force the analysis to err conservatively.

9.3 Exact Type Recovery

Following the NSAS approach, the first step towards type recovery is to define an exact analysis that will capture the notion of type recovery. Our exact semantics, which we will call ETREC, does not have to be computable; we will concern ourselves with a computable approximation in section 9.4.

For the purposes of this chapter, we'll drop back to the simpler, less formal semantics of chapter 3. So the semantics will have no store or side-effects, run-time error checks will be omitted, we'll gloss over the details of the nb gensym function, and the semantic functions will be curried over their multiple arguments instead of taking single tuples. These features can be restored as we did going from the informal control-flow semantics of chapter 3 to the formal semantics of chapter 4.

Exact type recovery gives us a *type cache*:

> A *type cache* for a CPS Scheme program P is a function δ that, for each variable reference r and each binding contour b over r, returns $\delta\langle r,\, b\rangle$, a type of the value to which r could evaluate in contour b.

Once we've computed a type cache, we can easily find the type for any variable reference $r{:}v$ in our program:

$$\text{RefType}[\![r{:}v]\!] = \bigsqcup_b \delta\langle r,\, b\rangle.$$

Here we are joining in the type lattice all the various types that a reference r had in all its various binding contexts. The least-upper bound of these types is a legitimate static type for r.

9.3.1 Basic Structure

The ETREC semantics maps a CPS Scheme program to its type cache. Its structure, however, is very similar to a standard semantics for CPS Scheme. Before proceeding to the details of the type-recovery machinery, I'll briefly sketch out this basic structure.

The sets Bas, Proc, and D are just as in the standard semantics of chapter 3. The domain of type caches, TCache, is the cpo of partial functions from variable-reference/contour pairs to types.

$$\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{Proc} &= (\text{LAM} \times \text{BEnv}) + \text{PRIM} + \{\mathit{stop}\} \\
\text{D} &= \text{Proc} + \text{Bas} \\
\text{TCache} &= (\text{REF} \times \text{CN}) \rightarrow \text{Type}
\end{aligned}$$

The basic environment and procedure structures are identical to the standard semantics case. Note that we've named the VAR $\times$ CN set of variable bindings; this will be convenient later on.

$$\begin{aligned}
&\text{CN} \quad \textit{Contours} \\
\text{VB} &= \text{VAR} \times \text{CN} \\
\text{BEnv} &= \text{LAB} \rightharpoonup \text{CN} \\
\text{VEnv} &= \text{VB} \rightharpoonup \text{D}
\end{aligned}$$

In the ETREC semantics, the procedure and argument forms of a call expression are evaluated to values in D with the $\mathcal{A}_v$ function, which is identical to the $\mathcal{A}$ function of the standard semantics.

$$\mathcal{A}_v\text{: ARG} \cup \text{FUN} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightharpoonup \text{D}$$

$$\begin{aligned}
\mathcal{A}_v\, [\![k]\!]\; \beta\; ve &= \mathcal{K}\; k \\
\mathcal{A}_v\, [\![prim]\!]\; \beta\; ve &= prim \\
\mathcal{A}_v\, [\![\ell]\!]\; \beta\; ve &= \langle \ell,\, \beta \rangle \\
\mathcal{A}_v\, [\![v]\!]\; \beta\; ve &= ve\, \langle v,\, \beta(binder\; v) \rangle
\end{aligned}$$

Procedure calls are handled by the $\mathcal{C}$ and $\mathcal{F}$ functions.

$$\mathcal{C}\text{: CALL} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightarrow \text{QEnv} \rightarrow \text{TTab} \rightarrow \text{TCache}$$

$$\begin{aligned}
&\mathcal{C}\, [\![(f\;\; a_1 \ldots a_n)]\!]\; \beta\; ve\; qe\; \tau = \\
&\qquad\qquad \delta \sqcup \mathcal{F}\; f'\; av\; qv\; ve\; qe\; \tau' \\
&\qquad\qquad \text{where } av{\downarrow}i = \mathcal{A}_v\; a_i\; \beta\; ve \\
&\qquad\qquad\qquad f' = \mathcal{A}_v\; f\; \beta\; ve \\
&\qquad\qquad\qquad \tau' = f \in \text{REF} \;\longrightarrow\; \tau \sqcap \top[\mathcal{A}_q\; f\; \beta\; qe \mapsto type/proc] \\
&\qquad\qquad\qquad\qquad \text{otherwise } \tau \\
&\qquad\qquad\qquad \delta = [\langle r_i,\, \beta\,(binder\, v_i)\rangle \mapsto \tau(\mathcal{A}_q\; v_i\; \beta\; qe)] \quad \forall f, a_i \in \text{REF} \\
&\qquad\qquad\qquad\qquad \text{where } a_i = [\![r_i{:}v_i]\!] \text{ and } f = [\![r_0{:}v_0]\!] \\
&\qquad\qquad\qquad qv{\downarrow}i = a_i \in \text{REF} \;\longrightarrow\; \mathcal{A}_q\; a_i\; \beta\; qe \quad \text{(quantity)} \\
&\qquad\qquad\qquad\qquad \text{otherwise } \mathcal{A}_t\; a_i \qquad\qquad \text{(type)}
\end{aligned}$$

$\mathcal{C}$ takes five arguments: a call expression, the lexical contour environment β, the variable environment ve, and two others used for type recovery. $\mathcal{C}$ processes the first three arguments just as in the standard semantics case; we will return to the processing of the type information in the next subsection.

The $\mathcal{F}$ function applies procedures to arguments.

$$\mathcal{F}\text{: Proc} \rightarrow \text{D}^* \rightarrow (\text{Quant} + \text{Type})^* \rightarrow \text{VEnv} \rightarrow \text{QEnv} \rightarrow \text{TTab} \rightarrow \text{TCache}$$

$$\begin{aligned}
&\mathcal{F}\; \langle [\![\ell{:}(\lambda\;\; (v_1 \ldots v_n)\;\; c)]\!],\, \beta \rangle\;\; av\;\; qv\;\; ve\;\; qe\;\; \tau = \\
&\qquad\qquad \mathcal{C}\; c\; \beta'\; ve'\; qe''\; \tau' \\
&\qquad\qquad \text{where } \;\; b = nb \\
&\qquad\qquad\qquad \beta' = \beta[\ell \mapsto b] \\
&\qquad\qquad\qquad ve' = ve[\langle v_i,\, b\rangle \mapsto av{\downarrow}i] \\
&\qquad\qquad\qquad qe' = qe[\langle v_i,\, b\rangle \mapsto qv{\downarrow}i] \;\; \forall i \;\ni\; qv{\downarrow}i \in \text{Quant} \qquad (*) \\
&\qquad\qquad\qquad \left.\begin{aligned} qe'' &= qe'[\langle v_i,\, b\rangle \mapsto \langle v_i,\, b\rangle] \\ \tau' &= \tau[\langle v_i,\, b\rangle \mapsto qv{\downarrow}i] \end{aligned}\right\} \forall i \;\ni\; qv{\downarrow}i \in \text{Type} \quad (**)
\end{aligned}$$

Besides the extra arguments used for tracking type information, qv, qe, and τ, and their associated processing, $\mathcal{F}$ is identical to the standard semantics case.

To fully specify $\mathcal{F}$, we must also give the functional representation for each primop and the terminal $stop$ continuation. We will return to this after considering the mechanics of type-tracking. We also need to specify how $\mathcal{C}$ handles call forms that are `letrec` expressions instead of simple procedure calls. Again, ignoring for the moment the type processing, this case is just like the standard semantics case:

$$
\begin{aligned}
\mathcal{C}\,[\![c\text{:}\texttt{(letrec ((}f_1\ \ l_1\texttt{)}\ldots\texttt{)}\ \ c'\texttt{)}]\!]\ \beta\ ve\ qe\ \tau = \mathcal{C}\ c'\ \beta'\ ve'\ qe'\ \tau'& \\
\text{where } b = nb& \\
\beta' = \beta[c \mapsto b]& \\
ve' = ve[\langle f_i,\, b\rangle \mapsto \mathcal{A}_v\ l_i\ \beta'\ ve]& \\
qe' = qe[\langle f_i,\, b\rangle \mapsto \langle f_i,\, b\rangle]& \\
\tau' = \tau[\langle f_i,\, b\rangle \mapsto type/proc]&
\end{aligned}
$$

9.3.2 Quantities and Types

We can now turn to the details of tracking type information through the ETREC interpretation. This will involve the quantity analysis model discussed in section 9.2. The general type recovery strategy is straightforward:

- Whenever a new computational quantity is created, it is given a unique qname. Over the lifetime of a given quantity, it will be bound to a series of variables as it is passed around the program. As a quantity (from D) is bound to variables, we also bind its qname (from Quant) with these variables.
- As execution proceeds through the program, we keep track of all currently known information about quantities. This takes the form of a *type table* τ that maps qnames to type information. Program structure that determines type information about a quantity enters the information into the type table, passing it forward.
- When a variable reference is evaluated, we determine the qname it is bound to, and index into the type table to discover what is known at that point in the computation about the named quantity. This tells us what we know about the variable reference in the current context. This information is entered into the answer type cache.

This amounts to instrumenting our standard semantics to track the knowledge determined by run-time type tests, recording relevant snapshots of this knowledge in the answer type cache as execution proceeds through the program.

The first representational decision is how to choose qnames. A simple choice is to name a quantity by the first variable binding $\langle v,\, b\rangle$ to which it is bound. Thus, the qname for the cons cell created by

```
(cons 1 2 (λ (x) ...))
```

is $\langle[\![\texttt{x}]\!],\, b\rangle$, where b is the contour created upon entering `cons`'s continuation. When future variable bindings are made to this cons cell, the binding will be to the qname $\langle[\![\texttt{x}]\!],\, b\rangle$. Thus, our qname set Quant is just the set of variable bindings VB:

$$\text{Quant} = \text{VB}.$$

Having chosen our qnames, the rest of the type-tracking machinery falls into place. A quantity environment ($qe \in$ QEnv) is a mapping from variable bindings to qnames. The qname analog of the variable environment ve, the quantity environment is a global structure that is augmented monotonically over the course of program execution. A type table ($\tau \in$ TTab) is a mapping from qnames to types. Our types are drawn from some standard type lattice; our toy dialect will use the obvious lattice over the three basic types: procedure, false, and integer.

$$\begin{aligned} \text{Type} &= \{type/proc, type/int, type/false, \bot, \top\} \\ \text{QEnv} &= \text{VB} \rightharpoonup \text{Quant} \\ \text{TTab} &= \text{Quant} \rightarrow \text{Type} \end{aligned}$$

We may now consider the workings of the type-tracking machinery in the $\mathcal{F}$ and $\mathcal{C}$ functions. Looking at $\mathcal{F}$, we see that the function linkage requires three additional arguments to be passed to a procedure: the quantity vector qv, the quantity environment qe, and the current type table τ. The quantity environment and type table are as discussed above. The quantity vector gives quantity information about the arguments passed to the procedure. Each element of qv is either a qname or a type. If it is a qname, it names the quantity being passed to the procedure as its corresponding argument. However, if a computational quantity has been created by the procedure's caller, then it has yet to be named — a quantity is named by the first variable binding to which it is bound, so the current procedure is responsible for assigning a qname to the new quantity when it binds it. In this case, the corresponding element of the quantity vector gives the initial type information known about the new quantity. Consider the cons example given above. The `cons` primop creates a new quantity — a cons cell — and calls the continuation `(`λ` (x) ...)` on it. Since the cons cell is a brand new quantity, it has not yet been given a qname; the continuation binding it to `x` will name it. So instead of passing the continuation a qname for the cell, the `cons` primop passes the type $type/pair$ (*i.e.*, $qv = \langle type/pair \rangle$) giving the quantity's initial type information.

We can see this information being used in the $\mathcal{F}$ equation. The line marked with (*) binds qnames from the quantity vector to their new bindings $\langle v_i,\ b\rangle$. The lines marked with (**) handle new quantities which do not yet have qnames. A new quantity must be assigned its qname, which for the ith argument is just its variable binding $\langle v_i,\ b\rangle$. We record the initial type information ($qv{\downarrow}i \in$ Type) known about the new quantity in the type table τ'. The new quantity environment qe'' and type table τ' are passed forward to the $\mathcal{C}$ function.

$\mathcal{C}$ receives as type arguments the current quantity environment qe, and the current type table τ. Before jumping off to the called procedure, $\mathcal{C}$ must perform three type-related tasks:

- Record in the final answer cache the type of each variable reference in the call. If a call argument a_i is a variable reference, $a_i = [\![r_i{:}v_i]\!]$, then it is evaluated to a qname by the auxiliary function $\mathcal{A}_q$, the qname analog to the $\mathcal{A}_v$ function. If the procedure form f is a variable reference, $f = [\![r_0{:}v_0]\!]$, it is similarly evaluated to a qname. The qname is used to index into the type table τ, giving the type information currently known about the quantity bound to variable v_i. Call this type t. We record in the type cache that the variable reference r_i evaluated to a value of type t in contour $\beta(\mathit{binder}\ v_i)$. This is the contribution δ that $\mathcal{C}$ makes to the final answer for the current call.

- Compute the quantity vector qv to be passed to the called procedure f. If a_i is a variable reference, its corresponding element in qv is the qname it is bound to in the current context; this is computed by the $\mathcal{A}_q$ auxiliary. On the other hand, if the argument a_i is a constant or a lambda, then it is considered a new, as-yet-unnamed quantity. The auxiliary $\mathcal{A}_t$ determines its type; the called procedure will be responsible for assigning this value a qname.

- Finally, note that $\mathcal{C}$ can do a bit of type recovery. If f does not evaluate to a procedure, the computation becomes undefined at this call. We may thus assume that f's quantity is a procedure in following code. We record this information in the outgoing type table τ': if f is a variable reference, we find the qname it is bound to, and intersect $\mathit{type/proc}$ with the type information recorded for the qname in τ. If f is not a variable reference, it isn't necessary to do this. (Note that in the `letrec` case, $\mathcal{C}$ performs similar type recovery, recording that the new quantities bound by the `letrec` all have type $\mathit{type/proc}$. This is all the type manipulation $\mathcal{C}$ does for the `letrec` case.)

$$\mathcal{A}_q\ [\![v]\!]\ \beta\ qe = qe\ \langle v,\ \beta\,(\mathit{binder}\ v)\rangle$$

$$\begin{aligned}
\mathcal{A}_t\ [\![(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!] &= \mathit{type/proc} \\
\mathcal{A}_t\ [\![n]\!] &= \mathit{type/int} \quad (\text{numeral } n) \\
\mathcal{A}_t\ [\![\texttt{\#f}]\!] &= \mathit{type/false}
\end{aligned}$$

Most of the type information is recovered by primop applications, defined by $\mathcal{F}$. We'll use `+` and `test-integer` for representative examples.

$$\begin{aligned}
\mathcal{F}\ [\![\texttt{+}]\!]\ \langle x,\ y,\ k\rangle\ \langle qx,\ qy,\ qk\rangle\ \ ve\ qe\ \tau = {} & \mathcal{F}\ k\ \langle x+y\rangle\ \langle ts\rangle\ \ ve\ qe\ \tau'' \\
& \text{where } tx = \text{QT}\ qx\ \tau \\
& \qquad\quad ty = \text{QT}\ qy\ \tau \\
& \qquad\quad ts = \text{infer+}\langle tx,\ ty\rangle \\
& \qquad\quad \tau' = \text{Tu}\ qx\ (tx \sqcap \mathit{type/int})\ \tau \\
& \qquad\quad \tau'' = \text{Tu}\ qy\ (ty \sqcap \mathit{type/int})\ \tau'
\end{aligned}$$

The `+` primop takes three arguments: two values to be added, x and y, and a continuation k. Hence the argument vector and quantity vector have three components each. The primop assigns tx and ty to be the types that execution has constrained x and y to have. These types are computed by the auxiliary function QT:(Quant + Type) $\rightarrow$ (Quant $\rightarrow$ Type) $\rightarrow$ Type.

$$\begin{aligned}
\text{QT}\ q\ \tau = {} & q \in \text{Quant} \ \longrightarrow\ \tau\ q \\
& q \in \text{Type} \ \ \longrightarrow\ q
\end{aligned}$$

QT maps an element q from a quantity vector to type information. If the element is a qname, then QT looks up its associated type information in the type table τ. If the element is a type (because the corresponding quantity is a new, unnamed one), then it is the initial type information for the quantity, and so QT simply returns that type. Having retrieved the type information for its arguments x and y, + can then compute the type ts of its result sum. This is handled by the auxiliary function infer+, whose details are not presented. Infer+ is a straightforward type computation: if both arguments are known to be integers, then the result is known to be an integer. If our language includes other types of numbers, infer+ can do the related inferences. For example, it might infer that the result of adding a floating point number to another number is a floating point number. However infer+ computes its answer, ts must be a subtype of the number type, since if control proceeds past the addition, + is guaranteed to produce a number. Further, + can make inferences about subsequent references to its arguments: they must be numbers (else the computation becomes undefined at the addition). The auxiliary function Tu updates the incoming type table with this inference; the result type table τ'' is passed forward to the continuation.

$$\begin{aligned} \text{Tu}\ q\ t\ \tau = q \in \text{Quant} &\longrightarrow \tau[q \to t] \\ q \in \text{Type} &\longrightarrow \tau \end{aligned}$$

Tu takes three arguments: an element q from a quantity vector (*i.e.*, a qname or type), a type t, and a type table τ. If q is a qname, the type table is updated to map $q \mapsto t$. Otherwise, the type table is returned unchanged (the corresponding quantity is ephemeral, being unnamed by a qname; there is no utility in recording type information about it, as no further code can reference it). With the aid of Tu, + constrains its arguments a and b to have the number type by intersecting the incoming type information tx and ty with $type/number$ and updating the outgoing type table τ'' to reflect this.[1] The sum $x + y$, its initial type information ts, and the new type table τ'' are passed forward to the continuation, thus continuing the computation. To simplify the equations, we omit the case of halting the computation if there is an error in the argument values, *e.g.*, x or y are not numbers.[2]

$$\begin{aligned} &\mathcal{F}\,[\![\texttt{test-integer}]\!]\ \langle x,\, k_0,\, k_1\rangle\ \langle qx,\, qk_0,\, qk_1\rangle\ ve\ qe\ \tau = \\ &\qquad x \in \mathcal{Z} \longrightarrow \mathcal{F}\ k_0\ \langle\rangle\ \langle\rangle\ ve\ qe\ \tau_t \\ &\qquad \text{otherwise}\ \mathcal{F}\ k_1\ \langle\rangle\ \langle\rangle\ ve\ qe\ \tau_f \\ &\qquad \text{where}\ tx = \text{QT}\ qx\ \tau \\ &\qquad\qquad \tau_f = \text{Tu}\ qx\ (tx - type/int)\ \tau \\ &\qquad\qquad \tau_t = \text{Tu}\ qx\ (tx \sqcap type/int)\ \tau \end{aligned}$$

[1] Note that the + primop is missing an opportunity to recover some available type information: it is not recovering type information about its continuation. For example, in code executed after the call to +'s continuation, we can assume that the quantity called has type *type/proc*. As discussed in section 3.8.3, all continuations, variables bound to continuations, and calls to continuations in a CPS Scheme program are introduced by the CPS converter; the converter can mark these forms as it introduces them. So the types of continuation variables can be inferred statically, and there's no point in tracking them in our type-recovery semantics.

[2] These equations will consistently omit the handling of run-time errors, *e.g.*, applying a two-argument procedure to three values, dividing by zero, or applying + to a non-number. This is simple to remedy: run-time errors are defined to terminate the program immediately and return the current type cache.

The primop `test-integer` performs a conditional branch based on a type test. It takes as arguments some value x and two continuations k_0 and k_1. If x is an integer, control is passed to k_0, otherwise control is passed to k_1. `Test-integer` uses QT to look up *tx*, the type information recorded in the current type table τ for x's qname *qx*. There are two outgoing type tables computed, one which assumes the test succeeds (τ_t), and one which assumes the test fails (τ_f). If the test succeeds, then *qx*'s type table entry is updated by Tu to constrain it to have the integer type. Similarly, if the test fails, *qx* has the integer type subtracted from its known type. The appropriate type table is passed forward to the continuation selected by `test-integer`, producing the answer type cache.

Finally, we come to the definition of the terminal $stop$ continuation, retrieved by $\mathcal{F}$. Calling the $stop$ continuation halts the program; no more variables are referenced. So the semantic function for $stop$ just returns the bottom type cache $\perp$:

$$\mathcal{F}\ [\![stop]\!]\ av\ qv\ ve\ qe\ \tau = \perp.$$

The bottom type cache is the one that returns the bottom type for any reference: $\perp_{\text{TCache}} = \lambda vb.\ \perp_{\text{Type}}$. Note that in most cases, the bottom type cache returned by calling $stop$ is *not* the final type cache for the entire program execution. Each call executed in the course of the program execution will add its contribution to the final type cache. This contribution is the expression δ in the $\mathcal{C}$ equation for simple call expressions (page 114).

Having defined all the ETREC type tracking machinery, we can invoke it to compute the type cache δ for a program by simply closing the top-level lambda ℓ in the empty environment, and passing it the terminal $stop$ continuation as its single argument:

$$\delta = \mathcal{F}\ \langle \ell,\ \texttt{[]} \rangle\ \langle stop \rangle\ \langle type/proc \rangle\ \texttt{[]}\ \texttt{[]}\ \texttt{[]}.$$

9.4 Approximate Type Recovery

9.4.1 Initial Approximations

Having defined our exact type recovery semantics, we can consider the problem of abstracting it to a computable approximation, while preserving some notion of correctness and as much precision as possible.

Let's start with the standard set of approximations that we used for simple control-flow analysis:

- We run the interpretation forward for both continuations to a conditional branch, joining the result type caches together.

- We fold the infinite contour set down to a finite contour set. The variable environment maps abstract variable bindings to sets of values.

9.4.2 Problems with the Abstraction

Because of the environment problem, this standard set of approximations breaks the correctness of our semantics. The reason is that by folding together multiple bindings of the same variable, information can leak across quantity boundaries. We can see this by adapting the environment puzzle of chapter 8 to the type recovery case:

```
(let ((f (λ (x h) (if (integer? x) (h)
                      (λ () x)))))
     (f 3 (f 2.7 '#f)))
```

Suppose that the procedural abstraction used by our analysis identifies together the contours created by the two calls to `f`. Consider the second execution of the `f` procedure: the variable `x` is tested to see if its value (3) is an integer. It is, so we jump to the value of `h`, which is simply `(λ () x)`. Now, we have established that `x` is bound to an integer, so we can record that this reference to `x` is an integer reference — which is an error, since `h` $=$ `(λ () x)` is closed over a different contour, binding `x` to a non-integer, 2.7. We tested one binding of `x` and referred to a different binding of `x`. Our analysis got confused because we had identified these two bindings together, so that the information gathered at the test was erroneously applied at the reference.

This is a deep problem of the approximation. Fixing it requires going to a reflow-based analysis.

9.4.3 Special Contours

To repeat the basic idea of the previous chapter, the problem with abstract variable bindings is that they refer to values bound at temporally distinct points in the abstract interpretation. We don't want to confuse the binding that we are tracking with past or future bindings in the abstract interpretation that will use the same abstract contour.

The fix is to define a special contour that will stand for the particular contour whose bindings we are tracking for a single reflow. Flow analysis then tracks this special contour, whose bindings will never be identified with any other bindings of the same variables. Information associated with quantities bound in this special contour cannot be confounded. The other approximate contours are used only to provide the approximate control flow information for tracing through the program's execution. We still have only a finite number of contours — the finite number of approximate contours plus the one special contour — so our analysis is still computable.

For example, suppose we know that abstract procedure $f = \left\langle [\![\ell\texttt{:(}\lambda\texttt{ (x y) ...)}]\!], \widehat{\beta} \right\rangle$ is called from abstract call context $\left\langle [\![c\texttt{:(g 3 false)}]\!], \widehat{\beta}' \right\rangle$. We can do a partial type recovery for references to `x` and `y`. When we perform the initial function call of the reflow to f, we allocate the unique contour $special$. The variables we are tracking are bound in this contour, with variable bindings $\langle [\![\mathrm{x}]\!], special \rangle$ and $\langle [\![\mathrm{y}]\!], special \rangle$. We create new qnames

$$
\begin{aligned}
\widehat{\text{Proc}} &= (\text{LAM} \times \widehat{\text{BEnv}}) + \text{PRIM} + \{stop\} \\
\hat{\text{D}} &= \mathcal{P}(\widehat{\text{Proc}}) \\
\widehat{\text{TCache}} &= (\text{REF} \times \widehat{\text{CN}}) \to \text{Type} \\
\widehat{\text{TCache}}_{\text{spec}} &= \text{REF} \to \text{Type} \\
\widehat{\text{CCtxt}} &= \text{CALL} \times \widehat{\text{BEnv}} \\
\widehat{\text{CN}} &= \text{LAB} + \{special\} \\
\widehat{\text{VB}} &= \text{VAR} \times \widehat{\text{CN}} \\
\widehat{\text{BEnv}} &= \text{LAB} \to \widehat{\text{CN}} \\
\widehat{\text{VEnv}} &= \widehat{\text{VB}} \to \hat{\text{D}} \\
\\
\widehat{\text{Quant}} &= \widehat{\text{VB}}_{\perp} \\
\widehat{\text{TTab}} &= \widehat{\text{Quant}} \to \text{Type}
\end{aligned}
$$

$$
\begin{aligned}
\hat{\mathcal{C}}&: \text{CALL} \to \widehat{\text{BEnv}} \to \widehat{\text{VEnv}} \to \widehat{\text{TTab}} \to \widehat{\text{TCache}}_{\text{spec}} \\
\hat{\mathcal{F}}&: \widehat{\text{Proc}} \to \hat{\text{D}}^* \to \widehat{\text{Quant}}^* \to \widehat{\text{VEnv}} \to \widehat{\text{TTab}} \to \text{LAB} \to \widehat{\text{TCache}}_{\text{spec}} \\
\hat{\mathcal{F}}_{\text{spec}}&: \widehat{\text{Proc}} \to \hat{\text{D}}^* \to \text{Type}^* \to \widehat{\text{VEnv}} \to \text{LAB} \to \widehat{\text{TCache}}
\end{aligned}
$$

Figure 9.2: Abstract Domains and Functionalities

for the arguments passed on this call to f, which are just the new special variable bindings $\langle [\![\text{x}]\!], special\rangle$ and $\langle [\![\text{y}]\!], special\rangle$. Our initial type table

$$\tau = [\langle [\![\text{x}]\!], special\rangle \mapsto type/int,\ \langle [\![\text{y}]\!], special\rangle \mapsto type/false]$$

is constructed from what is known about the types of the arguments in c (this may be trivially known, if the arguments are constants, or taken from a previous iteration of this algorithm, if the arguments are variables).

We may now run our interpretation forward, tracking the quantities bound in the special context. Whenever we encounter a variable reference to x or y, if the reference occurs in the contour $special$, then we can with certainty consult the current type table τ to obtain the type of the reference. Other contours over x and y won't confuse the analysis. Note that we are only tracking type information associated with the variables x and y, for a single call to ℓ. In order to completely type analyse the program, we must repeat our analysis for each lambda for each call to the lambda. This brings us to the Reflow semantics.

9.4.4 The Reflow Semantics

The abstract domains and functionalities of the Reflow semantics are given in figure 9.2. The abstract domains show the approximations discussed in subsection 9.4.1: basic values have been dropped, the contour set $\widehat{\mathrm{CN}}$ is finite, and elements of $\hat{\mathrm{D}}$ are sets of abstract procedures, not single values. We'll use the 1CFA abstraction, so the contours are call site labels plus the token $special$ to mark the special reflow contour.

Notice that we have two different abstract variants of the $\mathcal{F}$ function: $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{F}}_{\text{spec}}$. The $\widehat{\mathcal{F}}$ function is the ordinary variant which allocates abstract contours when applying procedures. The $\widehat{\mathcal{F}}_{\text{spec}}$ function is the special variant which allocates the special contour when it is applied. The $\widehat{\mathcal{F}}_{\text{spec}}$ function is used only once per reflow, during the initial call.

The idea of the Reflow semantics is to track only one closure's variables at a time. This is done by the Reflow function:

$$\text{Reflow}: \widehat{\text{CCtxt}} \rightarrow \widehat{\text{VEnv}} \rightarrow \text{Type}^* \rightarrow \widehat{\text{TCache}}.$$

For example, Reflow $\left\langle [\![\texttt{(f a b)}]\!], \widehat{\beta} \right\rangle\ \widehat{ve}\ \langle type/int,\ type/proc \rangle$ restarts the program at the call `(f a b)`, in the context given by the abstract contour environment $\widehat{\beta}$ and variable environment $\widehat{ve}$, assuming `a` has type $type/int$ and `b` has type $type/proc$. Reflow runs the program forward, tracking only the variables bound by the initial call to `f`, returning a type cache giving information about references to these variables. This is done by allocating a single, special contour for the initial procedure called from `(f a b)`, and tracking the variables bound in this contour through an abstract execution of the program.

The initial type vector given to Reflow comes from an auxiliary function, TVInit:

$$\begin{aligned}
&\text{TVInit}: \widehat{\text{CCtxt}} \rightarrow \widehat{\text{TCache}} \rightarrow \text{Type}^* \\
&\text{TVInit} \left\langle [\![(f\ \ a_1 \ldots a_n)]\!], \widehat{\beta} \right\rangle \hat{\delta} = \langle t_1, \ldots, t_n \rangle \\
&\qquad\qquad\qquad\qquad \text{where } t_i = a_i \in \text{REF} \ \longrightarrow\ \hat{\delta} \left\langle a_i,\ \widehat{\beta}(binder\ a_i) \right\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise } \mathcal{A}_t\ a_i
\end{aligned}$$

TVInit takes a call context and a type cache, and returns the types of all the arguments in the call. If the argument is a variable reference, the type cache is consulted; if the argument is a constant or a lambda, the auxiliary function $\mathcal{A}_t$ gives the appropriate type.

If we wish to restart our program at an arbitrary call context $\left\langle c,\ \widehat{\beta} \right\rangle$ with Reflow, we need a variable environment $\widehat{ve}$ that approximates all the variable environments that ever pertained at this call context. This is easy to handle: we always use the final variable environment that was present at the end of a control-flow analysis performed using the single-threaded environment algorithm of section 5.3. Since the variable environment is only augmented monotonically during the course of this analysis, the terminal environment is a superset of every intermediate variable environment through which the interpretation evolved. So, before performing the reflow analysis, we first perform a control-flow analysis with the single-threaded environment algorithm. This produces two items critical for the reflow analysis: the call cache $\hat{\gamma}$ and terminal variable environment $\widehat{ve}_{final}$. The domain of the call cache $Dom(\hat{\gamma})$ gives us the call contexts from which we must reflow, and $\widehat{ve}_{final}$ allows us to restart the interpretations in mid-stride.

Given TVInit and Reflow, we can construct a series of approximate type caches, converging to a fixed point. The initial type cache $\hat{\delta}_0$ is the most precise; at each iteration, we redo the reflow analysis assuming type cache $\hat{\delta}_i$, computing a weaker type cache $\hat{\delta}_{i+1}$. The limit $\hat{\delta}$ is the final result. The recomputation of each successive type cache is straightforward: for every call context $\left\langle c,\, \widehat{\beta}\right\rangle$ in the domain of call cache $\hat{\gamma}$ (that is, every call context recorded by the preliminary control-flow analysis), we use the old type cache to compute the types of the arguments in the call, then reflow from the call, tracking the variables bound by the call's procedure, assuming the old type information. The returned type caches are joined together, yielding the new type cache. So the new type cache is the one we get by assuming the type assignments of the old type cache. A fixed point is a legitimate type assignment. Since all of our abstract domains are of finite size, and our type lattice has finite height, the least fixed point is computable.

$$\begin{aligned}
\hat{\delta}_0 &= \lambda \left\langle r,\, \hat{b}\right\rangle .\ \bot \\
\hat{\delta}_{i+1} &= \hat{\delta}_i \ \sqcup \bigsqcup_{\langle c,\,\beta\rangle \in Dom(\hat{\gamma})} \text{Reflow}\ \langle c,\, \beta\rangle\ \widehat{ve}_{\mathit{final}}\ (\text{TVInit}\ \left\langle c,\, \widehat{\beta}\right\rangle\ \hat{\delta}_i) \\
\hat{\delta} &= \bigsqcup_i \hat{\delta}_i
\end{aligned}$$

Before we get to the machinery of the Reflow function itself, let us examine the functions that evaluate arguments in call expressions:

$$\begin{array}{ll}
\widehat{\mathcal{A}}_v\ [\![k]\!]\ \widehat{\beta}\ \widehat{ve} = \emptyset & \widehat{\mathcal{A}}_v\ [\![prim]\!]\ \widehat{\beta}\ \widehat{ve} = \{prim\} \\
\widehat{\mathcal{A}}_v\ [\![\ell]\!]\ \widehat{\beta}\ \widehat{ve} = \left\{\left\langle \ell,\, \widehat{\beta}\right\rangle\right\} & \widehat{\mathcal{A}}_v\ [\![v]\!]\ \widehat{\beta}\ \widehat{ve} = \widehat{ve}\left\langle v,\, \widehat{\beta}(\mathit{binder}\, v)\right\rangle
\end{array}$$

$$\begin{aligned}
\widehat{\mathcal{A}}_q\ [\![v]\!]\ \widehat{\beta} = {} & \widehat{\beta}(\mathit{binder}\, v) = \mathit{special} \ \longrightarrow\ \langle v,\, \mathit{special}\rangle \\
& \text{otherwise } \bot
\end{aligned}$$

The $\widehat{\mathcal{A}}_v$ function evaluates call arguments, and is the straightforward abstraction of its counterpart in the ETREC semantics. The $\widehat{\mathcal{A}}_q$ function is a little more subtle. Since the Reflow semantics only tracks variables bound in the special contour, $\widehat{\mathcal{A}}_q$ only returns quantities for these bindings; approximate bindings are mapped to an undefined qname, represented with $\bot$. This means that we no longer need a quantity environment to map variable bindings to qnames — the special bindings of the variables being tracked are also the qnames of their bound values.

Now we are in a position to examine the machinery that triggers off a single wave of special contour type tracking: the Reflow function, and its auxiliary $\widehat{\mathcal{F}}_{\text{spec}}$ function.

$$\begin{aligned}
\text{Reflow}\ \langle [\![c{:}(f\ \ a_1 \ldots a_n)]\!],\, \beta\rangle\ \widehat{ve}\ tv = {} & \bigsqcup_{f' \in F} \widehat{\mathcal{F}}_{\text{spec}}\ f'\ \widehat{av}\ tv\ \widehat{ve}\ c \\
& \text{where } F = \widehat{\mathcal{A}}_v\ f\ \widehat{\beta}\ \widehat{ve} \\
& \qquad \widehat{av}{\downarrow}i = \widehat{\mathcal{A}}_v\ a_i\ \widehat{\beta}\ \widehat{ve}
\end{aligned}$$

Reflow simply reruns the interpretation from each possible procedure that could be called from call context $\langle c,\, \beta\rangle$. Each procedure is applied with the "special" function $\widehat{\mathcal{F}}_{\text{spec}}$, which arranges for the initial application of the procedure to be a special one. The 1CFA contour c passed to $\widehat{\mathcal{F}}_{\text{spec}}$ is the abstract contour that $\mathit{special}$ will represent during this reflow. The type caches resulting from each call are joined together into the result cache.

The special application of a procedure performed by $\widehat{\mathcal{F}}_{\text{spec}}$ is done only at the beginning of the reflow. Instead of allocating the 1CFA contour $\hat{b}$, $\widehat{\mathcal{F}}_{\text{spec}}$ allocates the $\mathit{special}$ contour in its place, designating it the one and only special contour in a given execution thread. The incoming values are bound in the outgoing variable environment $\widehat{ve}'$ under the special contour. The incoming type information is used to create the initial type table $\hat{\tau}$ passed forward to track the values bound under the special contour. The rest of the computation is handed off to the $\widehat{\mathcal{C}}$ procedure. $\widehat{\mathcal{C}}$ is similar to $\mathcal{C}$ with the exception that it only records the type information of references that are bound in the special contour.

$$\begin{aligned}
\widehat{\mathcal{F}}_{\text{spec}} \left\langle [\![\ell\text{:}(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!],\, \widehat{\beta} \right\rangle\, \widehat{av}\ tv\ \widehat{ve}\ \hat{b} = \text{despec}\ \hat{b}\ \left(\widehat{\mathcal{C}}\ c\ \widehat{\beta}'\ \ \widehat{ve}'\ \hat{\tau}\right) \\
\text{where } \hat{\tau} &= [\langle v_i,\, \mathit{special}\rangle \mapsto tv{\downarrow}i] \\
\widehat{\beta}' &= \widehat{\beta}\,[\ell \mapsto \mathit{special}] \\
\widehat{ve}' &= \widehat{ve} \sqcup [\langle v_i,\, \mathit{special}\rangle \mapsto \widehat{av}{\downarrow}i]
\end{aligned}$$

$$\begin{aligned}
&\text{despec: CN} \to \widehat{\text{TCache}}_{\text{spec}} \to \widehat{\text{TCache}} \\
&\text{despec}\ \hat{b}\ \ \hat{\delta}_{\text{spec}} = \lambda \left\langle r,\, \hat{b}' \right\rangle.\ \ \hat{b} = \hat{b}' \ \longrightarrow\ \hat{\delta}_{\text{spec}}\, r \\
&\qquad\qquad\qquad\qquad\qquad \text{otherwise } \bot
\end{aligned}$$

Because a single reflow tracks only the bindings of one special contour, the result type cache has information only for variable references bound in that contour. So the type cache returned by the $\widehat{\mathcal{C}}$ application indexes on variable references, not reference/contour pairs — the contour is understood to be the special contour. Now, for this particular reflow, the special contour allocated by $\widehat{\mathcal{F}}_{\text{spec}}$ stands for a particular abstract contour, $\hat{b}$. So the result type cache must be converted to a type cache that replaces the simple variable reference r entries with $\left\langle r,\, \hat{b} \right\rangle$ entries. This is done with the auxiliary despec function. Let

$$\hat{\delta} = \text{despec}\ \hat{b}\ \ \hat{\delta}_{\text{spec}}\,.$$

Then $\hat{\delta} \left\langle r,\, \hat{b} \right\rangle$ just returns $\hat{\delta}_{\text{spec}}\, r$, and $\hat{\delta} \left\langle r,\, \hat{b}' \right\rangle$ gives bottom when $\hat{b}' \neq \hat{b}$.

$\widehat{\mathcal{F}}_{\text{spec}}$ must also be defined over primops. Primops do not have variables to be tracked, so instead primops pass the buck to their continuations. The `+` primop uses its initial-type vector $\langle tx,\, ty,\, tk \rangle$ to compute the initial-type vector $\langle ts \rangle$ for its continuation. The `+` primop then employs $\widehat{\mathcal{F}}_{\text{spec}}$ to perform type recovery on the variable bound by its continuation.

$$\begin{aligned}
\widehat{\mathcal{F}}_{\text{spec}}\, [\![p\text{:+}]\!]\ \left\langle \hat{x},\, \hat{y},\, \hat{k} \right\rangle\ \langle tx,\, ty,\, tk \rangle\ \ \widehat{ve}\ \hat{b} = \bigsqcup_{k \in \hat{k}} \widehat{\mathcal{F}}_{\text{spec}}\ k\ \ \langle \emptyset \rangle\ \ \langle ts \rangle\ \ \widehat{ve}\ ic_p \\
\text{where } ts = \text{infer+}\langle tx,\, ty \rangle
\end{aligned}$$

The `test-integer` primop is even simpler. Since its continuations do not bind variables, there is nothing to track, so the function just immediately returns the bottom type cache.

$$\widehat{\mathcal{F}}_{\text{spec}}\, [\![\texttt{test-integer}]\!]\ \left\langle \hat{x},\, \hat{k}_0,\, \hat{k}_1 \right\rangle\ \langle tx,\, tk_0,\, tk_1 \rangle\ \ \widehat{ve}\ \hat{b} = \bot$$

Once the initial call to $\widehat{\mathcal{F}}_{\text{spec}}$ has triggered a wave of type recovery for a particular lambda's variables, the actual tracking of type information through the rest of the program execution is handled by the $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ functions.

$$
\begin{aligned}
\widehat{\mathcal{C}}\,[\![c{:}(f\ \ a_1 \ldots a_n)]\!]\ \widehat{\beta}\ \widehat{ve}\ \hat{\tau} = \hat{\delta} \sqcup \Big(\bigsqcup_{f' \in F} \widehat{\mathcal{F}}\ f'\ \widehat{av}\ \widehat{qv}\ \widehat{ve}\ \hat{\tau}'\ c \Big)& \\
\text{where } F = \widehat{\mathcal{A}}_v\ f\ \ \widehat{\beta}\ \ \widehat{ve}& \\
\widehat{av}{\downarrow}i = \widehat{\mathcal{A}}_v\ a_i\ \ \widehat{\beta}\ \ \widehat{ve}& \\
[\![r_i{:}v_i]\!] = a_i \quad (\forall a_i \in \text{REF})& \\
[\![r_0{:}v_0]\!] = f \quad (\text{if } f \in \text{REF})& \\
\hat{b}_i = \widehat{\beta}(binder\ v_i)& \\
\hat{q}_i = \widehat{\mathcal{A}}_q\ v_i\ \ \widehat{\beta}& \\
\widehat{qv} = \langle \hat{q}_1/\bot, \ldots, \hat{q}_n/\bot \rangle& \\
\hat{\delta} = [r_i \mapsto \hat{\tau}\ \hat{q}_i] \quad \forall \hat{q}_i \neq \bot& \\
\hat{\tau}' = \ \hat{q}_0 \neq \bot \ \longrightarrow\ \hat{\tau} \sqcap \top [\hat{q}_0 \mapsto type/proc]& \\
\text{otherwise } \hat{\tau}&
\end{aligned}
$$

As $\widehat{\mathcal{C}}$ evaluates its arguments, it checks to see if any are variables whose types are being tracked. Suppose an argument a_i is a variable reference $[\![r_i{:}v_i]\!]$. The variable v_i is being tracked if it is bound in the special contour, that is, if $\hat{b}_i = special$, where $\hat{b}_i = \widehat{\beta}(binder\ v_i)$. In this case, it will be mapped to a non-bottom qname by $\widehat{\mathcal{A}}_q$. If an argument is being tracked, we look up its current type $\hat{\tau}\ \hat{q}_i$, and record this in $\widehat{\mathcal{C}}$'s contribution $\hat{\delta}$ to the type cache. An outgoing type table $\hat{\tau}'$ is constructed, reflecting that f must be of type *type/proc* (again, note that this fact is only recorded in $\hat{\tau}'$ if f is a variable currently being tracked). The notation $\hat{q}_i/\bot$ in the construction of the outgoing quantity vector $\widehat{qv}$ means "the value of $\hat{q}_i$ if it is defined, otherwise $\bot$." In other words, if an argument a_i is a variable reference, then the corresponding element in $\widehat{qv}$ is $\hat{q}_i$. If the argument is a lambda or constant, then it clearly isn't one of the special qnames created by $\widehat{\mathcal{F}}_{\text{spec}}$ at the beginning of the reflow, so the corresponding element in $\widehat{qv}$ is just $\bot$. The rest of $\widehat{\mathcal{C}}$'s stucture is a straighforward abstraction of the exact analysis. In keeping with the 1CFA abstraction, the call site c is passed to $\widehat{\mathcal{F}}$ to be used for its abstract contour.

Since multiple contours are identified together in the abstract semantics, values in the approximate domain are *sets* of abstract procedures. Because of this, the call must branch to each of the possible procedures f' to which the expression f could evaluate. The result type caches are then all joined together.

An approximate, non-special contour is allocated when $\widehat{\mathcal{F}}$ causes control to enter a closure. Because multiple environments are thus identified together by $\widehat{\mathcal{F}}$ applications, it cannot track type information for the variables bound by the procedure. Hence $\widehat{\mathcal{F}}$ has a fairly simple definition for the closure case, just augmenting the environment structure and passing the closure's body c off to $\widehat{\mathcal{C}}$.

$$\widehat{\mathcal{F}}\ \left\langle [\![\ell\text{:}(\lambda\ \ (v_1\ldots v_n)\ \ c)]\!],\ \widehat{\beta}\right\rangle\ \widehat{av}\ \widehat{qv}\ \widehat{ve}\ \ \hat{\tau}\ \ \hat{b} = \widehat{\mathcal{C}}\ c\ \widehat{\beta}'\ \ \widehat{ve}'\ \ \hat{\tau}$$
$$\text{where } \widehat{\beta}' = \widehat{\beta}\left[\ell \mapsto \hat{b}\right]$$
$$\widehat{ve}' = \widehat{ve} \sqcup \left[\left\langle v_i,\ \hat{b}\right\rangle \mapsto \widehat{av}{\downarrow}i\right]$$

$\widehat{\mathcal{F}}$'s definition for the terminal $stop$ continuation is, again, trivial, ignoring its argument v and returning the bottom type cache:

$$\widehat{\mathcal{F}}\ [\![stop]\!]\ \widehat{av}\ \widehat{qv}\ \widehat{ve}\ \ \hat{\tau}\ \ \hat{b} = \bot.$$

$\widehat{\mathcal{F}}$'s behavior on primops is more interesting. If the argument $\hat{x}$ is being tracked (*i.e.*, $\widehat{qx}$ is a quantity, not bottom), then we intersect $type/int$ with $\widehat{qx}$'s incoming type, passing the result type table $\hat{\tau}_t$ to the true continuation, and we subtract $type/int$ from $\widehat{qx}$'s type in $\hat{\tau}_f$ the table passed to the false continuation. In other words, we do the type recovery of the ETREC semantics, but only for the values being tracked.

$$\widehat{\mathcal{F}}\ [\![p\text{:}\texttt{test-integer}]\!]\ \left\langle \hat{x},\ \hat{k}_0,\ \hat{k}_1\right\rangle\ \left\langle \widehat{qx},\ \widehat{qk}_0,\ \widehat{qk}_1\right\rangle\ \widehat{ve}\ \ \hat{\tau}\ \ \hat{b} =$$
$$\Big(\bigsqcup_{k\in\hat{k}_0} \widehat{\mathcal{F}}\ k\ \ \langle\rangle\ \ \langle\rangle\ \ \widehat{ve}\ \ \hat{\tau}_t\ \ ic_p^0\Big) \sqcup \Big(\bigsqcup_{k\in\hat{k}_1} \widehat{\mathcal{F}}\ k\ \ \langle\rangle\ \ \langle\rangle\ \ \widehat{ve}\ \ \hat{\tau}_f\ \ ic_p^1\Big)$$
$$\begin{aligned}\text{where } \hat{\tau}_f &= \widehat{qx} \neq \bot \ \longrightarrow\ \hat{\tau}\left[\widehat{qx} \mapsto (\hat{\tau}\ \widehat{qx} - type/int)\right]\\ &\qquad \text{otherwise } \hat{\tau}\\ \hat{\tau}_t &= \widehat{qx} \neq \bot \ \longrightarrow\ \hat{\tau}\left[\widehat{qx} \mapsto (\hat{\tau}\ \widehat{qx} \sqcap type/int)\right]\\ &\qquad \text{otherwise } \hat{\tau}\end{aligned}$$

The `+` primop is similar. If the arguments $\hat{x}$ and $\hat{y}$ are being tracked, then we update the type table passed forward, otherwise we simply pass along the incoming type table $\hat{\tau}$ unchanged. Since the continuations k are applied by the approximate $\widehat{\mathcal{F}}$ function, the quantity vector is $\langle\bot\rangle$ — we will not be tracking the variables bound by the call to k.

$$\widehat{\mathcal{F}}\ [\![p\text{:}\texttt{+}]\!]\ \left\langle \hat{x},\ \hat{y},\ \hat{k}\right\rangle\ \left\langle \widehat{qx},\ \widehat{qy},\ \widehat{qk}\right\rangle\ \widehat{ve}\ \ \hat{\tau}\ \ \hat{b} =$$
$$\bigsqcup_{k\in\hat{k}} \widehat{\mathcal{F}}\ k\ \ \langle\emptyset\rangle\ \ \langle\bot\rangle\ \ \widehat{ve}\ \ \hat{\tau}''\ \ ic_p$$
$$\begin{aligned}\text{where } \hat{\tau}' &= \widehat{qx} \neq \bot \ \longrightarrow\ \hat{\tau}\left[\widehat{qx} \mapsto (type/number \sqcap \hat{\tau}\ \ \widehat{qx})\right]\\ &\qquad \text{otherwise } \hat{\tau}\\ \hat{\tau}'' &= \widehat{qy} \neq \bot \ \longrightarrow\ \hat{\tau}'\left[\widehat{qy} \mapsto (type/number \sqcap \hat{\tau}'\ \ \widehat{qy})\right]\\ &\qquad \text{otherwise } \hat{\tau}'\end{aligned}$$

To finish off the Reflow semantics, we must take care of `letrec`. Abstracting $\mathcal{C}$'s definition for `letrec` is simple. Evaluating the `letrec`'s bound expressions only involves closing lambdas, not referencing variables. So the `letrec` will not "touch" any of the variables we are currently tracking. Hence the `letrec` does not make any local contribution to the answer type cache, but simply augments the variable environment $\widehat{ve}$ with the procedure bindings and recursively evaluates the inner call c'.

$$\begin{aligned}\hat{\mathcal{C}}\,[\![c\text{:}(\texttt{letrec}\ ((f_1\ \ l_1)\ldots)\ \ c')]\!]\ \hat{\beta}\ \widehat{ve}\ \hat{\tau} = \hat{\mathcal{C}}\,c'\ \hat{\beta}'\ \widehat{ve}'\ \hat{\tau}&\\ \text{where } \hat{b} = c&\\ \hat{\beta}' = \hat{\beta}\left[c \mapsto \hat{b}\right]&\\ \widehat{ve}' = \widehat{ve} \sqcup \left[\left\langle f_i, \hat{b}\right\rangle \mapsto \hat{\mathcal{A}}_v\, l_i\ \hat{\beta}'\ \widehat{ve}\right]&\end{aligned}$$

One detail of `letrec` that we have neglected is tracking the types of the variables bound by `letrec`. There are several ways to handle this. We could add a case to the Reflow function to handle reflowing from `letrec` expressions, creating a special contour for the `letrec`'s binding. This is a fairly complex and expensive way to handle a simple case. Because `letrec` is syntactically restricted to binding only lambda procedures to its variables, we can statically analyse this case, and simply assign in advance the procedure type to all references to all `letrec` variables. The simplest place to insert this static assignment is in the initial type cache $\hat{\delta}_0$ used in the Reflow iteration:

$$\hat{\delta}_0 = \lambda\left\langle [\![r\text{:}v]\!],\, \hat{b}\right\rangle .\ \begin{array}{llll} \mathit{binder}\, v \in \text{CALL} & \longrightarrow & \mathit{type/proc} & (\texttt{letrec}) \\ \mathit{binder}\, v \in \text{LAM} & \longrightarrow & \bot & \text{(lambda).} \end{array}$$

This performs the type analysis of the `letrec` variables in one step, leaving the rest of the Reflow semantics free to concentrate on the variables bound by lambdas.

9.5 Implementation

I have written a prototype of the type recovery algorithm in T. This implementation uses the same general framework as the 1CFA implementation discussed in section 5.4. The type-recovery code is about 900 lines of heavily commented Scheme. The type lattice includes the symbol, pair, false, procedure, fixnum, bignum, flonum, vector, list, integer, and number types. The implementation is for the most part a straightforward transcription of the approximate type recovery semantics, using the single-threaded time-stamp techniques for the variable environment, store, and quantity environment.

Like the basic 1CFA implementation, I have made little effort to optimise or even compile the implementation — response time in the interpreter is sufficiently quick for small test cases. At this point in my experimentation, there is no reason to believe that efficiency of the analysis will be an overriding issue in practical application of the type recovery algorithm.

A reflow-based analysis is, however, quite a bit slower than a simple 1CFA-based analysis. I have applied the type recovery to the little puzzle example from section 9.4.2, and the `delq` and `fact` procedures of figure 9.1. Running interpreted T on a DECstation 3100 PMAX produced the following times:

puzzle	5.1 sec
`fact`	7.8 sec
`delq`	5.3 sec

As an example of the algorithm's power, it is able to completely recover the types of all variable references in the `delq` and `fact` procedures given in section 9.1. The types recovered for the puzzle and `fact` examples are shown in detail in appendix B.

The allure of type recovery, of course, is type-safe Scheme implementations with little run-time overhead. It remains to be seen whether there is enough recoverable type information in typical code to allow extensive optimisation. The algorithm has not been tested extensively on a large body of real code. However, these early results are encouraging.

9.6 Discussion and Speculation

This section is a collection of small discussions and speculations on various aspects of Scheme type recovery.

9.6.1 Limits of the Type System

From the optimising compiler's point of view, the biggest piece of bad news in the Scheme type system is the presence of arbitrary-precision integers, or "bignums." Scheme's bignums, an elegant convenience from the programmer's perspective, radically interfere with the ability of type recovery to assign small integer "fixnum" types to variable references. The unfortunate fact is that two's complement fixnums are not closed under any of the common arithmetic operations. Clearly adding, subtracting, and multiplying two fixnums can overflow into a bignum. Less obvious is that simple negation can overflow: the most negative fixnum overflows when negated. Because of this, not even fixnum division is safe: dividing the most negative fixnum by -1 negates it, causing overflow into bignums. Thus, the basic fixnum arithmetic operations cannot be safely implemented with their corresponding simple machine operations. This means that most integer quantities cannot be inferred to be fixnums. So, even though type recovery can guarantee that all the generic arithmetic operations in figure 9.1's factorial function are integer operations, this does not buy us a great deal.

Not being able to efficiently implement safe arithmetic operations on fixnums is terrible news for loops, because many loops iterate over integers, particularly array-processing loops. Taking five instructions just to increment a loop counter can drastically affect the execution time of a tight inner loop.

There are a few approaches to this problem:

- **Range analysis**
  Range analysis is a data-flow analysis technique that bounds the values of numeric

variables [Harrison 77]. For example, range analysis can tell us that in the body of the following C loop, the value of `i` must always lie in the range $[0, 10)$:

```
for(i=9; i>=0; i--) printf("%d ", a[i]);
```

Range analysis can probably be applied to most integer loop counters. Consider the `strindex` procedure below:

```
(define (strindex c str)
  (let ((len (string-length str)))
    (letrec ((lp (lambda (i)
                   (cond ((>= i len) -1) ; lose
                         ((char= c (string-ref str i)) i) ; win
                         (else (loop (+ i 1))))))) ; loop
      (lp 0))))
```

Type recovery can guarantee that `len`, being the result of the `string-length` primop, is a fixnum. Range analysis can show that `i` is bounded by 0 and a fixnum; this is enough information to guarantee that `i` is a fixnum. Range analysis is useful in its own right as well — in this example, it allows us to safely open-code the character access `(string-ref str i)` with no bounds check.

- **Abstract safe usage patterns**
  The poor man's alternative to range analysis is to take the usage patterns that are guaranteed to be fixnum specific, and package them up for the user as syntactic or procedural abstractions. These abstractions can be carefully decorated with `proclaim` declarations to force fixnum arithmetic. For example, a loop macro which has a `(for c in-string str)` clause can safely declare the string's index variable as a fixnum. This approach can certainly pick up string and array index variables.

- **Disable bignums**
  Another cheap alternative to range analysis is to live dangerously and provide a compiler switch or declaration which allows the compiler to forget about bignums and assume all integers are fixnums. Throwing out bignums allows simple type recovery to proceed famously, and programs can be successfully optimised (successfully, that is, until some hapless user's program overflows a fixnum...).

- **Hardware support**
  Special tag-checking hardware, such as is provided on the SPARC, Spur and Lisp Machine architectures [Garner$^+$ 88, Hill$^+$ 86, Moon 85], or fine-grained parallelism, such as is provided by VLIW architectures [Ellis 86, Fisher$^+$ 84], allow fixnum arithmetic to be performed in parallel with the bignum/fixnum tag checking. In this case, the limitations of simple type recovery are ameliorated by hardware assistance.

  VLIW's could be ideal target machines for languages that require run-time type-checking. For example, when compiling the code for a safe `car` application, the compiler can pick the trace through the type test that assumes the `car`'s argument is a legitimate pair. This will almost always be correct, the sort of frequency skew that allows trace scheduling to pay off in VLIW's. The actual type check operation can percolate down in the main trace to a convenient point where ALU and branch resources are available; the error-handling code is off the main trace.

Common cases for generic arithmetic operations are similarly amenable to trace picking. The compiler can compile a main fixnum trace (or flonum trace, as the common case may be), handling less frequent cases off-trace. The overhead for bignum, rational, and complex arithmetic ops will dominate the off-trace time in any event, whereas the lightweight fixnum or flonum case will be inlined. This idea can also be extended to other sorts of lightweight traps that higher-order languages typically require, such as bounds checking for inlined heap allocation.

The VLIW trace-scheduling approach to run-time type safety has an interesting comparison to the automatic tag checking performed by Lisp Machines. Essentially, we have taken the tag-checking ALU and branch/trap logic, promoted it to general-purpose status, and exposed it to the compiler. These hardware resources can now be used for non-typechecking purposes when they would otherwise lay idle, providing opportunities for increased fine-grained parallelism.

The Lisp Machine approach is the smart hardware/fast compiler approach; the VLIW approach is the other way 'round.

9.6.2 Declarations

The dangerous `proclaim` declaration is problematic. A purist who wants to provide a guaranteed safe Scheme implementation might wish to ban `proclaim` on the grounds that it allows the user to write dangerous code. A single address-space, personal computer implementation of Scheme, for example, might rely on run-time safety to prevent threads from damaging each other's data. Including `proclaim` would allow the compilation of code that could silently trash the system, or access and modify another thread's data.

On the other hand, safe declarations like `enforce` have limits. Some useful datatypes cannot be checked at run time. For example, while it is possible to test at run time if a datum is a procedure, it is *not* possible, in general, to test at run time if a datum is a procedure that maps floating-point numbers to floating-point numbers. Allowing the user to make such a declaration can speed up some critical inner loops. Consider the floating-point numeric integrator below:

```
(define (integ f x0 x1 n)
  (enforce procedure? f) (enforce fixnum? n)
  (enforce flonum? x0)   (enforce flonum? x1)
  (let ((delta (/ (- x1 x0) n)))
    (do ((i n (- i 1))
         (x x0 (+ x delta))
         (sum 0.0 (+ sum (f x))))
        ((= i 0) (* sum delta)))))
```

In some cases, an analysis might be able to find all applications of the integrator, and thus discover that `f` is always bound to a floating-point function. However, if the integrator is a top level procedure in an open system, we can't guarantee at compile time that `f` is a floating-point function, and we can't check it at run time. This means that the sum operation must check the return value of `f` each time through the loop to ensure it is a legitimate floating-point value.

While the `proclaim` declaration does allow the user to write dangerous code, it is at least a reasonably principled loophole. `Proclaim` red-flags dangerous assumptions. If the user can be guaranteed that only code marked with `proclaim` declarations can behave in undefined ways, debugging broken programs becomes much easier.

Finally, it might be worth considering a third declaration, `probably`. The declaration `(probably flonum? x)` is a hint to the compiler that `x` is most likely a floating-point value, but could in fact be any type. Having a `probably` declaration can allow trace-scheduling compilers to pick good traces or optimistically open-code common cases.

9.6.3 Test Hoisting

Having one branch of a conditional test be the undefined effect `$` or `error` primop opens up interesting code motion possibilities. Let us call tests with an `(error)` arm "error tests," and tests with a `($)` arm "$-tests." These tests can be hoisted to earlier points in the code that are guaranteed to lead to the test. For example,

```
(block (print (+ x 3))
       (if (fixnum? x) (g x) ($)))
```

can be legitimately transformed to

```
(if (fixnum? x) (block (print (+ x 3)) (g x))
    ($))
```

because the compiler can choose any effect it likes for the undefined effect operator, including the effect of `(print (+ x 3))`. This can be useful, because hoisting type tests increases their coverage. In the example above, hoisting the `fixnum?` test allows the compiler to assume that `x` is a fixnum in the `(+ x 3)` code.

Further, error and $-tests can be hoisted above code splits if the test is applied in both arms. For example, the type tests in

```
(if (> x 0)
    (if (pair? y) (bar) ($))
    (if (pair? y) (baz) ($)))
```

can be hoisted to a single type test:

```
(if (pair? y)
    (if (> x 0) (bar) (baz))
    ($))
```

Real savings accrue if loop invariant type tests get hoisted out of loops. For example, in a naive, declaration-free dot-product subroutine,

```
(define (dot-prod v w len)
  (do ((i (- len 1) (- i 1))
       (sum 0.0 (+ sum (* (vector-ref v i) (vector-ref w i)))))
      ((< i 0) sum)))
```

each time through the inner loop we must check that `v` and `w` have type vector. Since `v` and `w` are loop invariants, we could hoist the run-time type checks out of the loop, which would speed it up considerably. (In this particular example, we would have to duplicate the termination test `(< i 0)`, so that loop invariant code pulled out of the loop would only execute if the loop was guaranteed at least one iteration. This is a standard optimising compiler technique.)

Hoisting error tests requires us to broaden our semantics to allow for early detection of run-time errors. If execution from a particular control point is guaranteed to lead to a subsequent error test, it must be allowed to perform the error test at the control point instead.

In the general case, error and `$`-test hoisting is a variant of very-busy expression analysis [Aho+ 86]. Note that hoisting `$`-tests gives a similar effect to the backwards type inferencing of Kaplan and Ullman [Kaplan+ 80]. Finding algorithms to perform this hoisting is an open research problem.

Note that the sort of "dangerous" compilation permitted by the undefined effect, and its formal description via the `($)` operator is of dubious semantic merit. However, implementors persist in making dangerous implementations, so program analyses need to support them.

9.6.4 Other Applications

The general approach to solving quantity-based analyses presented in this chapter can be applied to other data-flow problems in higher-order languages. The range analysis discussed in subsection 9.6.1 is a possible candidate for this type of analysis.

One analysis very similar to type recovery is future analysis. Some parallel dialects of Scheme [Kranz+ 89] provide *futures*, a mechanism for introducing parallelism into a program. When the form `(future <exp>)` is evaluated, a task is spawned to evaluate the expression `<exp>`. The `future` form itself immediately returns a special value, called a *future*. This future can be passed around the program, and stored into and retrieved from data structures until its actual value is finally required by a "strict" operator such as `+` or `car`. If the future's task has completed before the value is needed, the strict operation proceeds without delay; if not, the strict operator must wait for the future's task to run to completion and return a value.

Futures have a heavy implementation expense on conventional hardware, because all strict operators must check their operands. Future checking can add a 100% overhead to the serial run time of an algorithm on conventional hardware.

"Future analysis" is simply realising that after a variable has been used as an argument to a strict operator, it's value is guaranteed to be a non-future. So following references to the variable don't have to perform the future check. Thus, in the lambda

```
(λ (x) (print (car x)) ... (f (cdr x)) ...)
```

the `(cdr x)` operation can be compiled without future checking. Clearly, this is identical to the type-recovery analysis presented in this chapter, and the same techniques apply.

9.7 Related Work

Steenkiste's dissertation [Steenkiste 87] gives some idea of the potential gains type recovery can provide. For his thesis, Steenkiste ported the PSL LISP compiler to the Stanford MIPS-X processor. He implemented two backends for the compiler. The "careful" backend did full run-time type checking on all primitive operations, including `car`'s, `cdr`'s, vector references, and arithmetic operations. The "reckless" backend did no run-time type checking at all. Steenkiste compiled about 11,500 lines of LISP code with the two backends, and compared the run times of the resulting executables. Full type checking added about 25% to the execution time of the program.

Clearly, the code produced by a careful backend optimised with type-recovery analysis will run somewhere between the two extremes measured by Steenkiste. This indicates that the payoff of compile-time optimisation is bounded by the 25% that Steenkiste measured. Steenkiste's data, however, must be taken only as a rough indicator. In LISP systems, the tiny details of processor architecture, compiler technology, data representations and program application all interact in strong ways to affect final measurements. Some examples of the particulars affecting his measurements are: his LISP system used high bits for type tags; the MIPS-X did not allow `car` and `cdr` operations to use aligned-address exceptions to detect type errors; his 25% measurement did not include time spent in the type dispatch of generic arithmetic operations; his generic arithmetic was tuned for the small integer case; none of his benchmarks were floating-point intensive applications; his measurements assumed interprocedural register allocation, a powerful compiler technology still not yet in common practice in LISP and Scheme implementations; and LISP requires procedural data to be called with the `funcall` primop, so simple calls can be checked at link time to ensure they are to legitimate procedures.

These particulars of language, hardware architecture, implementation, and program can bias Steenkiste's 25% in both directions. Steenkiste gives a careful description of most of these issues. However, even taken as a rough measurement, Steenkiste's data do indicate that unoptimised type-checking is a significant component of program execution time, and that there is room for compile-time optimisation to provide real speed-up.

The idea of type recovery for Scheme is not new. Vegdahl and Pleban [Vegdahl+ 89] discuss the possibility of "tracking" types through conditionals, although this was never pursued. The ORBIT compiler is able to track the information determined by conditional branches, thus eliminating redundant tests. ORBIT, however, can only recover this information over trees of conditional tests; more complex control and environment structures, such as loops, recursions, and joins block the analysis.

Curtis discusses a framework for performing static type inference in a Scheme variant [Curtis 90], along the lines of that done for statically typed polymorphic languages such as ML [Milner 85] or LEAP [Pfenning+ 90]. However, his work assumes that most "reasonable" Scheme programs use variables in a way that is consistent with a static typing discipline. In essence, Curtis' technique types variables, whereas the analysis developed in this chapter types variable references, an important distinction for Scheme. Note that without introducing type-conditional primitives that bind variables, and (perhaps automatically) rewriting Scheme code to employ these primitives, this approach cannot recover the information determined by conditional branches on type tests, an important source of type information.

Traditional data-flow analysis techniques have also been used to perform type recovery for latently-typed variants of traditional imperative languages [Tenenbaum 74, Kaplan+ 80][Aho+ 86, chapter 10]. These approaches differ from the technique developed in this chapter in the following ways:

- First, they focus on side-effects as the principal way values are associated with variables. In Scheme, variable binding is the predominant mechanism for associating values with variables, so the Scheme type recovery analysis must focus on variable binding.

- Second, they assume a fixed control-flow graph. Higher-order languages require the control-flow analysis technology developed in this dissertation.

- Third, they assume a single, flat environment. Scheme forces one to consider multiple bindings of the same variable. The reflow semantics of section 9.4 correctly handles this complexity.

- Finally, they are not semantically based. The type recovery analysis in this chapter is based on the method of non-standard abstract semantic interpretations. This establishes a formal connection between the analysis and the base language semantics. Grounding the analysis in denotational semantics allows the possibility of proving various useful properties of the analysis.

These differences are all connected by the centrality of lambda in Scheme. The prevalence of lambda is what causes the high frequency of variable binding. Lambda allows the construction of procedural data, which in turn prevent the straightforward construction of a compile-time control-flow graph. Lambda allows closures to be constructed, which in turn provide for multiple extant bindings of the same variable. And, of course, the mathematical simplicity and power of lambda makes it much easier to construct semantically-based program analyses.

Chapter 10

Applications III: Super-β

If our behavior is strict, we do not need fun!
— Zippy the Pinhead,
on normal-order optimisations.

10.1 Copy Propagation

Copy propagation is a standard code improvement technique based on data-flow analysis [Aho+ 86, chapter 10]. Copy propagation essentially involves removing variables that are redundant copies of other variables. That is, if the only definitions of `x` reaching the expression `x+3` are of the form `x := y`, then we can substitute `y` for its copy `x`: `y+3`. If we can do this with all the references to `x`, then we can remove the assignment to `x` as well, completely eliminating the variable from the program.

In Scheme, copy propagation is handled in some cases by simple β-substitution. We can substitute `y` for `x` in:

```
(let ((x y))
  (foo x))        ⇒     (foo y)
```

However, β-substitution does not handle cases involving circular or join dependencies. For example, consider the loop which multiplies each element in `vec` by `x`:

```
(letrec ((lp (λ (y i)
               (cond ((>= i 0)
                      (set (vref vec i) (* (vref vec i) y))
                      (lp y (- i 1)))))))
  (lp x 20))
```

Clearly, references to `y` can be replaced by `x`, although simple β-substitution will not pick this up.

Note that this sort of copy propagation can frequently and usefully be applied to the continuation variable in the CPS expansions of loops. An example is the summation loop from the end of chapter 3. Partially CPS-converted, that loop looks like this:

```
(λ (n k)
  (letrec ((lp (λ (i sum c)
                 (if (zero? i) (c sum)
                     (lp (- i 1) (+ sum i) c)))))
    (lp n 0 k)))
```

The loop's continuation `c` is always bound to the procedure's continuation `k`, and is redundantly passed around the loop until it is invoked on loop termination. Applying copy propagation (and one round of useless-variable elimination to clean up afterwards) removes the iteration variable `c` entirely:

```
(λ (n k)
  (letrec ((lp (λ (i sum)
                 (if (zero? i) (k sum)
                     (lp (- i 1) (+ sum i))))))
    (lp n 0)))
```

This sort of situation crops up frequently in CPS-expanded loops.

The general form of Scheme copy propagation, then, is:

> For a given variable v, find all lexically inferior variables that are always bound to the value of v, and replace their references with references to v.

A variable is lexically inferior to v if its binding lambda occurs within the scope of v's binding lambda. With this definition, Scheme copy propagation acts as a sort of "super-β" substitution rule. (If we allow side-effects to variables, this can disallow some possible substitutions. If, on the other hand, we use an intermediate representation that has no variable side-effects, we can substitute at will. This is one of the attractions of removing variable assignments from the intermediate representation with assignment conversion. ML elegantly avoids the issue by disallowing assignments to variables altogether.)

Note that the environment problem rears its head here. For example, we cannot substitute `x` for `y` in the following code:

```
(let ((f (λ (x h) (if (null? h) (λ () x)
                      (let ((y (h))) ... y ...)))))
  (f 3 (f 5 nil)))
```

Even though `y` is certainly bound to `x` (because `h` is bound to `(λ () x)` at that point), it's the wrong binding of `x` — we need the one that is in the `y` reference's context. We have multiple bindings of the same variable around at the same time, and so we have to be careful when we are doing copy propagation to keep these multiple environments separate.

10.1.1 Copy Propagation and Reflow Analysis

We can use reflow analysis to handle the environment problem raised by copy propagation.

Suppose we have some lambda ℓ = `(λ (x y z) ...)`, and we wish to perform copy propagation on `x`. That is, we want to find the lexically inferior variables that are always bound to the (lexically apparent) value of `x` so we can replace them with `x`. We do a reflow, starting with each entry to ℓ. As we enter ℓ, we create a special contour for ℓ's variables. All the lexically inferior lambdas in this scope will be closed over ℓ's special contour, and so won't get their references to `x` confused with other bindings of `x`.

In order to perform the analysis, we need an abstract value domain of two values: a value to represent whatever is bound to the binding of `x` that we're tracking, and a value to represent everything else in the system. Call `x`'s value 0 and the other value 1. The initial system state when we begin a reflow by entering ℓ is that

- the current binding of `x` has value 0
- all other bindings in the environment — including the current bindings of `y` and `z` and other extant bindings of `x` — and all values in the abstract store have value 1.

As we proceed through the abstract interpretation, whenever we enter a lambda ℓ' that is lexically inferior to ℓ, we check to see if ℓ' is closed over the special ℓ contour we're tracking. If it is, we save away the value sets passed in as arguments to the variables of ℓ'.

After we've reflowed ℓ from all of its call sites, we examine each lexically inferior variable. Consider the value set accumulated for variable v. If it is $\emptyset$, then v's lambda was never called — it's dead code. If it is $\{0\}$, then it is always bound to the value of `x` in its lexical context, and so is a candidate for copy propagation — we can legitimately replace all references to v with `x`. If it is $\{0, 1\}$ or $\{1\}$, then it can potentially be bound to another value, and so is not a candidate for copy propagation.

Notice the conservative nature of the analysis: the presence of 1 in a value set marks a variable as a non-copy. This is because after setting up the initial special contour, all the other variable bindings to value sets happen in approximate contours. This means that different bindings can get mapped together. If our analysis arranges for this to affect the analysis in the safe direction, this identification is OK. Ruling out a possible copy is erring on the safe side, and that is the effect we get by unioning together the value sets and looking for 1's .

10.2 Lambda Propagation

In the CPS Scheme intermediate representation, a call argument can be a variable, a constant, or a lambda. Constant propagation (from section 7.3) and copy propagation are essentially just finding out when a variable is bound to a known variable or constant, and doing the appropriate substitution. Clearly there's one important case left to be done: binding a variable to a known lambda, which we can call "lambda propagation."

With lambda propagation, the environment problem returns in force. Like variables, lambda expressions are not context-independent. A procedure is a lambda plus its environment. When we move a lambda from one control point to another, in order for it to evaluate to the same procedure we must be sure that environment contexts at both points are equivalent. This is reflected by the "environment consonance" condition in the following statement of lambda propagation:

> For a given lambda l, find all the places it is used in contexts that are environmentally consonant with its point of closure.

By "environmentally consonant," I mean that all the lambda's free variables are lexically apparent at the point of use, and are also bound to the same values as they are at the lambda's point of closure. That is, if we determine that lambda ℓ = `(λ (x) (+ x y))` is called from call c = `(g z)`, we must also determine that c appears in the scope of `y`, and further, that it is the same binding of `y` that pertained when we closed ℓ. In this sense, the more free variables a lambda has, the more constraints there are on it being "environmentally consonant" with other control points. At one extreme, combinators — lambdas with no free variables — are environmentally consonant with any call site in the program, and can be substituted at any point, completely unconstrained by the lexical context.

10.2.1 Simplifying the Consonance Constraint

Suppose we want to find all the places in a program we can substitute lambda expression

```
ℓ:(λ (x) ... y ... z ... w)
```

The lambda expression ℓ is closed over three free variables: `y`, `z`, and `w`. Let us suppose that `y` is the innermost free variable, that is, ℓ's lexical context looks something like this:

```
(λ (w)
  ...
  (λ (z)
    ...
    (λ (y)
      ...
      (λ (a)
        ...
        ℓ:(λ (x) ... y ... z ... w)
        ...)
      ...)
    ...)
  ...)
```

We can substitute ℓ only within the scope of `y`. (Notice that since `a` is not free in ℓ, occurrences of ℓ can migrate outside of its scope.) The key to ensuring environment consonance is to realise that if we ensure that ℓ is consonant with its innermost free variable — `y` — then it is consonant with the rest of its free variables. For example, if we call ℓ

from a call context that binds y in the same run-time contour as ℓ is closed over, then we can be sure the same goes for z and w.

Given this observation, we can see that a lambda propagation analysis really only needs to be careful about a single contour: the one over the innermost free variable's lambda. In a reflow-based analysis procedure, this means we need to track only one special contour at a time.

10.2.2 Lambda Propagation and Reflow Analysis

Given a lambda μ, we will do lambda propagation for all inferior lambdas ℓ whose innermost free variable is bound by μ. We perform a reflow from all calls to an abstract closure over μ. As we enter μ beginning the reflow, we allocate a special contour for μ's variables. As we proceed through the abstract interpretation, we will make closures, some over the special contour that we're tracking. Just as in copy propagation, when we enter a lambda μ' that is lexically inferior to μ, we check to see if μ' is closed over the special μ contour. If it is, we save away the value sets passed in as arguments to the variables of μ'.

When we're done with our reflows, we examine each variable v that is lexically inferior to μ. Consider the value set accumulated for v. If (1) it consists entirely of closures over a single lambda ℓ, (2) ℓ is lexically inferior to μ, (3) ℓ's innermost free variable is bound by μ, and (4) each closure over ℓ has the special contour for μ's scope, then we succeed: ℓ can be substituted for any reference $r{:}v$ to v — the reference is guaranteed to evaluate to a closure over ℓ and the environment that pertains at $r{:}v$ is consonant with the one over which ℓ is closed.

10.2.3 Constraints on Lambda Substitution

Once we've found the variables for which we can substitute a given lambda, we must decide which cases are appropriate. If a lambda can be substituted at multiple points in a program, performing all the substitutions can lead to code blow-up. We might wish to limit our substitutions to cases where there is only one possible target, or the lambda is below a certain size. We might choose some subset of the references to substitute for, where the compiler thinks there is payoff in expanding out the lambda, leaving other references unchanged.

If our language definition requires multiple closures over identical lambdas to be non-eq, then we have to be careful when substituting a lambda for multiple variable references. If more than one of the references is a call argument (instead of a call procedure), substitution will break the eqness of procedures. With such a language definition, we could not, for example, substitute for x in

```
(let ((x (λ () #f))) (eq? x x))
```

Duplicating the lambda would cause the eq? test to return false. ML avoids this problem by disallowing eq tests on procedures; the Scheme report is less clear on this issue.

One case definitely worth treating with caution is when a lambda is used in its own body, *e.g.*:

```
(letrec ((fact (λ (n)
                 (if (= n 0) 1
                     (* n (fact (- n 1)))))))
  (fact m))
```

Lambda propagation tells us we can legitimately substitute the lambda definition of `fact` for the variable reference `fact` in its own body, giving us (after one round of substitution):

```
(letrec ((fact (λ (n)
                 (if (= n 0) 1
                     (* n ((λ (n) (if (= n 0) 1
                                      (* n (fact (- n 1)))))
                           (- n 1)))))))
  (fact m))
```

Clearly, blindly performing this substitution can get us into an infinite regress. This sort of situation can happen without `letrec`, in cases involving self-application

```
(let ((lp0 (λ (lp1) (lp1 lp1)))) (lp0 lp0))
```

or indirection through the store:

```
(let* ((pair (cons #f #f))
       (lp (λ () ((car pair)))))
  (set-car! pair lp)
  (lp))
```

Substituting a lambda for a variable inside the lambda's body is not necessarily a bad thing to do. It is a generalisation of loop unrolling, which can be desirable in some contexts. But the code optimiser must be aware of this case, and avoid it when it is undesirable.

Most of these substitution constraints also pertain to code improvers that use simple β-substitution; detailed discussions are found in the Rabbit, Orbit and Transformational Compiler theses [Steele 78, Kranz$^+$ 86, Kelsey 89]. Note that with the exception of procedural eqness, these caveats are all issues of implementation, not semantics (code blow-up is not, for example, a semantic concern). This casts some suspicion on the procedural eqness requirement.

A final note on lambda propagation. As we've just seen, there are many constraints determining when we can substitute a lambda for a variable known to be bound to a closure over that lambda: the variable's reference context must be environmentally consonant with the lambda's point of closure; the variable must be referenced only once (to avoid code blow-up); and so forth. However, even when these constraints are not satisfied, it is useful to know that a variable reference must evaluate to a closure over a particular lambda. The compiler can compile a procedure call as a simple branch to a known location instead of a jump to a run-time value which must be loaded into a register. If all calls to a lambda are known calls, the compiler can factor the lambda out of the run-time representation of the closure, allowing for smaller, more efficient representations. This is exactly what Orbit's strategy analysis does, for the simple cases it can spot.

10.3 Cleaning Up

When we substitute an expression — a variable, a constant or a lambda — for all references to some variable v, then v becomes a useless variable. It is therefore a candidate for being removed from its lambda's parameter list. This transformation is the one used by the second phase of useless-variable elimination (section 7.2). The following sequence of transformations shows the role of UVE in post-substitution cleanup.

```
;;; Original loop
(letrec ((lp (λ (sum n c)
               (if (= 0 n) (c sum)
                   (lp (+ sum n) (- n 1) c)))))
  (lp 0 m k))

;;; After copy propagation
(letrec ((lp (λ (sum n c)
               (if (= n 0) (k sum)
                   (lp (+ sum n) (- n 1) k)))))
  (lp 0 m k))

;;; After UVE
(letrec ((lp (λ (sum n)
               (if (= n 0) (k sum)
                   (lp (+ sum n) (- n 1))))))
  (lp 0 m))
```

10.4 A Unified View

Copy and constant propagation are classical code improvements based on data-flow analysis. Standard imperative languages typically use side-effects to associate variables with values, and this is what the standard formulations of copy and constant propagation focus on. In Scheme, however, variables are typically associated with values by binding them. With this change of emphasis, copy and constant propagation become simply special cases of an interesting generalisation of the λ-calculus β-substitution rule. This is particularly true in the CPS Scheme internal representation, where side-effects are not allowed to interfere with the substitution rules. This "super-β" model of code improvement gives us one more alternative to copy and constant propagation: "lambda propagation" — substituting lambda expressions for variable references.

At this point, we can see that constant, copy and lambda propagation are actually three facets of the same basic problem: substituting known expressions for the variables bound to them in the CPS intermediate representation.

In the case of copy and lambda propagation, the context-dependence of variable references and lambda expressions requires our analysis to be sensitive to the environment that exists at the source and target points of the expression being substituted. This is the environment problem introduced in chapter 8; its solution is based on the general technique of reflow analysis.

Chapter 11

Future Work

Dissertations are not finished;
they are abandoned.
— H. Q. Bovik

The research I've reported on in this dissertation opens up many possible avenues for future work. In this chapter, I will sketch what I think are some of the more interesting ones.

11.1 Theory

Classical research in optimising compilers has had little use for formal semantics. The great allure of non-standard abstract semantic interpretation is that it can bridge these two areas. It provides a framework for compile-time analysis that is grounded in denotational semantics. This raises the possibility of proving that the optimising transformations implemented by a compiler are denotationally invariant: mathematically showing they leave the meaning of a program unchanged. Anyone who has ever been tripped up by a buggy optimising compiler will recognise the attraction of a formally verified optimiser.

Pleban's dissertation [Pleban 81] clearly sets forth the NSAS manifesto: to unify the fields of language semantics and compile-time program analysis under the formalisms of denotational semantics. Although concerned specifically with the problem of storage analysis ("compile-time garbage collection"), Pleban calls for a comprehensive program to express the corpus of traditional compiler optimisation techniques in terms of language semantics. His dissertation does not, however, deliver on this vision.

Neither does mine. The mathematics in this dissertation is concerned mainly with the foundations of control-flow analysis. I have not done any theoretical work on the optimisations that are based on control-flow analysis. It is still an open problem to show, for example, that the induction-variable elimination analysis and transformation of chapter 7 preserves the meaning of the program. In the applications chapters (7, 9, and 10), I have relied on the reader's intuition and informal arguments of correctness.

I suspect that this sort of work needs tools easier to manipulate than Scott-Strachey denotational semantics. The classical framework is unwieldy to use as a compiler tool, a shortcoming that has been noticed by other researchers [Lee 89]. An approach based on higher-level, modular semantics might be better for this kind of "denotational engineering." Oftentimes we wish to manipulate structures much simpler than the ones defined in my semantics. Modular approaches to semantic description [Mosses$^+$ 86] might provide a way to abstract away from the complexity inherent in denotational models. Still, the firm mathematical underpinnings provided by denotational semantics are likely to be important in any future approac to reasoning about optimising compilers.

Another theoretical lacuna in this dissertation concerns reflow analysis. A careful, rigorous development of reflow analysis, complete with precise statements about its properties and supported by proven theorems is beyond the scope of this dissertation. Although it's not essential to my thesis, which centers on control-flow analysis in a CPS representation, I felt that the basic idea of reflow analysis and the class of optimisations it enables warranted inclusion in this dissertation. In a λ-calculus setting, reflow analysis is the natural extension of control-flow analysis to handle environment information. A full exploration of reflow analysis and its capabilities, however, lies ahead.

Finally, a complexity analysis of the control-flow algorithms should be done. Since the complexity of the algorithms depends on which abstraction is chosen, this is probably best deferred until more experience is gained in which abstractions are best. A complexity analysis is not necessary to establish my thesis, which is that control-flow analysis is feasible and useful for higher-order languages. Empirical tests show that the analyses are acceptably quick for small procedures. I have no reason to believe that the analyses will not scale reasonably. Even if this turned out not to be the case, these analyses are not essential for compilation — they are for *optimisation*. During development, when a program is being frequently recompiled, the programmer can compile with optimisation switched off (or, better still for purposes of source-level debugging and rapid turnaround, avoid the compiler entirely, and run in the interpreter). When the code has been debugged and is ready for release, then it can be recompiled once with the optimisations enabled. So even if control-flow analysis turns out to be computationally expensive in production compilers, it will still be useful.

11.2 Other Optimisations

More of the classical data-flow optimisations should be mapped over to the CPS/NSAS framework. The handful of optimisations developed in this dissertation do not even begin to exhaust the possibilities for optimisation applications. The entire corpus of data-flow optimisations is available for consideration. With the basic control-flow structure in hand, the challenge is to try and bring over the entire list of optimisations from figure 1.1.

I believe that a good high-level goal for compiler research in higher-order languages is achieving parity with good FORTRAN and C compilers in compiling scientific code.

Scientific code is a good target because it requires the full array of data-flow optimisations and because it is a compute-intensive domain that requires extensive optimisation.

I also believe that higher-order languages are well suited to scientific code. Sussman and his research group at MIT have done work exploring the advantages of Scheme for programming numerically-intensive codes [Halfant+ 88]. Berlin has demonstrated the feasibility of compiling a restricted class of scientific applications written in Scheme to a horizontal architecture [Berlin 89]. General applicability, however, remains open.

11.2.1 Representation Analysis

One penalty paid by higher-order languages is the cost incurred by uniform data representations. Many higher-order languages allow polymorphic procedures. For example, both Scheme's latent type system and ML's static type system allow polymorphic procedures. This means that values bound to polymorphic variables must have a representation of uniform size. If we don't know whether a value will be a character or a double-precision floating-point value, for example, we don't know whether we should allocate one or two 32-bit registers to contain it, or how many bytes in a vector to reserve for its storage. The solution invariably taken on standard 32-bit architectures is to mandate that all values be represented with exactly 32 bits. Data types that need less space (*e.g.*, characters) are padded; data types requiring more space (*e.g.*, double-precision floating point numbers or cons cells) are represented by a 32-bit pointer to a block of storage in the heap containing the actual value. The 32-bit datum can be reliably passed around in a single register, stored into 4 bytes of memory, and so forth.

Uniform representations incur significant space and time overhead. They are particularly disastrous for floating-point numbers, since the uniform "boxed" representation requires a memory load for every operation on a number, and a heap allocation and store for every number produced by an arithmetic operation. So, instead of floating-point addition taking a single instruction, it requires two loads, an addition, a heap allocation, and a store. This is unacceptable for scientific code.

Analysis to determine when specialised, efficient representations can be used is called "representation analysis." The ORBIT Scheme compiler performed very simple representation analysis in cases where the user allowed it to make "dangerous" assumptions (*i.e.*, disabling the normal run-time type checks). Leroy has also done work layering representation analysis on the ML type system [Leroy 89]. Better information should be produced, however, by an approach that uses data-flow analysis of the sort that this dissertation makes possible.

11.2.2 Register Allocation

Control-flow analysis could be used to implement an extension of the trace-scheduling on-the-fly register allocator used in ORBIT. In rough terms, this allocator moves values between registers and memory as the compiler processes code along an execution trace in much the same way that virtual-memory systems move pages of data between primary memory and disk. However, the compiler's register allocator has an advantage over the operating system's VM pager: it can "look into the future" by scanning forward through the control-flow graph. This means we can use the optimal MIN algorithm for doing register spills: when we have to spill a register, we can spill the one whose value will be used the furthest away in time. ORBIT's register allocator does not fully exploit this model, and in any event does not have the detailed control-flow information for the MIN lookahead that control-flow analysis provides.

Other global register allocators, such as the various techniques based on graph-coloring [Chaitin 82, Chaitin+ 81, Chow+ 84], can also be considered for CPS-based compilers now that control-flow information is available.

11.2.3 Compiling for VLIW Architectures

Another interesting application for control-flow analysis is trace-scheduling compilation for superscalar and VLIW architectures [Ellis 86]. This again depends entirely on having detailed control-flow information. For reasons I outline in section 9.6, latently-typed languages with generic arithmetic operators, like Scheme, are prime candidates for trace-scheduling compilation and VLIW execution.

11.3 Extensions

The basic control-flow analysis could be extended in several interesting ways.

11.3.1 Better Store Models

The analyses I have developed in this dissertation use a minimal store abstraction. Investigating more detailed side-effects models in this framework is an open area. Many vectorising supercomputer compilers, for example, expend a tremendous amount of effort to disambiguate vector and pointer references, removing spurious data dependencies and so allowing loops to be parallelised [Ellis 86, chapter 5]. This sort of effects or dependency analysis is an alternative to schemes that move the effects analysis into the type system, a technique introduced by Reynolds [Reynolds 78] and explored further by Lucassen, Gifford and other designers of the FX language [Lucassen+ 88]. Effects analysis can be useful where a type system breaks down (such as between multiple references to a single vector), or if the notational load of an effects-based type system is cumbersome.

A second alternative to compiler analysis for disambiguating vector references is a linguistic approach: to design a language with high-level vector operations (such as Blelloch and Sabot's scan-vector model [Blelloch+ 90, Blelloch 90]) that eliminate the loop and the references to the individual elements entirely. This sort of high-level notation is better for both the programmer and the compiler. The programmer can express himself more succinctly and is freed of the error-prone burden of spelling out the details of the iteration.

The compiler is saved the trouble of inferring the aggregate, parallel operation from the serial iteration over the individual elements.

If you are considering applying control-flow analysis to effects analysis, you should first consider the alternative schemes of an effects-based type system or a primitive vector algebra. Research in higher-order languages does not suffer the "dusty deck" constraints of FORTRAN compiler research — there is much less investment in code written in Scheme or ML than in FORTRAN. So it's much more feasible to consider approaches that alter the actual language itself.

There are other areas where a detailed store analysis could prove fruitful. Many analyses have been developed that attempt to determine when run-time data structures can be allocated on the stack instead of the heap, or can be explicitly freed at certain points in the code [Hudak 86b, Deutsch 90]. A compiler that uses a CPS intermediate representation might want to adapt these analyses to the framework I've developed in this dissertation.

11.3.2 Modular Analysis

For reasons of computational expense and separate compilation, it is worth considering how to break up the analysis of a large program into separate blocks, where a block might be the collection of procedures that comprises a single module or library of a program, or, at finer granularity, an individual top-level procedure. This gives us a hierarchical analysis: analysing a block in detail produces summary information on the behavior of the block, which is then used when analysing the other blocks that are clients of the first block.

For example, the summary information describing the control flow of a particular top-level lambda `(λ (x y k) ...)` might be that its continuation argument `k` never escapes, but is always called; or that its first argument `x` escapes into the store; or that when control is passed to the lambda, it subsequently branches to the external lambda. These are the sort of actions that affect the control flow of this lambda's clients, so they must be represented in the summary description of the lambda.

A summary of the control-flow behavior of a program module or shared library could vastly improve the quality of the analysis of that module's clients. This is worth pursuing for a serious implementation of control-flow analysis in a production compiler.

11.3.3 Tuning the Contour Abstraction

The choice of contour abstraction determines the accuracy and expense of the corresponding control-flow analysis. Choosing a good abstraction that usefully distinguishes classes of exact contours with minimal cost is an engineering issue that needs to be investigated. It's quite possible to have a mixed abstraction, where features of the program determine whether a precise, expensive abstraction will be used for a given contour, or an approximate, cheap one will be used instead. In fact, we could employ a kind of "iterative deepening" strategy, where the results of a very cheap, very approximate analysis would determine the abstractions used at different points in a second pass, providing precise abstractions only at the places they are required.

One source of useful abstractions to consider is Deutsch's work on abstract semantics for λ-calculus semantic descriptions [Deutsch 90]. See also the generalisations of 1CFA discussed on page 55.

11.3.4 Basic Blocks

A basic block is a block of code such that control can only enter at the top of the block, and can only exit at the bottom — there are no jumps into or out from the middle of the block. Traditional flow-analysis algorithms consider basic blocks as single vertices in their control-flow graphs, which saves a lot of time when flowing information around the graph.

In Scheme, a piece of code like

```
(let* ((desc-root (sqrt (- (* b b) (* 4 a c))))
       (denom (* 2 a))
       (neg-b (- b))
       (soln1 (/ (+ neg-b desc-root) denom))
       (soln2 (/ (- neg-b desc-root) denom)))
  (cons soln1 soln2))
```

is straight-line code, and for efficiency purposes, could be considered a basic block in flow-analysis code. The analysis I've presented in this dissertation, however, separates this block of code into its separate primitive operations, and then serially flows information through the block.

A simple definition of a basic block in CPS is a chain of lambda expressions, such that each lambda's internal call is to a (non-conditional) primop, and the continuation given to the primop is the next lambda in the chain, *e.g.*:

```
(λ (k) (+ a b (λ (s) (- x y (λ (d) (* s d k))))))
```

It should be fairly straightforward to collapse these chains of lambdas into single nodes for the purposes of flow analysis. This should speed the analysis up considerably.

In this approach, the contours allocated for the internal lambdas in a basic block will be, in effect, dependent values on the contour allocated for the initial lambda of the block. By eliminating the need to generate all these intermediate contours, we can use more expensive contour abstractions for the ones that remain.

This kind of optimisation eliminates time spent on trivial control-flow paths, but still retains the semantic, λ-calculus foundation of the CPS-based analysis. In effect, what we are doing here is applying a very cheap control-flow analysis as a pre-pass to eliminate the trivial cases before applying the fully general analysis.

11.3.5 Identifying Equivalent Procedures

Consider two different abstract closures $\left\langle \ell,\ \widehat{\beta}_1 \right\rangle$ and $\left\langle \ell,\ \widehat{\beta}_2 \right\rangle$ over a single lambda, where

$$\begin{aligned} \widehat{\beta}_1 &= [\ell_{17} \mapsto 3,\ \ell_2 \mapsto 1,\ \ell_8 \mapsto 33] \\ \widehat{\beta}_2 &= [\ell_{17} \mapsto 3,\ \ell_2 \mapsto 5,\ \ell_8 \mapsto 33]. \end{aligned}$$

These two contour environments differ only in the contour assigned to ℓ_2. If ℓ contains no references to variables from ℓ_2, this difference is irrelevant, and the two closures can be considered equivalent. As a limit case, all closures over a combinator — a lambda with no references to free variables — are equivalent. Identifying these closures together leads to faster convergence of the control-flow analysis, giving an improvement in speed.

11.4 Implementation

The implementation I have used to support my thesis is very much a toy implementation. It has been applied to, at most, a few hundred lines of code. It has not been interfaced to a back-end that produces machine code.

An obvious next step is to scale up the implementation and try the analyses and optimisations on a large body of code. If the system were interfaced to a compiler back-end, the optimised code could be timed, which would give a way to measure the effectiveness of the optimisations.

Scaling the system will require some work to write an efficient implementation of the analysis algorithms. My current, research implementation uses very simple data structures and algorithms. For example, sets are represented with linked lists instead of bit vectors, which slows down membership tests and set operations. Contour environments are also represented with linked lists instead of byte vectors, which slows down contour lookups. Comparing two contour environments could be sped up as well by storing a hash value with the environment. Similarly, the variable environment is not tuned for lookups. Besides these sorts of efficiency considerations, one might also want to incorporate some of the extensions discussed in the previous section.

Chapter 12

Related Work

The fields of abstract semantic interpretation, higher-order language optimisation, and data-flow analysis are far too large to cover comprehensively. In this chapter I will cover work by others that is either fundamental or directly relevant to my research. I will focus on NSAS-based analyses. Discussions of work that is relevant to the particular optimisations I've presented in this dissertation are found in the associated chapters.

12.1 Early Abstract Interpretation: the Cousots and Mycroft

The seminal work on non-standard abstract interpretation is by the Cousots [Cousot 77, Cousot 79]. Starting with a standard semantics that maps process states to process states, they introduce a "static semantics" (what Mycroft calls a "collecting semantics") that collects all possible states the process evolves through during execution of the program. Any compile-time analysis is then an abstraction of this static semantics. These abstractions are defined by an abstraction function (and its related inverse, called a "concretisation function"). The Cousots define necessary conditions for abstraction functions to obey typical compile-time properties (*e.g.*, safely approximating the static semantics). The set of suitable abstractions is shown to have a lattice structure.

The Cousots' work, however, is all in terms of imperative, non-functional languages, developed using a flowchart-based intermediate representation. Mycroft's dissertation [Mycroft 81] extends the idea of abstract semantics to a functional language. His main intended application is strictness analysis for normal-order languages — spotting opportunities to evaluate function arguments with call-by-value rules instead of call-by-need rules without changing the result of the program. Mycroft's semantics uses a complex semantic structure called a powerdomain. His analyses are also developed for first-order, normal-order functional languages. My thesis is concerned with higher-order, applicative-order languages that allow side-effects.

A useful collection of work on abstract semantic interpretation is *Abstract Interpretation of Declarative Languages* [Abramsky 87].

12.2 Hudak

Closer to my thesis, and in fact a direct influence on my work, is the work of Hudak and his students. Hudak's paper, "Collecting Interpretations of Expressions" [Hudak 86], defines the problem of a collecting semantics for several variants of functional language — including the higher-order case. Hudak states the critical question of abstract analysis in a functional setting: "what are all possible values to which the expression *exp* might evaluate during program execution?" To answer this question, Hudak introduces the idea of a cache function that collects this information. This is related to the cache function γ that my control-flow semantics computes, with two distinctions. First, my semantics only collects procedural values, ignoring other data. This distinction is merely one of focus; it is trivial to modify my semantics to collect abstractions of other datatypes, as well (as in the case of performing constant propagation — see section 7.3). Secondly, my cache functions are finer-grained than Hudak's because they don't lump all evaluations of a single expression together. That is, where Hudak's caches have functionality Exp $\rightarrow$ Val, my caches have functionality Exp $\times$ Env $\rightarrow$ Val.

Hudak abandons Mycroft's powerdomains for a simpler powerset construction, which I have also used in my semantics. Hudak's constructions are distinguished from the Cousots' in that his semantics are not presented as the closure of a state$\rightarrow$state function. Instead, his semantics functions directly compute the cache produced by evaluating an expression. Interestingly, in the CPS representation I use, these two approaches converge. Although the structure and intent is similar to Hudak's, the CPS representation forces a kind of linear state-to-state transition as evaluation proceeds. This is reflected in the names of the FState and CState sets, implying that they represent the process states that the interpreter evolves through during program execution. I find this dual view to be one of the attractions of the CPS representation: it provides the functional elegance and power of a representation based on the λ-calculus, while simultaneously allowing for a low-level, serialised, operational interpretation.

While Hudak does not provide for computable abstractions for the higher-order case, "Collecting Interpretation of Expressions" provides the general framework for the research that I've done to support my thesis.

Other Hudak papers apply his framework to particular analyses. For example, Hudak and Young have developed a strictness analysis for higher-order languages [Hudak$^+$ 86b]. Hudak has also designed an analysis for determining when functional array updates can be done destructively [Hudak 86b]. However, the higher-order strictness analysis is not guaranteed to terminate, and the update analysis is restricted to first-order languages.

In general, Hudak's work has focussed on purely functional, side-effectless, normal-order languages. The languages I am interested in — Scheme and ML — are applicative and allow side-effects. Further, the set of optimisations I have considered for applications are the classical data-flow analysis optimisations. These optimisations have largely been ignored by the functional-language community. (This is not to say that these optimisations could not be adapted to functional, normal-order languages. But compiler technology for these sorts of languages has not yet progressed to the stage where this is a pressing need.)

12.3 Higher-Order Analyses: Deutsch and Harrison

As I have mentioned above, most work on abstract semantic interpretation has not dealt with the full set of features that languages like Scheme or ML provide. Higher-order languages are rarely treated, and when they are, side-effects are ignored. Two other recent research efforts carried out independently and simultaneously with my own, however, address all of these issues, and so deserve special mention.

Deutsch has developed an analysis for determining the lifetimes of dynamically allocated data at compile time [Deutsch 90]. This allows explicit deallocation of data structures that would otherwise have to be garbage collected, and the functional updating of aggregate data structures by destructively modifying them. Deutsch's intended application is optimising semantic specifications of programming languages. Since he deals with a higher-order language that allows side-effects, however, his techniques apply to Scheme and ML as well.

Deutsch's analysis is defined in terms of an intermediate representation that is somewhat like assembler for a stack machine or Landin's SECD machine [Landin 64]. I believe this representation is not well suited for program transformation and optimisation. An intermediate representation based on a stack machine isn't a high-level representation that permits interesting source-to-source transformations; neither is it a low-level representation that exposes interesting properties of the program in terms of the actual target architecture. In contrast, CPS has the ability to span both ends of this representation spectrum simply by choosing an appropriate set of primops. Steele made this claim for CPS as a general framework for expressing different levels of representation [Steele 78]; Kelsey exploits this property in his transformational compiler [Kelsey 89]. Further, since Deutsch's semantics is a direct, stack-oriented semantics, it requires separate domains and equations for procedures and continuations, whereas a CPS-based representation is much simplified by unifying all control structures as procedures.

One of the most interesting technical aspects of Deutsch's work is the "structured abstraction" approach he takes to developing his abstract semantics. The idea is to define abstractions for composite domains in terms of the domain's structure and abstractions for its component subdomains. For example, if we have a cross-product domain $A \times B$, then we can develop a "generic" abstraction function $\alpha_\times$ for cross-products, expressed in terms of abstraction functions α_A and α_B for the A and B components of the domain.

The advantage of this approach is twofold. First, we can develop a "toolbox" of abstractions for the basic mathematical constructs, such as sets, vectors, series, and functions. After proving that these abstractions satisfy some basic properties in isolation, we can easily verify the correctness of abstractions built with these building blocks. Factoring the correctness proofs in this way simplifies the math in an elegant way.

Further, if we have several abstractions for, say, cross-products, then we can swap between these abstractions in a given application. While the different cross-product abstractions will have different cost/precision properties, they will be interchangeable with respect to correctness considerations. This sort of modularity allows for easy experimentation with different abstractions for a given analysis.

Structure abstractions are first mentioned in a paper by the Cousots' [Cousot 79]. Deutsch's work, however, gives a strong example of this technique in application.[1]

Harrison has also developed semantically-based analyses for Scheme, with the intention of parallelising Scheme programs [Harrison 89]. For example, he performs analysis to disambiguate structure aliasing in order to eliminate spurious edges in the graph of read/write dependencies that limits available parallelism.

Harrison handles full Scheme. His intermediate abstraction, again, is an imperative, non-CPS intermediate representation. Harrison's work takes a coarse-grained view of procedures. His semantics is tuned to handle procedures that have roughly the same size and functionality of FORTRAN or C procedures. His intermediate representation does not use lambdas for "intraprocedural" control structures: goto's, conditional branches, loops and so forth. Instead, there are special, assembler-like syntactic constructs for each of these structures. This increases the complexity of his semantics. Like Deutsch, his non-CPS semantics requires a separate (and complex) domain for continuations, and his semantics equations must have separate cases to handle `call/cc`, procedure call, return, and continuation invocation, which greatly increases the size and complexity of his semantics.

Harrison's semantics is also non-tail-recursive, which fits together with his focus on procedures-as-subroutines. His semantics has the general structure of the Cousot-style state-to-state functions, as opposed to the Hudak-style direct semantics.

An interesting technical aspect of Harrison's work is his contour abstraction, which is based on finite regular expressions over relative procedure call/return sequences. This is an abstraction that is well matched to his focus on subroutine boundaries. Note that this abstraction could be adapted to my framework. Suitably modified to account for tail-recursive procedure calls, Harrison's abstraction might work well for the coarse-grained procedural boundaries for which it was designed. (Whereas other abstractions could be used for the finer-grained lambdas, giving a sort of hybrid abstraction. See section 11.3.3.)

In contrast to Harrison's, my work takes a fine-grained view of procedures. My intermediate representation represents all transfers of control with a tail-recursive procedure call. This provides a uniform representation for control flow that simplifies the analysis machinery. While Harrison focusses on inter-procedural control and data flow, I am more interested in intra-procedural applications — the classical data-flow optimisations, in a functional, higher-order setting.

Harrison's work does not delve into as much detail on his analysis as I have in this dissertation. Harrison skips many of the more tedious correctness proofs that I've presented in chapter 4. He also doesn't present details of his algorithms or prove their correctness. This is almost certainly due to time and space restrictions: Harrison's semantic analysis is only one part of his research on interprocedural analysis and parallelisation.

Finally, neither Harrison, Deutsch, nor any of the other researchers I've discussed in this chapter have dealt with the environment problem. The solution I present in chapter 8 is a unique contribution, to the best of my knowledge.

---

[1] Deutsch's research report describes more of the structured abstractions than the conference publication.

Chapter 13

Conclusions

My research makes the following contributions:

- A solution to the control-flow problem for higher-order languages, where the allowed language features includes side-effects and `call/cc`.
- A theoretical foundation for this analysis, complete with correctness proofs.
- Algorithms to compute the analysis, again, complete with correctness proofs.
- Example applications using the analysis.
- Reflow analysis: a powerful extension for solving the environment problem.
- Example applications using this extension.

This all adds up to a general framework for performing data-flow analysis for higher-order languages.

As I stated at the beginning of this dissertation, my thesis is that control-flow analysis is feasible and useful for higher-order languages. The analysis, algorithms, and associated correctness proofs should establish its feasibility. The example optimisations should establish its utility.

A secondary thesis of this work might be called the "CPS representation thesis." I hope I have strengthened the case for CPS-based compiler intermediate representations – their simplicity, theoretical underpinnings, and power support the transition of compilers for higher-order languages into the highly-optimising niche. With CPS, we arrive at a new view of traditional data-flow optimisations in a functional, higher-order setting: one that emphasises variable binding over variable assignment, and where the basic control/environment construct is lambda/procedure-call.

A closing thought: As we've seen, control-flow analysis in higher-order languages is really data-flow analysis in the higher-order procedure case. As procedural data flows to points in the control-flow graph, new paths become possible, opening up new routes for data flow, and so the loop is closed. Thus the title of this dissertation is perhaps a bit misleading. In order to solve the control-flow problem, we must also solve the data-flow problem, and our analysis must further involve considerations of environment structure.

This is a fundamental property of having the lambda operator in our language. Lambda is an operator that simultaneously establishes data, control, and environment structure:

- Evaluating a lambda creates a closure, which is a first-class data structure.
- A lambda is a control point to which control can be transferred by procedure call; in a CPS representation, lambda is *the* fundamental control structure.
- Entering a lambda causes environment structure to be established by the binding of the lambda's parameters; again, in higher-order languages lambda binding is *the* fundamental environment structure.

In the lambda operator, all three fundamental program structures — data, control, and environment — meet and intertwine. Thus, any analysis technique for a higher-order language must be prepared to handle the three facets of lambda. In essence, the work in this dissertation has been the application of classical data-flow analysis in the presence of lambda.

Appendix A

1CFA Code Listing

Dear Mr. Shivers,
Your code horrified me ... patches to failures ... not principled
— mob 9/22/90

... the techniques described in your ... lisp posting are quite frankly horrid ... isn't even legal ... it breaks enormously ... ??? bogus!!
— ted 9/22/90

This appendix contains a code listing for my 1CFA implementation. Before running this analysis, a modified copy of the ORBIT compiler's front end is used to CPS-convert the code and produce a syntax tree. This tree is the input to the 1CFA analysis listed here.

A.1 Preliminaries

A.1.1 Details of the dialect

The program is written in T, a dialect of Scheme with additional extensions from several macro packages. Here is a brief description of the various macro packages that are used in this code.

The UCI LISP `for` macro is syntactic sugar for `map`-type loops. The expression

```
(for (x in <list1>)
     (y in <list2>)
  (save (f (- y x))))
```

expands into

```
(map (λ (x y) (f (- y x))) <list1> <list2>)
```

If the `do` keyword is used instead of the `save` keyword, then the macro expands into a `for-each` (in T, a `walk`) instead of a `map`.

A more powerful iteration construct is the Yale `loop` macro. Without giving a detailed description of the entire macro, the following example should give the general structure:

```
(loop (initial (m 0 (max m x)))
      (for x in l)
      (incr i .in 0 to (vector-length v))
      (do (set (vref v i) x))
      (result m))
```

This expression iterates over a list `l` and a vector `v`, storing the elements of `l` into `v`. The loop terminates when it runs off the end of either `l` or `v`. The `.in` keyword means the range of the `i` variable includes `0` but not `(vector-length v)` (`in.` is the other way around; `.in.` includes both endpoints; `in` includes neither). The variable `m` is initialised to `0`; each time through the loop, it is updated to be the largest element scanned. This maximum value is returned as the result of the loop.

The `def-struct` macro is macro support for T's record structures. The structure

```
(def-struct ship
    x
    y
    (size 1000))
```

defines a `ship` record with three fields. The `size` field is initialised by default to `1000`. This instance of `def-struct` defines three accessor functions: `ship:x`, `ship:y` and `ship:size`. These slots can be assigned with the T `set` form:

```
(set (ship:x shipa) 100)
```

There is also a predicate `ship?` that returns true only when applied to ship structures. Finally, the constructor macro `ship:new` is used to create new structures, with record slots optionally specified by name. For example,

```
(ship:new x 7 size 4)
```

creates a new ship structure, with the `x` field initialised to `7`, the `size` field initialised to `4`, and the `y` field unspecified.

I use `?` as an abbreviation for `cond`. The T `cond` form has a useful extension. If a clause is of the form

(*test* `=>` *fn*)

then if *test* is true, *fn* is applied to the value it produces. T also defines the `else` constant to be a true value. So the expression

```
(? ((assq var alist) => cdr)
   (else (error "unbound variable ~s" var)))
```

looks up a variable binding in an alist, signalling an error if the variable is unbound.

Finally, this code uses a small package for manipulating lists as sets. The function `set:new` is identical to the `list` function, so `(set:new 2 3 5)` constructs a set (list) of the first three primes. The procedure

```
(union elt= seta setb)
```

returns the union of `seta` and `setb`, where the elements of the list are compared with the `elt=` function. The macro

```
(difference-and-union elt= s1 s2 (d u) . body)
```

computes the difference and union of sets `s1` and `s2`. The `body` forms are then executed in an environment with `d` bound to the difference and `u` bound to the union.

A.1.2 The syntax tree

The syntax tree produced by the ORBIT front end is built with the nodes shown in figure A.1. I'll only describe here the parts of the node data structures that are useful to the 1CFA analysis. The node data structures are not defined by the `def-struct` macro, and the field syntax is different. The procedure that accesses the *field* component of node type *node* is named *node-field*, not *node:field*. For example, `reference` node `r`'s `variable` field is accessed with `(reference-variable r)`.

Reference A variable reference node has a single field, the variable node it references.

Primop Every occurence of a primop in the code tree is a distinct primop node. A primop node has one field, its value. This is a separate primop def-struct, defined as:

```
(def-struct primop
    id                    ; Name of this primop
    ic                    ; Single ic or a vector of ic's
    1cfa-propagator)      ; Defines action of F
```

(In fact, there are a few other fields in the primop def-struct, but they are not used by the 1CFA analysis.) The `primop:ic` field holds either a single call node (for simple primops), or a two-element vector of call nodes (for conditional primops). These call nodes are the ic markers for the primop. Note that all occurrences of a primop share a single primop def-struct, so they all share the same ic nodes. This is acceptable in the 1CFA abstraction, since the continuation calls from the different occurrences of the primop will be distinguished in the call cache by the the contour environment part of the $\left\langle ic, \widehat{\beta}\right\rangle$ call context. The `primop:1cfa-propagator` field holds a procedure that defines $\widehat{\mathcal{F}}$ for this primop — see the `functionalise` procedure in the code listing.

Literal A literal node has no fields that are useful for control-flow analysis.

```
reference
    variable

primop
    value

literal
    value

lambda
    body          ; call or labels node
    variables     ; bound variables

call
    proc+args     ; list of subforms

labels
    variables     ; bound variables
    lambdas       ; label functions
    body          ; call or labels node

variable
    binder        ; call or labels node
```

Figure A.1: Code tree data structures

Lambda A lambda node has two fields: one for the lambda's body (a call or labels node), and one for the lambda's variables list (a list of variable nodes). The predicate `body-node?` is defined to return true on both call and labels nodes.

Call A call node has a single field, the list of its subexpressions: `proc+args`.

Labels `Labels` is an old name for `letrec`; a labels node is a `letrec` form. The `variables` field is a list of the variable nodes bound by the labels node. The `lambdas` field is a list of the lambdas evaluated by the labels node and bound to its variables. The `body` field is the body of the labels node, and is a call or labels node.

Variable A variable has a `binder` field that is the lambda or labels node that binds it. A variable node has other fields as well, such as the list of all its reference nodes, its name, and so forth, but these are not used by the 1CFA analysis.

Continuations come first in a call node's argument list and a lambda node's bound-variable list.

```
(herald 1cfa (env t (tutil for)
                    (tutil loop)
                    (tutil def-struct)
                    (tutil hacks)))
(require def-struct (tutil def-struct))
;(import t-implementation-env hash)

;;; This file contains code for doing 1st-order control flow analysis
;;; with 1st-order functional results.

(define (1clo-analyse node)
  (clear-store)
  (clear-venv)
  (clear-xconts)
  (clear-ans-cache)
  (clear-call-memos)
  (xfun-fun (list (set:new xcont) (A node (empty-benv))) xcall))


;;; 1st-order closure def
;;; ===========================================================================
;;; A "1st-order closure" (1clo) is a <code,env> pair, where the env is a
;;; 1st-order env. A "1st-order function" (1fun) is a 1clo or a primop.

(define-constant make-1clo cons) ; 1clo = (lambda . benv)
(define-constant 1clo:lam  car)
(define-constant 1clo:benv cdr)
(define (1clo? x) (and (pair? x) (lambda-node? (car x)))) ; Not precise

(define (1fun= a b)
  (if (primop? a) (eq? a b)
      (and (1clo? b)
           (eq? (1clo:lam a) (1clo:lam b))
           (benv= (1clo:benv a) (1clo:benv b)))))

;;; Call contexts
;;; ===========================================================================
;;; A call context is a <call,benv> pair.

(define-constant make-cctxt cons) ; call context = (call . benv)
(define-constant cctxt:call car)
(define-constant cctxt:benv cdr)
(define-constant cctxt? pair?)
(define-constant cctxt= (lambda (a b) (and (eq? (car a) (car b))
                                           (benv= (cdr a) (cdr b)))))
(define (hash-cctxt cc)
  (fixnum-logxor (descriptor-hash (car cc))
                 (benv-hash (cdr cc))))
```

```
;;; Answer cache code
;;; ===========================================================================
;;; Ans cache maps a call context (a <call,benv> pair) to a set of 1funs
;;; that could have been called from CALL in context BENV.

(lset *ans-cache* nil)
(define (clear-ans-cache)
  (set *ans-cache* (make-hash-table cctxt? hash-cctxt cctxt= t 'ans-cache)))

;;; Record in the ans cache that we called FUNS from call context <CALL,BENV>
(define (record-call call benv funs)
  (modify (table-entry *ans-cache* (make-cctxt call benv))
          (lambda (entry) (union 1fun= entry funs))))

;;; Global variable environment code
;;; ===========================================================================
;;; The VEnv maps a contour (a lambda or labels node) and a var to a
;;; set of 1clo's. (but not primops)
(lset *venv* nil)
(lset *venv-stamp* 0) ; When we alter the venv, bump this guy.
(define (clear-venv)
  (set *venv* (make-hash-table pair?
                               (lambda (x)
                                 (fixnum-logxor (descriptor-hash (car x))
                                                (descriptor-hash (cdr x))))
                               (lambda (k1 k2)
                                 (and (eq? (car k1) (car k2))
                                      (eq? (cdr k1) (cdr k2))))
                               t
                               'venv-table))
  (set *venv-stamp* 0))

(define (venv-lookup contour var)
  (table-entry *venv* (cons contour var)))

(define (venv-augment contour vars arglis)
  (for (v in vars)
       (argset in arglis)
       (do (let* ((index (cons contour v))
                  (entry (table-entry *venv* index)))
             (difference-and-union 1fun= argset entry (d u)
               (? (d
                   (set (table-entry *venv* index) u)
                   (set *venv-stamp* (fx+ *venv-stamp* 1)))))))))
```

```
;;; Contour env code
;;; ===========================================================================
(define-integrable (empty-benv) '())

(define (benv-lookup benv binder)
  (? ((assq binder benv) => cdr)
     (else (error "benv-lookup: couldn't find ~a in ~a" binder benv))))

(define-integrable (benv-augment benv binder contour); benv[binder |-> contour]
  (cons (cons binder contour) benv))

(define (benv= a b) ; Assumes the benv's are over the same code, as it were.
  (assert (every? (lambda (a b) (eq? (car a) (car b))) a b)) ; paranoia
  (every? (lambda (a-entry b-entry) (eq? (cdr a-entry) (cdr b-entry)))
          a b))

(define-constant (benv-hash benv)
  (loop (for elt in benv)
        (initial (val 42 (fixnum-logxor (descriptor-hash (cdr elt)) val)))
        (result val)))
```

```
;;; Call memoising
;;; ===========================================================================
;;; For every C application, we remember the Benvs and (implicit args) Venv
;;; timestamp, store timestamp and xconts timestamp indexed by the call.
;;; This information is stored as a hash table mapping <call,benv> pairs
;;; to cmemo structs that track the time stamps for the venv, store, and
;;; xconts. Note that this hash table keys on <call,benv> pairs,
;;; just like the answer cache, so we can snarf the basic structure
;;; of the a.c. table for the call memo table.

(def-struct cmemo
  (venv-ts 0)
  (store-ts 0)
  (xconts-ts 0))

;;; The *CALL-MEMOS* table maps call contexts to cmemo structs.
(lset *call-memos* nil)

(define (clear-call-memos)
  (set *call-memos* (make-hash-table cctxt? hash-cctxt cctxt= t 'call-memos)))

(define (fetch-call-memo call benv)
  (let ((key (make-cctxt call benv)))
    (or (table-entry *call-memos* key)
        (let ((entry (cmemo:new))) ; 1st time thru CALL
          (set (table-entry *call-memos* key) entry)
          entry))))

(define (call-memoised? call benv)
  (let ((cm (fetch-call-memo call benv)))
    (and (fx>= (cmemo:venv-ts  cm)  *venv-stamp*)
         (fx>= (cmemo:store-ts cm)  *store-stamp*)
         (fx>= (cmemo:xconts-ts cm) *xconts-stamp*))))

(define (memoise-call call benv)
  (let ((entry (fetch-call-memo call benv)))
    (set (cmemo:venv-ts entry)   *venv-stamp*)
    (set (cmemo:store-ts entry)  *store-stamp*)
    (set (cmemo:xconts-ts entry) *xconts-stamp*)))
```

```
;;; A & C
;;; ============================================================================

(define (A form benv)
  (? ((reference-node? form)
      (let ((var (reference-variable form)))
        (venv-lookup (benv-lookup benv (variable-binder var)) var)))
     ((primop-node? form) (set:new (primop-value form)))
     ((literal-node? form) (empty-set))
     ((lambda-node? form) (set:new (make-1clo form benv)))
     (else (error "type error"))))

(define (C form benv)
  (assert (body-node? form))
  (? ((not (call-memoised? form benv))
      (memoise-call form benv)
      (if (call-node? form) (C-call form benv)
          (C-labels form benv)))))

(define (C-call call benv)
  (assert (call-node? call))
  (destructure (( (fun . args) (call-proc+args call) ))
    (let ((funset (A fun benv))
          (arglis (for (arg in args) (save (A arg benv)))))
      (record-call call benv funset)
      (for (f in funset)
           (do ((functionalise f) arglis call))))))

(define (functionalise funspec)
  (? ((primop? funspec) (P funspec))
     ((eq? funspec xfun) xfun-fun)      ; Calling the external function
     ((eq? funspec xcont) xcont-fun)    ; Calling the external continuation.
     (else
      (let* ((lam (1clo:lam funspec))
             (body (lambda-body lam))
             (formals (lambda-variables lam))
             (nformals (length formals))
             (benv (1clo:benv funspec)))
        (lambda (arglis b)
          (? ((fx= nformals (length arglis))
              (venv-augment b formals arglis)
              (C body (benv-augment benv lam b)))))))))


(define (C-labels lab benv)
  (let ((benv (benv-augment benv lab lab)))
    (venv-augment lab
                  (labels-variables lab)
                  (for (f in (labels-lambdas lab))
                       (save (A f benv))))
    (C (labels-body lab) benv)))
```

```
;;; Primop definitions
;;; ===========================================================================

;;; Since primops aren't 1st-class, we can statically check their
;;; argspectrum, so no need to check at runtime.

(define-integrable (P prim) (primop:1cfa-propagator prim))

(define-syntax (define-p prim fun)
  (let ((name (concatenate-symbol 'p/ prim))
        (syn-name (concatenate-symbol 'primop/ prim)))
    `(block (define ,name ,fun)
            (set (primop:1cfa-propagator ,syn-name) ,name))))

(define-syntax (define-simple-p prim)
  (let ((syn-name (concatenate-symbol 'primop/ prim)))
    `(define-p ,prim
       (lambda (arglis contour)
         (destructure (( (c a b) arglis ))
           (let ((ic (primop:ic ,syn-name))
                 (benv (primop-env ,syn-name contour)))
             ;; record a call to the conts from ctxt <ic, [prim -> contour]>
             (record-call ic benv c)
             (for (f in c)
                  (do ((functionalise f) <@> ic))))))))))

(define-simple-p +)
(define-simple-p *)
(define-simple-p -)

(define-p test
  (lambda (arglis contour)
    (destructure (( (c a p . v) arglis ))
      (let* ((benv (primop-env primop/test contour))
             (ic-vec (primop:ic primop/test))
             (ic0 (vref ic-vec 0))
             (ic1 (vref ic-vec 1)))
        ;; record a call to C conts from context <ic0test, [test -> contour]>
        (record-call ic0 benv c)
        (for (f in c)
             (do ((functionalise f) <> ic0)))
        ;; record a call to A conts from context <ic1test [test -> contour]>
        (record-call ic1 benv a)
        (for (f in a)
             (do ((functionalise f) <> ic1)))))))

(define (primop-env primop contour)
  (benv-augment (empty-benv) primop contour))

(define-constant <@> (list (empty-set))) ; arglis of 1 empty set.
(define-constant <> '()) ; arglis of 0 length
```

```
;;; The approximate store
;;; ============================================================================

(lset *store* nil)
(lset *store-stamp* 0)

(define (clear-store)
  (set *store* (set:new xfun)) ; initial store contains the xfun.
  (set *store-stamp* 1)) ; Must use the xfun...

(define (update-store new)
  (difference-and-union 1fun= new *store* (d u)
    (? (d
        (set *store* u)
        (set *store-stamp* (fx+ *store-stamp* 1))))))
```

```
;;; (DEFINE-STORE-P prim args stashers ic-call-argv)
;;; ===========================================================================
;;; This is for defining primops that access or modify the store.  This
;;; form defines a primop PRIM with arguments ARGS. The first element of
;;; ARGS is the continuation. STASHERS is a subset of the args. When this
;;; primop is invoked
;;; - A call from the primop's i.c. site to the continuations is recorded.
;;; - The stasher arguments are placed in the store.
;;; - The primop's continuations are called with the argument
;;;   vector IC-CALL-ARGV, which is typically <@> or (LIST *STORE*).
;;;
;;; For example,
;;; (DEFINE-STORE-P NEW (C V) (V) <@>)
;;; define the NEW primop, which takes two arguments: a continuation C
;;; and a value V. The values passed in for V should be put in the store.
;;; NEW produces no procedures, so the continuation should be applied
;;; to the empty set @ -- that is, the arg vector <@>.

(define-syntax (define-store-p prim args stashers ic-call-argv)
  (let ((syn-name (concatenate-symbol 'primop/ prim)))
    `(define-p ,prim
       (lambda (arglis contour)
         (destructure (( ,args arglis ))
           (let ((ic (primop:ic ,syn-name))
                 (benv (primop-env ,syn-name contour)))
             ;; record a call to the conts from ctxt <ic, [prim -> contour]>
             (record-call ic benv c)
             ;; All stashed args -> store:
             ,@(for (s in stashers) (save `(update-store ,s)))
             ;; Do the ic call:
             (for (f in ,(car args)) ; continuations
                  (do ((functionalise f) ,ic-call-argv ic))))))))))


(define-store-p new (c v) (v) <@>)
(define-store-p contents (c a) () (list *store*))
(define-store-p set (c a v) (v) <@>)

;;; The simple-minded CPS-converter assumes all expressions evaluate
;;; to exactly one value (so their continuations have one formal parameter).
;;; However, SET is not defined to produce a value. We hack this by assuming
;;; SET returns a constant (e.g., #f). So the abstract arg vec for the
;;; continuation call is <@>, not <>.

(define-store-p cons (c a d) (a d) <@>)
(define-store-p car (c a) () (list *store*))
(define-store-p cdr (c a) () (list *store*))
```

```
;;; XFUN and XCONT
;;; ============================================================================
(define xlam  (create-lambda-node '(#f #f) '#f))
(define xcall (create-call-node 3 1)) ;(proc cont arg) or something.
(define xret  (create-call-node 2 0))  ;(cont arg) or something.
(define xfun  (make-1clo xlam (empty-benv)))
(define xcont-lam (create-lambda-node '(#f) '#t))
(define xcont (make-1clo xcont-lam (empty-benv)))

;;; One benv will do; splitting it out based on where the xfun and xcont
;;; are called from doesn't do any good, since we mix all the args together
;;; anyway.
(define x-benv (benv-augment (empty-benv) xlam '#f)) ; Bogus contour

(define xfun-fun
  (lambda (arglis contour)
    (update-xconts (car arglis)) ; First arg is continuation.
    (for (a in (cdr arglis)) (do (update-store a))) ; rest are non-conts
    ;; This memo check does double duty for the xcall & the xret.
    (? ((not (call-memoised? xcall x-benv))
        (memoise-call xcall x-benv)

        (bind-xarg-hackery (lambda () ; Reset the arglist hackery
          ;; Do the xret: (*xconts* *store* *store* ...)
          (record-call xret x-benv *xconts*)
          (for (f in *xconts*)
               (do ((functionalise f) (xret-arglis-rep (arity f)) xret)))

          ;; Do the xcall: (*store* *xconts* *store* *store* ...)
          (record-call xcall x-benv *store*)
          (for (f in *store*)
               (do ((functionalise f) (xcall-arglis-rep (arity f)) xcall)))
          ))))))

(define (arity fun)
  (if (primop? fun)
      (error "Why are you calling ARITY on a primop (~a)?" fun)
      (length (lambda-variables (1clo:lam fun)))))
```

```
;;; Almost identical to XFUN-FUN, except that all args are non-continuations.
(define xcont-fun
  (lambda (arglis contour)
    (for (a in arglis) (do (update-store a))) ; all args are non-conts
    ;; This memo check does double duty for the xcall & the xret.
    (? ((not (call-memoised? xcall x-benv))
        (memoise-call xcall x-benv)

        (bind-xarg-hackery (lambda () ; Reset the arglist hackery
          ;; Do the xret: (*xconts* *store* *store* ...)
          (record-call xret x-benv *xconts*)
          (for (f in *xconts*)
               (do ((functionalise f) (xret-arglis-rep (arity f)) xret)))

          ;; Do the xcall: (*store* *xconts* *store* *store* ...)
          (record-call xcall x-benv *store*)
          (for (f in *store*)
               (do ((functionalise f) (xcall-arglis-rep (arity f)) xcall)))
          ))))))


;;; Make arg lists of repeated sets for calls from XRET and XCALL.
;;; ===========================================================================
;;; XRV[i] is a i elt list, each elt of which is *STORE*.

(lset xcall-rep-vec nil)
(lset xret-rep-vec nil)
(lset *xret-rep-index*  0)      ; Max slot in xr-r-v we have lazily installed.
(lset *xcall-rep-index* 0)      ; Max slot in xc-r-v we have lazily installed.

(define rep-vec-pool
        (make-pool 'rep-vec-pool (lambda () (vector-fill (make-vector 20) '#f))
                   5 vector?))

(define-integrable (bind-xarg-hackery cont)
  (bind ((xcall-rep-vec (obtain-from-pool rep-vec-pool))
         (xret-rep-vec  (obtain-from-pool rep-vec-pool))
         (*xret-rep-index*  0)
         (*xcall-rep-index* 0))
    (cont)
    (return-to-pool rep-vec-pool xcall-rep-vec)
    (return-to-pool rep-vec-pool xret-rep-vec)))
```

```
;;; (*store* *store* ...)
(define (xret-arglis-rep nargs)
  (if (fx>= nargs 20) (error "arglis-rep: nargs too big (~a)" nargs))
  (if (fx<= nargs *xret-rep-index*)
      (vref xret-rep-vec nargs)
      (loop (initial (arglis (vref xret-rep-vec *xret-rep-index*)))
            (incr i in. *xret-rep-index* to nargs)
            (next (arglis (cons *store* arglis)))
            (do (set (vref xret-rep-vec i) arglis))
            (after (set *xret-rep-index* nargs))
            (result arglis))))

;;; (*xconts* *store* *store* ...)
(define (xcall-arglis-rep nargs)
  (? ((fx>= nargs 20)
      (error "arglis-rep: nargs too big (~a)" nargs))
     ((fx<= nargs *xcall-rep-index*) ; Already computed?
      (vref xcall-rep-vec nargs))
     (else
      (loop (initial (arglis (xret-arglis-rep (fx- nargs 1)) (cdr arglis)))
            (decr i .in nargs to *xcall-rep-index*)
            (do (set (vref xcall-rep-vec i)
                     (cons *xconts* arglis)))
            (after (set *xcall-rep-index* nargs))
            (result (vref xcall-rep-vec nargs))))))


;;; Escaped continuations
;;; ===========================================================================

(lset *xconts* nil)
(lset *xconts-stamp* 0)

(define (clear-xconts)
  (set *xconts* (set:new xcont)) ; initial xconts set contains the xcont.
  (set *xconts-stamp* 1)) ; Must use the xcont...

(define (update-xconts new)
  (difference-and-union 1fun= new *xconts* (d u)
    (? (d
        (set *xconts* u)
        (set *xconts-stamp* (fx+ *xconts-stamp* 1))))))
```

Appendix B

Analysis Examples

B.1 Puzzle example

The first example is the little environment puzzle from page 120. The program is shown in figure B.1, both in Scheme and its CPS intermediate representation. In the CPS form, lambda and call expressions are subscripted with numeric labels (the particular labels used are just those chosen by the T run-time system when printing out this example). As a special case, the CPS `test` primop requires its first argument to be a type or comparison predicate (*e.g.*, `integer?` or `eq?`). This syntactic extension, inherited from the ORBIT front end, is equivalent to defining separate primops `test-integer`, `test-eq?`, and so forth.

CPS conversion introduces continuations into parameter lists and call expressions. ORBIT's front-end and the analysis code listed in appendix A place these continuation arguments first in calls and parameter lists, *e.g.*, `(+ k 3 4)`. This has some programming benefits, but makes the code unreadable. So, I have edited the CPS code shown in this appendix to use the continuation-last convention, *e.g.*, `(+ 3 4 k)`.

Applying the 1CFA program of appendix A to the puzzle example takes .67 seconds of CPU time, running interpreted T on a DECstation 3100 PMAX. Applying the type-recovery algorithm of chapter 9 takes 5.1 seconds under the same configuration.

The information produced by the type-recovery algorithm is given in table B.2. If a variable is qualified with a call site, *e.g.* `x(c51)`, then the type applies to the reference occuring in that call expression. Otherwise, the type applies to all references to that variable. So type recovery is able to correctly distinguish the two bindings of `x`, and determine from the type test `(test fixnum? ...)` and the control-flow information that the `x` closed over in lambda `l36` is a floating-point number.

The call cache produced by the 1CFA program is listed in figure B.3. It is simply a dump of the `*ans-cache*` table constructed by the program. The output has been lightly edited to improve its legibility. Each entry in the table maps a call context *cc* to a list of abstract procedures called from that call context:

$$cc \texttt{ -> } (proc_1 \ proc_2 \ \ldots)$$

```
(λ () (let ((f (λ (x h)
                   (test (fixnum? x) (h) (λ () x)))))
        (f 3 (f 2.7 '#f))))
```

Scheme

```
(λ (k) ((λ (f) (f 2.7 '#f (λ (v) (f 3 v k))))
 34    41 42   43         49    40
        (λ (x h k1)
         39
          (test fixnum? x
           44
                (λ () (h k1))
                 47   50
                (λ () (k1 (λ (k2) (k2 x)))))))))
                 37   48   36     51
```

Annotated CPS

Figure B.1: Puzzle example

f, v, k, k1, k2	proc
x(c44)	fixnum|float
x(c51)	float
h	false|proc

Table B.2: Recovered type information

An abstract closure is a list whose head is a lambda and whose tail is an abstract contour environment. So

`(l47 (l39 . c40) (l34 . xcall))`

is the closure over lambda l47 with contour environment [`l39` $\mapsto$ `c40`, `l34` $\mapsto$ `xcall`]. This is the environment made in the 1CFA abstraction by entering lambda `l34` from the external call, and subsequently entering lambda `l39` from call `c40`. Most of the binding contours are given mnemonic names such as `[top]` or `[if]` that are listed in the initial comment lines of the figure. The special symbols `xcall`, `xret`, `xlam`, `xcont`, `ictest0`, and `ictest1` represent the external call, external return, external lambda, external continuation, and internal call sites for the `test` primop, respectively.

Note that the

`(xcall (xlam)) -> (l36 [else] [f2] [topcall])`

edge in the call graph could be refined away by a post-type-recovery pass. In call context `([f2] [topcall])`, `x` is known to be bound to an integer, so the else branch is not taken. Removing this control-flow arc would also prune the extra arc

`(c51 (l36 . xcall) [else] [f2] [topcall]) -> (xcont)`

```
;;; [top]       (l34 . xcall)
;;; [let]       (l42 . c41)
;;; [f1]        (l39 . c43)
;;; [f1-ret]    (l49 . c48)
;;; [f2]        (l39 . c40)
;;; [if]        (test . c44)
;;; [then]      (l47 . ictest0)
;;; [else]      (l37 . ictest1)

(xret (xlam)) -> (xcont)
(xcall (xlam)) -> ((l36 [else] [f2] [topcall])
                   (l34)
                   (xlam))
(c41 [topcall]) -> ((l42 [topcall]))
(c43 [let] [topcall]) -> ((l39 [topcall]))
(c44 [f1] [topcall]) -> (test)
(ictest0 [if]) -> ((l47 [f2] [topcall])
                   (l47 [f1] [topcall]))
(ictest1 [if]) -> ((l37 [f2] [topcall])
                   (l37 [f1] [topcall]))
(c48 [else] [f1] [topcall]) -> ((l49 [let] [topcall]))
(c40 [f1-ret] [let] [topcall]) -> ((l39 [topcall]))
(c44 [f2] [topcall]) -> (test)
(c50 [then] [f2] [topcall]) -> ((l36 [else] [f1] [topcall]))
(c51 (l36 . c50) [else] [f1] [topcall]) -> (xcont)
(c48 [else] [f2] [topcall]) -> (xcont)
(c51 (l36 . xcall) [else] [f2] [topcall]) -> (xcont)
```

Figure B.3: 1CFA call cache for puzzle example

B.2 Factorial example

The second example is an iterative factorial procedure, shown in figure B.4. Note the initial type-check performed before entering the loop. Type recovery is able to deduce from this test that all variable references inside the loop are integers (table B.5). The type table doesn't list an entry for `fact`. As discussed in chapter 9, variables bound by `letrec` expressions are not typed by the analysis, since they are known to be have type `proc`. The 1CFA analysis takes .59 seconds to compute and the type recovery takes 7.8 seconds, under the same conditions as the puzzle measurements. The call cache produced by the 1CFA analysis is shown in figure B.6.

```
(λ (n) (letrec ((fac (λ (a m) (test (= 0 m) a
                                   (fac (* a m) (- m 1))))))
         (test (integer? n) (fac 1 n)
               -1))) ; error value
```

Scheme

```
(λ (n k)
34
  (letrec ((fac (λ (a m c)
      37        42
                   (test = 0 m
                   43
                     (λ () (c a))
                     44    46
                     (λ () (* a m (λ (v1)
                     50    51        54
                       (- m 1 (λ (v0)
                       55        58
                         (fac v1 v0 c))))))))))
                         45
    (test integer? n (λ () (k -1))
    43               48    49
                     (λ () (fac 1 n k)))))
                     40    41
```

Annotated CPS

Figure B.4: `fact` example

n(c36)	top
v0, v1, m, a, n(c41)	int
k, c	proc

Table B.5: Recovered type information

```
;;; [letrec]    (lab37 . lab37)
;;; [topcall]   (l34 . xcall)
;;; [if1]       (test1 . c36)
;;; [then1]     (l40 . icif0)
;;; [else1]     (l48 . icif1)
;;; [if2]       (test2 . c43)
;;; [enter]     (l42 . c41)
;;; [loop]      (l42 . c45)
;;; [then2]     (l44 . icif0)
;;; [else2]     (l50 . icif1)
;;; [ic*]       (* . c51)
;;; [ic-]       (- . c55)
;;; [*ret]      (l54 . c53)
;;; [-ret]      (l58 . c57)

(xret (xlam)) -> (xcont)
(xcall (xlam)) -> ((l34) (xlam))
(c36 [letrec] [topcall]) -> (test1)
(icif0 [if1]) -> ((l40 [letrec] [topcall]))
(c41 [then1] [letrec] [topcall]) -> ((l42 [letrec] [topcall]))
(c43 [enter] [letrec] [topcall]) -> (test1)
(icif0 [if2]) -> ((l44 [loop] [letrec] [topcall])
                  (l44 [enter] [letrec] [topcall]))
(c46 [then2] [enter] [letrec] [topcall]) -> (xcont)
(icif1 [if1]) -> ((l48 [letrec] [topcall]))
(c49 [else1] [letrec] [topcall]) -> (xcont)
(icif1 [if2]) -> ((l50 [loop] [letrec] [topcall])
                  (l50 [enter] [letrec] [topcall]))
(c51 [else2] [enter] [letrec] [topcall]) -> (*)
(c53 [ic*]) -> ((l54 [else2] [loop] [letrec] [topcall])
                (l54 [else2] [enter] [letrec] [topcall]))
(c55 [*ret] [else2] [enter] [letrec] [topcall]) -> (-)
(c57 [ic-]) -> ((l58 [*ret] [else2] [loop] [letrec] [topcall])
                (l58 [*ret] [else2] [enter] [letrec] [topcall]))
(c45 [-ret] [*ret] [else2] [enter] [letrec] [topcall]) ->
        ((l42 [letrec] [topcall]))
(c43 [loop] [letrec] [topcall]) -> (test1)
(c46 [then2] [loop] [letrec] [topcall]) -> (xcont)
(c51 [else2] [loop] [letrec] [topcall]) -> (*)
(c55 [*ret] [else2] [loop] [letrec] [topcall]) -> (-)
(c45 [-ret] [*ret] [else2] [loop] [letrec] [topcall]) ->
        ((l42 [letrec] [topcall]))
```

Figure B.6: 1CFA call cache for `fact` example

Bibliography

A book is a fat thing bound by itself, or the equivalent in some other medium of a fat thing bound by itself. A non-book is a thin thing or a thing that's part of something else, or the equivalent in some other medium of a thin thing or a thing that's part of something else.
— M-C. van Leunen

[Abramsky 87] Samson Abramsky and Chris Hankin (editors).
Abstract Interpretation of Declarative Languages.
Ellis Horwood, 1987.

[Aho+ 86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman.
Compilers, Principles, Techniques and Tools.
Addison-Wesley, 1986.

[Appel+ 89] Andrew W. Appel and Trevor Jim.
Continuation-passing, closure-passing style.
In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, January 1989.

[Berlin 89] Andrew A. Berlin.
A compilation strategy for numerical programs based on partial evaluation.
Master's thesis, MIT, February 1989.

[Blelloch 90] Guy E. Blelloch.
Vector Models for Data-Parallel Computing.
MIT Press, 1990.

[Blelloch+ 90] Guy E. Blelloch and Gary W. Sabot.
Compiling collection-oriented languages onto massively parallel computers.
Journal of Parallel and Distributed Computing 8(2), February 1990.

[Bloss+ 86] Adrienne Bloss and Paul Hudak.
Variations on strictness analysis.
In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 132–142, August 1986.

[Brookes 89] Stephen Brookes.
Course notes, graduate class on programming language semantics.
School of Computer Science, CMU, 1989.
These notes are currently being turned into a book.

[Chaitin 82] G. J. Chaitin.
Register allocation & spilling via graph coloring.
In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* published as *SIGPLAN Notices* 17(6), pages 98–105. Association for Computing Machinery, June 1982.

[Chaitin+ 81] G. J. Chaitin, *et al.*
Register allocation via coloring.
Computer Languages 6:47–57, 1981.

[Chow+ 84] Frederick Chow and John Hennessy.
Register allocation by priority-based coloring.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction,* published as *SIGPLAN Notices* 19(6), pages 222–232. Association for Computing Machinery, June 1984.

[Cousot 77] Patrick Cousot and Radhia 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.

[Cousot 79] Patrick Cousot and Radhia Cousot.
Systematic design of program analysis frameworks.
Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, pages 269–282, 1979.

[Curtis 90] Pavel Curtis.
Constrained Quantification in Polymorphic Type Analysis.
Ph.D. dissertation, Cornell University. Published as Xerox PARC Technical Report CSL 90-1, February 1990.

[Deutsch 90] Alain Deutsch.
On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications (extended version).
Research Report LIX/RR/90/11, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France.
A conference-length version of this report appears in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[Ellis 86] John Ellis.
Bulldog: A Compiler for VLIW Architectures.
Ph.D. dissertation, Yale University. MIT Press, 1986.
Also available as Research Report 364, Yale University, Department of Computer Science.

[Fisher+ 84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau.
Parallel processing: A smart compiler and a dumb machine.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction,* published as *SIGPLAN Notices* 19(6), pages 37–47. Association for Computing Machinery, June 1984.

[Garner+ 88] R. Garner, *et al.*
Scaleable processor architecture (SPARC).
In *COMPCON*, pages 278–283. IEEE Computer Society, March 1988.

[Halfant+ 88] Mathew Halfant and Gerald Jay Sussman.
Abstraction in numerical methods.
In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* pages 1–7, July 1988.

[Harrison 77] William H. Harrison.
Compiler analysis of the value ranges for variables.
IEEE Transactions on Software Engineering SE-3(3):243–250, May 1977.

[Harrison 89] Williams Ludwell Harrison III.
The interprocedural analysis and automatic parallelization of Scheme programs.
Lisp and Symbolic Computation 2(3/4):179–396, October 1989.

[Hecht 77] Matthew S. Hecht.
Data Flow Analysis of Computer Programs.
American Elsevier, New York, 1977.

[Hill+ 86] Mark Hill, *et al.*
Design decisions in Spur.
COMPUTER, 19(11):8–22, November 1986.

[Hudak 86] Paul Hudak.
Collecting interpretations of expressions (preliminary version).
Research Report 497, Yale University, Department of Computer Science, August 1986.

[Hudak 86b] Paul Hudak.
A semantic model of reference counting and its abstraction (detailed summary).
In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363, August 1986.

[Hudak+ 86b] Paul Hudak and Jonathan Young.
Higher-order strictness analysis in untyped lambda calculus.
In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages,* pages 97–109, January 1986.

[Johnston 71] John B. Johnston.
The contour model of block structured processes.
In *Proceedings of a Symposium on Data Structures in Programming Languages,* published as *SIGPLAN Notices* 6(2), pages 55–82. Association for Computing Machinery, February 1971.

[Kaplan+ 80] Marc A. Kaplan and Jeffrey D. Ullman.
A scheme for the automatic inference of variable types.
JACM 27(1):128–145, January 1980.

[Kelsey 89] Richard Kelsey.
Compilation by Program Transformation.
Ph.D. dissertation, Yale University, May 1989. Research Report 702, Department of Computer Science.
A conference-length version of this dissertation appears in *POPL 89*.

[Kranz 88] David Kranz.
ORBIT: An Optimizing Compiler for Scheme.
Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science.
A conference-length version of this dissertation appears in *SIGPLAN 86* [Kranz+ 86].

[Kranz+ 86] David Kranz, *et al.*
ORBIT: An optimizing compiler for Scheme.
In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* published as *SIGPLAN Notices* 21(7), pages 219–233. Association for Computing Machinery, July 1986.

[Kranz+ 89] David A. Kranz, Robert H. Halstead, Jr. and Eric Mohr.
Mul-T: A high-performance parallel Lisp.
In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* published as *SIGPLAN Notices* 24(7), pages 81–90. Association for Computing Machinery, July 1989.

[Landin 64] Peter J. Landin.
The mechanical evaluation of expressions.
Volume 6, Computer Journal, January 1964.

[Lee 89] Peter Lee.
Realistic Compiler Generation.
Ph.D. dissertation, University of Michigan. MIT Press, 1989.

[Leroy 89] Xavier Leroy.
Efficient data representation in polymorphic languages.
In *Programming Language Implementation and Logic Programming 90*, P. Deransart and J. Małuszyński (editors). Lecture Notes in Computer Science, volume 456, Springer-Verlag 1990.

[Lucassen+ 88]
John M. Lucassen and David K. Gifford.
Polymorphic effect systems.
In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[Milner 85] Robin Milner.
The Standard ML core language.
Polymorphism II(2), October 1985.
Also ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland, March 1986.

[Milner+ 90] Robin Milner, Mads Tofte and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[Moon 85] David A. Moon.
Architecture of the Symbolics 3600.
In proceedings of the *12th Annual International Symposium on Computer Architecture,* pages 76–83. IEEE Computer Society, June 1985.

[Mosses+ 86] Peter D. Mosses and David A. Watt.
The use of action semantics.
Technical Report DAIMI PB-217, Computer Science Department, Aarhus University, Denmark.

[Mycroft 81] Alan Mycroft.
Abstract Interpretation and Optimizing Transformations for Applicative Programs.
Ph.D. dissertation, University of Edinburgh, 1981.

[Pfenning+ 90]
Frank Pfenning and Peter Lee.
Metacircularity in the polymorphic lambda-calculus.
To appear in *Theoretical Computer Science*.

[Pleban 81] Uwe Pleban.
Preexecution Analysis Based on Denotational Semantics.
Ph.D. dissertation, University of Kansas, May 1981.

[Rees+ 82] Jonathan A. Rees and Norman I. Adams IV.
T: A dialect of Lisp or, Lambda: The ultimate software tool.
In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming,* pages 114–122, August 1982.

[Rees+ 84] Jonathan A. Rees, Norman I. Adams IV and James R. Meehan.
The T Manual.
4th edition, Yale University, Department of Computer Science, January 1984.

[Rees+ 86] J. Rees and W. Clinger (editors).
The revised^3 report on the algorithmic language Scheme.
SIGPLAN Notices 21(12):37–79, December 1986.

[Reynolds 72] John C. Reynolds.
Definitional interpreters for higher-order programming languages.
In *Proceedings of the ACM Annual Conference,* pages 717–740, August 1972.

[Reynolds 78] John C. Reynolds.
Syntactic control of interference.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[Riesbeck] Online documentation for the T3 implementation of the Yale `loop` package is distributed by its current maintainer: Prof. Chris Riesbeck, Yale University, Department of Computer Science. (`riesbeck@yale.edu`).

[Schmidt 86] David A. Schmidt.
Denotational Semantics: a Methodology for Language Development.
Allyn and Bacon, Boston, 1986.

[Sharir$^+$ 81] Micha Sharir and Amir Pnueli.
Two approaches to interprocedural data flow analysis.
Chapter 7 of *Program Flow Analysis: Theory and Applications*, Steven S. Muchnick and Neil D. Jones (editors), Prentice-Hall, 1981.

[Steele 76] Guy L. Steele Jr.
Lambda: The ultimate declarative.
AI Memo 379, MIT AI Lab, November 1976.

[Steele 78] Guy L. Steele Jr.
RABBIT: A Compiler for SCHEME.
Technical Report 474, MIT AI Lab, May 1978.

[Steele 90] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, Maynard, Mass., second edition 1990.

[Steenkiste 87] Peter Steenkiste.
Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.
Ph.D. dissertation, Stanford University, March 1987. Technical Report 87-324, Computer Systems Laboratory.

[Stoy 77] Joseph E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
MIT Press, 1977.

[Tenenbaum 74]
A. Tenenbaum.
Type determination for very high level languages.
Report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974.

[Vegdahl+ 89] Steven R. Vegdahl and Uwe F. Pleban.
The runtime environment for Screme, a Scheme implementation on the 88000.
In *ASPLOS-III Proceedings* (Third International Conference on Architectural Support for Programming Languages and Operating Systems), pages 172–182, April 1989. Proceedings published as simultaneous issue of SIGARCH Computer Architecture News 17(2), April 1989; SIGOPS Operating Systems Review 23(special issue), April 1989; and SIGPLAN Notices 24(special issue), May 1989. Association for Computing Machinery.

[Waters 82] Richard C. Waters.
LETS: an expressional loop notation.
AI Memo 680, MIT AI Lab, October 1982.

[Waters 89] Richard C. Waters.
Optimization of series expressions, part I: User's manual for the series macro package.
AI Memo 1082, MIT AI Lab, January 1989.

[Waters 89b] Richard C. Waters.
Optimization of series expressions, part II: Overview of the theory and implementation.
AI Memo 1083, MIT AI Lab, January 1989.

[Wijngaarden 66]
A. van Wijngaarden.
Recursive definition of syntax and semantics.
In the proceedings of the IFIP Working Conference on Formal Language Description Languages, pages 13–24, 1964. Proceedings published as *Formal Language Description Languages for Computer Programming*, T. B. Steel, Jr. (editor). North-Holland, Amsterdam, 1966.

[Young+ 86] Jonathan Young and Paul Hudak.
Finding fixpoints on function spaces.
Research Report 505, Yale University, Department of Computer Science. December 1986.