The Semantics of Scheme Control-Flow Analysis

Olin Shivers

School of Computer Science
Carnegie Mellon
Pittsburgh, Pennsylvania 15213
Olin.Shivers@cs.cmu.edu

Abstract

This is a follow-on to my 1988 PLDI paper, "Control-Flow Analysis in Scheme" [9]. I use the method of abstract semantic interpretations to explicate the control-flow analysis technique presented in that paper.

I begin with a denotational semantics for CPS Scheme. I then present an alternate semantics that precisely expresses the control-flow analysis problem. I abstract this semantics in a natural way, arriving at two different semantic interpretations giving approximate solutions to the flow analysis problem, each computable at compile time. The development of the final abstract semantics provides a clear, formal description of the analysis technique presented in "Control-Flow Analysis in Scheme."

1 Introduction

1.1 Control-flow analysis

Scheme control-flow analysis (CFA) is a useful technique for compile-time analysis of the control-flow structure of Scheme programs (or, more generally, programs written in languages allowing first-class functions). In a previous paper [9], I introduced the technique, gave an algorithm for it, and demonstrated two example optimisations (induction-variable elimination and useless-variable elimination) that could be achieved with the results of the analysis. Other useful applications of control-flow analysis are type recovery [11], and copy, constant and lambda propagation [13]. The fundamental ideas of control-flow analysis have also been utilised in other work on functional programming languages [8, 2].

The basic technique for performing Scheme control-flow analysis consists of translating the Scheme program into a simple intermediate representation: continuation-passing style (CPS) Scheme with a primitive functional conditional operator and all side-effects to variables converted into side-effects to data-structures. After the CPS conversion, all transfers of control in the program — sequencing, iteration, conditional transfers, procedure call/return — are represented as tail-recursive procedure calls. Thus the problem of determining the control-flow structure of the program reduces to the problem of determining for each call site the set of all lambda expressions in the program that could be branched to from that call site.

To appear at the First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 1991, Yale University, New Haven, Conn. Also available as Technical Report CMU-CS-91-119, CMU School of Computer Science.

1.2 Non-standard abstract semantic interpretations

Non-standard abstract semantic interpretation is an elegant method for formally describing program analyses. Suppose we have a programming language L with a denotational semantics S, and we wish to determine some property X at compile time. Our first step is to develop an *alternate semantics* S_X for L that precisely expresses property X. That is, whereas semantics S might say the meaning of a program is a function "computing" the program's result value given its inputs, semantics S_X would say the meaning of a program is a function "computing" the property X on its corresponding inputs.

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 depend on the runtime inputs. The second step, then, is to *abstract* S_X to a new semantics, $\hat{S}_X$ which trades off 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.

For example, allow property X to be the set of all "useless variables" in a program, where a useless variable is one referenced only to compute values bound to other useless variables. Such variables, and the computations referencing them, can then be eliminated from the program without altering its result. Our alternate semantics S_X would map a program P to a function that "computes" P's useless-variable set. This semantics would probably be uncomputable, depending on perfect knowledge of the control-flow behavior of P. A useful, conservative abstraction $\hat{S}_X$ would be one that occasionally misses a truely useless variable, but never includes a useful variable in its result set.

The method of non-standard abstract semantic interpretation has several benefits. Since the analysis is expressed in terms of a formal semantics, it is possible to prove important properties about the analysis. In particular, we can prove that the abstract semantics $\hat{S}_X$ 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, the simple expression of an analysis in terms of abstract semantic interpretations helps clarify it. The abstract semantic interpretation method of program analysis has been applied to an array of program analyses [3, 5, 12, 4, 11].

In this paper, I will explicate Scheme control-flow analysis using this framework. I will show a series of semantics for CPS Scheme, beginning with the standard semantics, evolving through exact control analysis, and ending up with two different computable abstractions (with different cost/precision tradeoffs).

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

Figure 1: CPS Scheme Syntax

1.3 Notation

D^* is used to indicate all vectors of finite length over the set D. Functions are updated with brackets: $e\big[a \mapsto b,\ c \mapsto d\big]$ is the function mapping a to b, c to d, and everywhere else identical to function e. This notation is extended by taking an update standing by itself to imply an update to the appropriate bottom function $\bot$; hence [ ] is equivalent to $\bot$. Unused function variables are, by convention, subscripted with x, *e.g.*, e_x. The power set of D is $\mathbf{P}(D)$. Vectors are written $\langle a\ p\ z\rangle$. Lambda functions are sometimes written with a vector-destructuring syntax: the function $\lambda\,\langle a\ b\rangle$. *exp* takes a two element vector as its single argument, binding a to the first element, and b to the second. The ith element of vector v is written $v{\downarrow}i$. Functions with power-set ranges can be joined with the $\sqcup$ operator: $f \sqcup g = \lambda x.\ (f\ x) \cup (g\ x)$. 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 object is just a set, not a domain; following Reynolds [7], I attempt to introduce domains only where necessary in semantic constructions, avoiding "spurious values."

2 CPS Scheme

The practice of converting programs into continuation-passing style (CPS) as an intermediate representation for compilation has been discussed in several papers [15, 6, 1]. CPS can be summarised by stating that function calls are one-way transfers — they do not return. So a function call can be viewed as a `GOTO` that passes values. In this section, we will define a very simple language, called *CPS Scheme*, for expressing programs in written in this style.

The syntax of CPS Scheme is shown in figure 1. A program is a single lambda expression. Lambda expressions bind variables $v_1 \ldots v_n$; the body of a lambda expression must be a single call expression. There are two kinds of call expressions. A simple call expression is a function applied to a series of arguments. The function expression may only be a variable, a lambda or a primitive operation (primop). An argument expression may only be a variable, a lambda, or a constant. Notice that our syntax directly reflects the "function calls never return" prohibition of CPS: it is impossible to nest function calls, *e.g.*, `(+ (* c d) e)` is not syntactically legal, since `(* c d)` is not a legitimate argument expression in the `+` call. A `letrec` call expression is special syntax for establishing mutually recursive sets of functions. The lambda expressions l_i are evaluated, and the result functions bound to the corresponding variables f_i. The l_i are closed in an environment that includes the f_i. The inner call expression c of the `letrec` form is then evaluated in this environment. This is simply a CPS version of the Scheme `labels` or `letrec` form. It would be possible to eliminate the `letrec` form by including the `Y` operator as a primop; this approach has been adopted in other presentations [6, 9]. However, we would like to limit the arguments we are willing to circularly close to be lambda expressions. By elevating this restriction to the syntactic level, we will simplify the semantics equations, and sidestep the Y operator. There is no syntax for assigning variables in this language. If we wish to allow side effects, we can introduce appropriate primops to create and side-effect mutable data structures; assignments to variables in the source language can be converted into equivalent data structure side effects during the CPS conversion [6]. Finally, we'll assume that all variables are unique in a program — that is, no identifier is bound by more than one lambda expression.

It bears emphasizing that this rather minimal language is a useful intermediate representation for compiling higher-order languages such as Scheme. Variants of CPS Scheme have been used in several Scheme and ML compilers. A full discussion of the many advantages of CPS-based intermediate representations, however, is beyond the scope of this paper [1, 6, 15].

CPS Scheme has a very simple semantics. The semantic domains and functions are given in figure 2. There is a set of basic values, Bas, which consists of the integers and a special false value (I will follow traditional Lisp practice in assuming no special boolean type; anything not false is a true value). The value set D consists of the basic values and CPS Scheme functions. CPS Scheme functions are represented as functions from vectors of values to the answer set. The domain of answers Ans is the value set D plus a special error element denoting a run-time error, and a bottom element denoting non-termination[1]. An environment is a function from variables to values.

Note that an environment only maps to values in D — it will never map a variable to $\bot$ or error. This is one of the happy consequences of CPS conversion. Also note the absence of a store in this semantics. Side effects have been dropped completely from this semantics to simplify the presentation; they are not difficult to reinstate [10] once the basic methods outlined in this paper are understood.

$\mathcal{PR}$ maps a program to its result. It simply calls the $\mathcal{A}$ function to close its lambda ℓ in the empty environment [ ], and calls the result function on a one-element argument vector, which contains the terminal continuation. If the terminal continuation is ever applied to a one-element argument vector av, that element $av{\downarrow}1$ is the result of running the program. If av doesn't contain exactly one element, the program is aborted with a run-time error.

The $\mathcal{A}$ function evaluates function and argument expressions. It is defined by cases. A variable v is simply looked up in the environment. A constant k is passed to the constant function $\mathcal{K}$, which presumably maps numerals to their corresponding integers, and the

[1]The bottom element also allows Ans and, hence, $\text{D}^* \rightarrow \text{Ans}$ to be legitimate domains, necessary for the `letrec` expression to be well-defined. This sort of semantic fine detail is beyond the scope of this paper, and will not concern us further.

$$\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{D} &= \text{Bas} + (\text{D}^* \rightarrow \text{Ans}) \\
\text{Ans} &= (\text{D} + \{\text{error}\})_\perp \\
\text{Env} &= \text{VAR} \rightarrow \text{D}
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathcal{PR} &: \text{PR} \rightarrow \text{Ans} \\
\mathcal{C} &: \text{CALL} \rightarrow \text{Env} \rightarrow \text{Ans} \\
\mathcal{A} &: \text{ARG} \cup \text{FUN} \rightarrow \text{Env} \rightarrow \text{D} \\
\mathcal{K} &: \text{CONST} \rightarrow \text{D} \\
\mathcal{P} &: \text{PRIM} \rightarrow \text{D}
\end{aligned}$$

$$\mathcal{PR}\,\ell = \mathcal{A}\,\ell\,[\,]\left\langle \begin{array}{l} \lambda av.\ \text{length}(av) = 1 \ \longrightarrow\ av{\downarrow}1 \\ \qquad\ \ \text{otherwise error} \end{array}\right\rangle$$

$$\begin{aligned}
\mathcal{A}\,[\![v]\!]\,e &= e\,v \\
\mathcal{A}\,[\![k]\!]\,e_x &= \mathcal{K}\,k \\
\mathcal{A}\,[\![p]\!]\,e_x &= \mathcal{P}\,p \\
\mathcal{A}\,[\![(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]\,e &= \lambda av.\ \text{length}(av) = n \ \longrightarrow\ \mathcal{C}\,c\,e\big[v_i \mapsto av{\downarrow}i\big] \\
&\qquad\qquad \text{otherwise error}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}\,[\![(f\ \ a_1 \ldots a_n)]\!]\,e &= f' \notin \text{Bas} \ \longrightarrow\ f'\,\langle a'_1 \ldots a'_n\rangle \\
&\quad\ \text{otherwise error} \\
&\quad\ \text{where } f' = \mathcal{A}\,f\,e,\ \ a'_i = \mathcal{A}\,a_i\,e \\
\mathcal{C}\,[\![(\texttt{letrec}\ \ ((f_1\ \ l_1)\ldots)\ \ c)]\!]\,e &= \mathcal{C}\,c\,e' \\
&\quad\ \text{whererec } e' = e\big[f_i \mapsto \mathcal{A}\,l_i\,e'\big]
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}\,[\![\texttt{+}]\!] &= \lambda\,\langle a\ b\ c\rangle\,.\ \textit{bad argument} \ \longrightarrow\ \text{error} \\
&\qquad\qquad\quad \text{otherwise } c\,\langle a+b\rangle \\
\mathcal{P}\,[\![\texttt{if}]\!] &= \lambda\,\langle p\ c\ a\rangle\,.\ \textit{bad argument} \ \longrightarrow\ \text{error} \\
&\qquad\qquad\quad p \neq \text{false} \ \longrightarrow\ c\,\langle\rangle \\
&\qquad\qquad\quad \text{otherwise } a\,\langle\rangle
\end{aligned}$$

Figure 2: Standard Semantics

false identifier to the false value. The precise definition of $\mathcal{K}$ will not concern us further. A primop p is passed to the primop function $\mathcal{P}$. A lambda $(\lambda\ \ (v_1 \ldots v_n)\ \ c)$ is mapped by $\mathcal{A}$ to a function that takes as its argument an argument vector av. If av contains exactly as many elements as the lambda has formal parameters v_i, then the lambda's call body c is evaluated in an environment which is the closure environment e extended by mapping each v_i to the ith element in the argument vector. If, however, av has too few or too many elements to match up with the lambda's variables, the program is aborted, with the error value for result.

Note that it is always possible to evaluate a function or argument expression. Since function and argument expressions are so simple — variables, lambdas, primops, and constants — they can always be evaluated. This is why the range of $\mathcal{A}$ does not include $\perp$. Furthermore, the only possible case in which $\mathcal{A}$ might generate an error value lies in referencing an unbound variable. However, since CPS Scheme is lexically scoped, we can relegate this to syntax, and simply declare that programs containing unbound variable references are not syntactically legal CPS Scheme programs. Thus we can omit the error value as well from $\mathcal{A}$'s range.

The $\mathcal{C}$ function evaluates a call expression in a given environment. It is also defined by cases. In the simple function case, $\mathcal{C}$ evaluates the function expression and each of the argument expressions in the current environment. The argument values are packaged into an argument vector, which is passed to the function value. However, if the function expression evaluates to a non-function, the program is aborted with the error value for result. Given a `letrec` call expression, $\mathcal{C}$ establishes the inner environment by evaluating the lambda expressions l_i in the appropriate recursive environment e', and then evaluates the inner call c in the result environment.

The $\mathcal{P}$ function must map each primop in CPS Scheme to an appropriate function. For expository purposes, I present its definition for two values: the `+` function, to illustrate ordinary primitive functions, and the `if` function, to show a primitive function that determines control-flow.

The `+` primop denotes a function whose argument is a triple composed of two integers a and b, and a continuation function c. The integers are added, and the sum is packaged into a singleton argument vector which is passed to c. If the `+` primop is called on a "bad argument," the program is aborted with the error value for result. For the purposes of `+`, an argument is bad if it contains fewer than or more than three elements; if its first or second element is not an integer; or if its third element is not a function.

The `if` primop denotes a function whose argument is a triple composed of some value p and two continuation functions c and a. If the argument does not conform to this requirement, a runtime error is returned. If the predicate p is true (*i.e.*, anything but the false value), the consequent continuation c is called with no arguments (*i.e.*, the empty argument vector). Similarly, if p is false, the alternate continuation a is called.

$$\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{D} &= \text{Bas} + (\text{D}^* \to \text{VEnv} \to \text{Ans}) \\
\text{Ans} &= (\text{D} + \{\text{error}\})_\perp \\
\text{CN} & \quad \mathit{Contours} \\
\text{BEnv} &= \text{LAB} \to \text{CN} \\
\text{VEnv} &= \text{CN} \times \text{VAR} \to \text{D}
\end{aligned}
\qquad
\begin{aligned}
nb &: \to \text{CN} \\
\mathcal{PR} &: \text{PR} \to \text{Ans} \\
\mathcal{C} &: \text{CALL} \to \text{BEnv} \to \text{VEnv} \to \text{Ans} \\
\mathcal{A} &: \text{ARG} \cup \text{FUN} \to \text{BEnv} \to \text{VEnv} \to \text{D} \\
\mathcal{K} &: \text{CONST} \to \text{D} \\
\mathcal{P} &: \text{PRIM} \to \text{D}
\end{aligned}$$

$$\mathcal{PR}\,\ell = \mathcal{A}\,\ell\,[\,]\,[\,]\left\langle \lambda av\; e_x.\; \begin{array}{l} \text{length}(av) = 1 \;\longrightarrow\; av{\downarrow}1 \\ \text{otherwise error} \end{array} \right\rangle [\,]$$

$$\begin{aligned}
\mathcal{A}\,[\![v]\!]\;\epsilon\;e &= e\,\langle \epsilon(\textit{binder}\,v)\;v\rangle \\
\mathcal{A}\,[\![k]\!]\;\epsilon_x\;e_x &= \mathcal{K}\,k \\
\mathcal{A}\,[\![p]\!]\;\epsilon_x\;e_x &= \mathcal{P}\,p \\
\mathcal{A}\,[\![\ell\!:\!(\lambda\;\;(v_1 \ldots v_n)\;\;c)]\!]\;\epsilon\;e_x &= \lambda av\;e.\;\begin{array}[t]{l} \text{length}(av) = n \;\longrightarrow\; \mathcal{C}\,c\,\epsilon'\,e' \\ \text{otherwise error} \\ \text{where } b = nb \\ \qquad \epsilon' = \epsilon\big[\ell \mapsto b\big] \\ \qquad e' = e\big[\langle b\;v_i\rangle \mapsto av{\downarrow}i\big] \end{array}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}\,[\![(f\;\;a_1 \ldots a_n)]\!]\;\epsilon\,e &= \begin{array}[t]{l} f' \notin \text{Bas} \;\longrightarrow\; f'\,\langle a_1' \ldots a_n'\rangle\,e \\ \text{otherwise error} \\ \text{where } f' = \mathcal{A}\,f\;\epsilon\;e,\;\; a_i' = \mathcal{A}\,a_i\;\epsilon\;e \end{array} \\
\mathcal{C}\,[\![\ell\!:\!(\texttt{letrec}\;\;((f_1\;\;l_1)\ldots)\;\;c)]\!]\;\epsilon\,e &= \begin{array}[t]{l} \mathcal{C}\,c\,\epsilon'\,e' \\ \text{where } b = nb \\ \qquad \epsilon' = \epsilon\big[\ell \mapsto b\big] \\ \qquad e' = e\big[\langle b\;f_i\rangle \mapsto \mathcal{A}\,l_i\;\epsilon'\;e\big] \end{array}
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}\,[\![\texttt{+}]\!] &= \lambda\,\langle a\;b\;c\rangle\;e.\;\begin{array}[t]{l} \textit{bad argument} \;\longrightarrow\; \text{error} \\ \text{otherwise } c\,\langle a+b\rangle\,e \end{array} \\
\mathcal{P}\,[\![\texttt{if}]\!] &= \lambda\,\langle p\;c\;a\rangle\;e.\;\begin{array}[t]{l} \textit{bad argument} \;\longrightarrow\; \text{error} \\ p \neq \text{false} \;\;\;\;\longrightarrow\; c\,\langle\rangle\,e \\ \text{otherwise } a\,\langle\rangle\,e \end{array}
\end{aligned}$$

Figure 3: Factored Env Semantics

3 Factoring the environment

Before proceeding to the control-flow semantics, we must first develop a slight variant of our standard CPS Scheme semantics. This will make it easier to eventually abstract the semantics.

In the new semantics, called the factored-env semantics, we split the environment into two different structures: the *contour environment* and a global, shared *variable environment*. A contour environment ϵ maps a syntactic binding construct — a lambda or `letrec` call — to a *contour*. A contour is just some token — an integer will do — that serves to distinguish one binding instance from another. The variable environment maps a contour/variable pair $\langle b\;v\rangle$ to a value.

This model requires us to alter our syntax slightly, by requiring that all our expressions have unique labels, *e.g.*,

$$\ell\,\texttt{:(}\lambda\;\texttt{(x y) (x 3 y))}$$

has label ℓ. Labels are drawn from the syntactic set LAB. To avoid cluttering our code, we'll suppress labels whenever convenient. We define a syntactic function *binder* which maps a variable to the label of its binding lambda or `letrec` expression. Hence, in the example above, *binder*$[\![\texttt{x}]\!] = \ell$.

Our syntax augmented, we can now define our new semantics (figure 3). The nb function creates new binding contours. We assume that each call to nb produces a new, unused binding contour from CN. In this respect, nb serves a "gensym" role, and is not a properly defined function. It is clear, however, that we could easily define nb properly by modifying all the other functions in our factored-env semantics to pass around as an extra argument the set of binding contours already allocated; this set would serve as the argument to nb. Such modifications to the equations are straightforward, tedious, and obfuscatory; we will pass them by.

Since the variable environment is supposed to model a global table, all functions that could access or modify the environment must pass it around as an extra argument. CPS Scheme functions, for instance, are now represented by elements of $D^* \to \text{VEnv} \to \text{Ans}$ — they are invoked with the current variable environment as a second argument. It is a general feature of this semantics that as the computation progresses forwards, the updated variable environment is passed forwards in a tail-recursive fashion.

The $\mathcal{A}$ function has changed for the variable and lambda cases. In the variable case, $\mathcal{A}$ looks up the binding contour b introduced by the variable's binding lambda in the lexical contour environment. Then the contour/variable pair $\langle b\;v\rangle$ is used to fetch the appropriate

$$
\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{D} &= \text{Bas} + (\text{D}^* \rightarrow \text{LAB} \rightarrow \text{Ans}) \\
\text{Ans} &= \text{LAB} \rightarrow \mathbf{P}(\text{LAB} + \text{PRIM}) \\
\text{Env} &= \text{VAR} \rightarrow \text{D}
\end{aligned}
\qquad
\begin{aligned}
\mathcal{PR} &: \text{PR} \rightarrow \text{Ans} \\
\mathcal{C} &: \text{CALL} \rightarrow \text{Env} \rightarrow \text{Ans} \\
\mathcal{A} &: \text{ARG} \cup \text{FUN} \rightarrow \text{Env} \rightarrow \text{D} \\
\mathcal{K} &: \text{CONST} \rightarrow \text{D} \\
\mathcal{P} &: \text{PRIM} \rightarrow \text{D}
\end{aligned}
$$

$$\mathcal{PR}\,\ell = \mathcal{A}\,\ell\,[\,]\left\langle \lambda av_x\,\ell_c.\ \left[\ell_c \mapsto \{\ell_{\text{top}}\}\right]\right\rangle c_{\text{top}}$$

$$
\begin{aligned}
\mathcal{A}[\![v]\!]\,e &= e\,v \\
\mathcal{A}[\![k]\!]\,e_x &= \mathcal{K}\,k \\
\mathcal{A}[\![p]\!]\,e_x &= \mathcal{P}\,p \\
\mathcal{A}[\![\ell\text{:}\,(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]\,e &= \lambda av\,\ell_c.\ \left[\ell_c \mapsto \{\ell\}\right] \sqcup \text{length}(av) = n \longrightarrow \mathcal{C}\,c\,e\left[v_i \mapsto av{\downarrow}i\right] \\
&\qquad \text{otherwise } [\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}[\![c\text{:}\,(f\ \ a_1 \ldots a_n)]\!]\,e &= f' \notin \text{Bas} \longrightarrow f'\,\langle a'_1 \ldots a'_n\rangle\,c \\
&\quad \text{otherwise error} \\
&\quad \text{where } f' = \mathcal{A}\,f\,e,\ a'_i = \mathcal{A}\,a_i\,e \\
\mathcal{C}[\![\texttt{(letrec ((}f_1\ \ l_1\texttt{)...)}\ \ c\texttt{)}]\!]\,e &= \mathcal{C}\,c\,e' \\
&\quad \text{whererec } e' = e\left[f_i \mapsto \mathcal{A}\,l_i\,e'\right]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}[\![\texttt{+}]\!] &= \lambda\,\langle a\ b\ c\rangle\ \ell_c.\ \left[\ell_c \mapsto \{\texttt{+}\}\right] \sqcup \textit{bad argument} \longrightarrow [\,] \\
&\qquad \text{otherwise } c\,\langle a+b\rangle\,ic_{\texttt{+},\ell_c} \\
\mathcal{P}[\![\texttt{if}]\!] &= \lambda\,\langle p\ c\ a\rangle\ \ell_c.\ \left[\ell_c \mapsto \{\texttt{if}\}\right] \sqcup \textit{bad argument} \longrightarrow [\,] \\
&\qquad p \neq \text{false} \longrightarrow c\,\langle\rangle\,ic^1_{\texttt{if},\ell_c} \\
&\qquad \text{otherwise } a\,\langle\rangle\,ic^2_{\texttt{if},\ell_c}
\end{aligned}
$$

Figure 4: Exact Control-Flow Semantics

value from the variable environment. Factoring the environment means that $\mathcal{A}$ now closes a lambda expression in the current contour environment. It produces a function, which, when called, creates a new binding contour b with nb, augments the contour environment ϵ with the new contour, and updates the variable environment e, binding parameters $v_1 \ldots v_n$ to the elements of argument vector av in the new contour. Since b is a new contour, these bindings won't collide with those made by other calls to this lambda.

Similarly, $\mathcal{C}$ introduces a new contour b into the binding environment ϵ whenever execution proceeds through a `letrec` call, and binds the lambda expressions l_i in the new environment. It is interesting to note in passing that this formulation removes the circularity from the `letrec` environment update (since l_i is a lambda expression, $\mathcal{A}$ ignores e).

4 Semantics of exact control-flow analysis

Now we may turn to control-flow analysis. To repeat an earlier point, the point of using a CPS-based intermediate representation is that all control transfers are represented with the same mechanism: tail-recursive procedure call. So the control-flow analysis problem for CPS Scheme is to find a function $\gamma : \text{LAB} \rightarrow \mathbf{P}(\text{LAB} + \text{PRIM})$ that given the label ℓ_c of a call expression, will return $\gamma(\ell_c)$, the set of all lambdas and primops called from ℓ_c. We will call such a function a *call cache.*

This is easy to cast as an alternate semantics: the meaning of a program is its cache function. We can compute this cache function by taking the standard semantics and "instrumenting" it to record all the calls that happen during program execution. Instead of returning the actual value computed by the program, the new, non-standard semantics returns the cache constructed by the instrumentation.

Figure 4 presents the new, non-standard semantics. It is initially developed as an unfactored-env semantics. The answer domain Ans is the domain of cache functions; its bottom element $\bot_{\text{Ans}}$ is simply the function $\lambda x.\ \emptyset$. CPS Scheme functions are now represented by functions that take an extra argument: the label ℓ_c of the site from which they were called. The function augments the answer cache by including its CPS Scheme lambda l in the set of lambdas called from ℓ_c. The meaning of call expressions is changed accordingly to pass along to the called function f' the call site c from which it has been called.

The alterations to the primops are similarly straightforward. The `+` primop updates the cache to indicate where it has been called from, and passes to its continuation c a marker pseudo-label $ic_{\texttt{+},\ell_c}$ to indicate that the continuation was called from its internal call site. The `if` primop is similar, its main variation being the use of two different internal call sites, one to mark consequent calls ($ic^1_{\texttt{if},\ell_c}$), and one to mark alternate calls ($ic^2_{\texttt{if},\ell_c}$).

The $\mathcal{PR}$ function starts the program by calling the top function, passing it a designated call-site label c_{top} to mark the top level call. The top level continuation has a designated label ℓ_{top} to indicate calls to it; when it is called, it records the final call and stops the program.

$$\begin{aligned}
\text{Ans} &= \text{LAB} \rightarrow \mathbf{P}(\text{LAB} + \text{PRIM}) \\
\text{D} &= \mathbf{P}(\text{D})^* \rightarrow \text{VEnv} \rightarrow \text{LAB} \rightarrow \text{Ans} \\
\text{CN} & \quad \mathit{Contours} \\
\text{BEnv} &= \text{LAB} \rightarrow \text{CN} \\
\text{VEnv} &= \text{CN} \times \text{VAR} \rightarrow \mathbf{P}(\text{D})
\end{aligned}
\qquad
\begin{aligned}
nb &: \rightarrow \text{CN} \\
\mathcal{PR} &: \text{PR} \rightarrow \text{Ans} \\
\mathcal{C} &: \text{CALL} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightarrow \text{Ans} \\
\mathcal{A} &: \text{ARG} \cup \text{FUN} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightarrow \mathbf{P}(\text{D}) \\
\mathcal{P} &: \text{PRIM} \rightarrow \text{D}
\end{aligned}$$

$$\begin{aligned}
\mathcal{PR}\,\ell &= f\left\langle\left\{\,\lambda av_x\, e_x\, \ell_c.\ \left[\ell_c \mapsto \{\ell_{\text{top}}\}\right]\ \right\}\right\rangle [\,]\, c_{\text{top}} \\
&\quad \text{where } \{f\} = \mathcal{A}\,\ell\,[\,]\,[\,]
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}\,[\![v]\!]\ \epsilon\ e &= e\,\langle\epsilon(\mathit{binder}\,v)\ v\rangle \\
\mathcal{A}\,[\![k]\!]\ \epsilon_x\ e_x &= \emptyset \\
\mathcal{A}\,[\![p]\!]\ \epsilon_x\ e_x &= \{\mathcal{P}\,p\} \\
\mathcal{A}\,[\![\ell\colon (\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]\ \epsilon\ e_x &= \left\{\begin{array}{l}
\lambda av\ e\ \ell_c.\ \left[\ell_c \mapsto \{\ell\}\right] \sqcup\ \text{length}(av) = n \ \longrightarrow\ \mathcal{C}\ c\ \epsilon'\ e' \\
\qquad\qquad\qquad\qquad\qquad \text{otherwise } [\,] \\
\qquad\qquad \text{where } b = nb \\
\qquad\qquad\qquad \epsilon' = \epsilon\left[\ell \mapsto b\right] \\
\qquad\qquad\qquad e' = e \sqcup \left[\langle b\ v_i\rangle \mapsto av{\downarrow}i\right]
\end{array}\right\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}\,[\![c\colon (f\ \ a_1 \ldots a_n)]\!]\ \epsilon\ e &= \bigsqcup \{f'\,\langle a'_1 \ldots a'_n\rangle\ e\ c \mid f' \in \mathcal{A}\ f\ \epsilon\ e\} \\
&\quad \text{where } a'_i = \mathcal{A}\ a_i\ \epsilon\ e \\
\mathcal{C}\,[\![\ell\colon \texttt{(letrec ((}f_1\ \ l_1\texttt{)...)}\ \ c\texttt{)}]\!]\ \epsilon\ e &= \mathcal{C}\ c\ \epsilon'\ e' \\
&\quad \text{where } b = nb \\
&\qquad\quad \epsilon' = \epsilon\left[\ell \mapsto b\right] \\
&\qquad\quad e' = e \sqcup \left[\langle b\ f_i\rangle \mapsto \mathcal{A}\ l_i\ \epsilon'\ e\right]
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}\,[\![\texttt{+}]\!] &= \lambda\,\langle a\ b\ c\rangle\ e\ \ell_c.\ \left[\ell_c \mapsto \{\texttt{+}\}\right] \sqcup\ \textit{bad argument}\ \longrightarrow\ [\,] \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{otherwise} \bigsqcup\left\{c'\,\langle\emptyset\rangle\ e\ ic_{\texttt{+},\ell_c} \mid c' \in c\right\} \\
\mathcal{P}\,[\![\texttt{if}]\!] &= \lambda\,\langle p\ c\ a\rangle\ e\ \ell_c.\ \left[\ell_c \mapsto \{\texttt{if}\}\right] \sqcup \left(\begin{array}{l}
\textit{bad argument}\ \longrightarrow\ [\,] \\
\text{otherwise} \bigsqcup\left\{c'\,\langle\rangle\ e\ ic^1_{\texttt{if},\ell_c} \mid c' \in c\right\} \\
\qquad\qquad \sqcup \bigsqcup\left\{a'\,\langle\rangle\ e\ ic^2_{\texttt{if},\ell_c} \mid a' \in a\right\}
\end{array}\right)
\end{aligned}$$

Figure 5: Control-Flow Semantics with ambiguous if and factored env

5 Abstracting the control-flow analysis semantics

Now that we've defined our control-flow analysis semantics, we have a formal description of the control-flow problem. The next step is to abstract our semantics to a computable approximate semantics that is useful but safe. Since we do not in general know at compile time which way a conditional branch will go, we must abstract away conditional dependencies. (Had we included i/o in our original exact control-flow semantics, this would be the appropriate time to remove those dependencies, as well.) While we are abstracting, we'll go ahead and factor the environment as well, which will be useful in the next section. The result ambiguous-if factored-env semantics is presented in figure 5.

We have introduced three major changes into our new semantics. First, the environment has been factored. This is essentially identical to the factoring performed upon the standard CPS Scheme semantics in section 3. Second, the `if` primop now "branches both ways." That is, the caches arising from both the consequent and alternate continuations are computed; these are joined together to give the result cache returned by the `if` primop. Removing this data dependency has a further consequence: the semantics no longer needs the basic value domain Bas. Since our semantics concerns itself solely with control flow, the only values that need to be considered are those representing CPS Scheme functions. The final change is to arrange for argument and function expressions to evaluate to *sets* of values, instead of simple values. This is actually an isomorphic shift, since all the sets that result in the new formulation are singleton sets. The extra machinery will come in useful in the next approximation, however, since the semantics now tolerates ambiguity in argument evaluation: if a function expression can only be determined to lie in some set, the $\mathcal{C}$ function will find the call caches resulting from calling all the functions f' in that set, and join them together to form the result cache. Also, note that because of the shift to value sets, variable environment updates are now performed with join operations, *e.g.*, $e' = e \sqcup \left[\langle b\ v_i\rangle \mapsto av{\downarrow}i\right]$, with bottom element $\bot_{\text{VEnv}} = \lambda x.\ \emptyset$.

$$\begin{aligned} \text{Ans} &= \text{LAB} \to \mathbf{P}(\text{LAB} + \text{PRIM}) \\ \text{D} &= \mathbf{P}(\text{D})^* \to \text{VEnv} \to \text{LAB} \to \text{Ans} \\ \text{VEnv} &= \text{VAR} \to \mathbf{P}(\text{D}) \end{aligned}$$

$$\begin{aligned} \mathcal{PR} &: \text{PR} \to \text{Ans} \\ \mathcal{C} &: \text{CALL} \to \text{VEnv} \to \text{Ans} \\ \mathcal{A} &: \text{ARG} \cup \text{FUN} \to \text{VEnv} \to \mathbf{P}(\text{D}) \\ \mathcal{P} &: \text{PRIM} \to \text{D} \end{aligned}$$

$$\begin{aligned} \mathcal{PR}\,\ell = {} & f\left\langle\left\{\,\lambda av_x\; e_x\; \ell_c.\ \left[\ell_c \mapsto \{\ell_{\text{top}}\}\right]\,\right\}\right\rangle [\,]\; c_{\text{top}} \\ & \text{where } \{f\} = \mathcal{A}\,\ell\,[\,] \end{aligned}$$

$$\begin{aligned} \mathcal{A}\,[\![v]\!]\; e \ &= e\; v \\ \mathcal{A}\,[\![k]\!]\; e_x &= \emptyset \\ \mathcal{A}\,[\![p]\!]\; e_x &= \{\mathcal{P}\,p\} \end{aligned}$$

$$\mathcal{A}\,[\![\ell\text{:}\,(\lambda\ \ (v_1 \ldots v_n)\ \ c)]\!]\; e_x = \left\{\begin{array}{c} \lambda av\; e\; \ell_c.\ \left[\ell_c \mapsto \{\ell\}\right] \sqcup\ \text{length}(av) = n \ \longrightarrow\ \mathcal{C}\; c\; e' \\ \text{otherwise } [\,] \\ \text{where } e' = e \sqcup \left[v_i \mapsto av{\downarrow}i\right] \end{array}\right\}$$

$$\begin{aligned} \mathcal{C}\,[\![c\text{:}\,(f\ \ a_1 \ldots a_n)]\!]\; e = {} & \bigsqcup \{f'\,\langle a_1' \ldots a_n'\rangle\; e\; c \mid f' \in \mathcal{A}\; f\; e\} \\ & \text{where } a_i' = \mathcal{A}\; a_i\; e \end{aligned}$$

$$\begin{aligned} \mathcal{C}\,[\![\ell\text{:}\,(\texttt{letrec}\ \ ((f_1\ \ l_1)\ldots)\ \ c)]\!]\; e = {} & \mathcal{C}\; c\; e' \\ & \text{where } e' = e \sqcup \left[f_i \mapsto \mathcal{A}\; l_i\; e\right] \end{aligned}$$

$$\begin{aligned} \mathcal{P}\,[\![\texttt{+}]\!] = \lambda\,\langle a\ b\ c\rangle\; e\; \ell_c.\ \left[\ell_c \mapsto \{\texttt{+}\}\right] \sqcup {} & \textit{bad argument} \ \longrightarrow\ [\,] \\ & \text{otherwise} \bigsqcup \left\{c'\,\langle\emptyset\rangle\; e\; ic_{\texttt{+},\ell_c} \mid c' \in c\right\} \end{aligned}$$

$$\mathcal{P}\,[\![\texttt{if}]\!] = \lambda\,\langle p\ c\ a\rangle\; e\; \ell_c.\ \left[\ell_c \mapsto \{\texttt{if}\}\right] \sqcup \left(\begin{array}{l} \textit{bad argument} \ \longrightarrow\ [\,] \\ \text{otherwise} \bigsqcup \left\{c'\,\langle\rangle\; e\; ic^1_{\texttt{if},\ell_c} \mid c' \in c\right\} \\ \qquad\qquad \sqcup \bigsqcup \left\{a'\,\langle\rangle\; e\; ic^2_{\texttt{if},\ell_c} \mid a' \in a\right\} \end{array}\right)$$

Figure 6: 0CFA

6 Computable control-flow analysis semantics

The problem with the previous semantics, abstracted though it may be, is that it is difficult to compute the fixed-point cache for a given program because the environment structure is infinite. Consider the following expression (written in full Scheme, not CPS Scheme, for clarity):

```
(letrec ((loop (λ (f)
                 (loop (λ (n) (* 2 (f n)))))))
   (loop (λ (m) 1)))
```

The variable `f` is bound to an infinite set of functions

$$\left\{\lambda x.\, 2^i \mid i \geq 0\right\}.$$

So it is difficult for any propagation-based fixed-point algorithm to know when to stop propagating.

There are only a finite number of lambda expressions in a given program; the infinite sets of functions 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.

If we examine the abstract semantics developed in the previous section (figure 5), we can see that the infinite environment structure is built with all the calls to the nb function in the $\mathcal{C}$ and $\mathcal{A}$ functions. If we replaced each call $b = nb$ with $b = \ell$, we would then fold all contours created by a given lambda expression together, and our infinite environment set would collapse into a finite, manageable set. This is precisely 0CFA, the Zeroth-Order Control-Flow Analysis technique presented in my PLDI '88 paper. As a final figure, I present the resultant 0CFA semantics in figure 6 (note that this is the degenerate contour environment case, and so this artifact has disappeared entirely).

An alternative approximation, only briefly mentioned as 1CFA in the earlier paper, is to distinguish contours created by calling a lambda from different call sites. Suppose, for example, that some lambda `(λ (x) ...)` is called from two different call sites c_1 and c_2. In 1CFA, the values bound to `x` by calls from c_1 are kept distinct from the values bound to `x` by calls from c_2. This yields a tighter, higher-precision analysis.

We are now in a position to precisely express this approximation: replace the call to nb in $\mathcal{A}$ with ℓ_c, the call site from which the lambda was called: $b = \ell_c$. Since there are only a finite number of call sites, the environment structure this engenders is still finite in size, and hence our caches are still computable. However, the finer granularity (or increased environment structure size) will cause the cache to be more expensive to compute.

Thus, 0CFA and 1CFA allow us to trade off compile-time efficiency for compile-time precision of analysis.

Now that we've abstracted the control-flow analysis, the reason for factoring the environment should be clear. Factoring the environment exposed the binding mechanisms that gave rise to the infinite environment structure. Abstracting the contours and merging bindings was the critical step that allowed us to reduce this infinite structure to a finite, computable one.

7 Implementation

I have written a prototype implementation of 1CFA; it is a straightforward translation of the semantics into Scheme code. The recursions in the semantics equations are terminated with a variant of Young and Hudak's memoised pending analysis [16]. The type recovery analysis mentioned in section 1 is built on top of this implementation. The prototype implementation uses a modified copy of the ORBIT compiler's front end to produce CPS Scheme code from programs written in full Scheme. The implemented 1CFA semantics extends the semantics presented in this paper to include side effects, external procedures and external calls. In addition, it statically separates user procedures from continuations introduced by the CPS conversion. This last point is worth briefly discussing.

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.

Running interpreted, the 1CFA implementation is able to analyse small examples (such as `fact` or `delq`) in about a half second on a DECstation 3100 PMAX. This is quick enough that I have not bothered to either compile the code or tune the simple algorithm and data structures.

Type recovery is a particularly interesting optimisation from the semantics point of view. I managed to implement induction-variable elimination and useless-variable elimination using an early *ad hoc* control-flow analysis algorithm [9], before I cast CFA into the non-standard abstract semantic interpretations framework presented in this paper. I could not have done so with type recovery analysis. Due to its dependence on sophisticated environment analysis, its design depended on the guidance of the semantics presented in this paper. The semantic definition of control-flow analysis has proved to be a valuable engineering tool for delivering useful Scheme program optimisations.

Detailed discussion of the CFA implementations and the optimisations built on them is beyond the scope of this paper; they are treated elsewhere [10, 11, 13, 14].

8 Discussion

A note on 0CFA that may be of interest to those who have read the PLDI '88 paper: The algorithm I presented in that paper was fundamentally different from the semantics presented in this paper in one important respect. In this paper, the environment information is propagated along paths through the control structure of the program. That is, when we determine that some lambda's variable `v` can be bound to some new function, we pass that information along to the lambda's call body, who passes it to the functions it calls, and so forth. Eventually, this information may propagate to a reference to `v`, where it will used.

In the algorithm presented in the early paper, the information propagates along paths through the environment structure of the program. That is, once we determine that `v` can be bound to some new function, we jump straight to all references to `v`, and propagate from there. This, of course, saves time, and allows for simpler convergence tests.

While this approach is correct, it does not generalise. In 1CFA, we allow multiple distinct contours over a single lambda. So we can't just propagate forward from all references to `v` — the environment structure determining propagations must be established by following the control-flow paths.

This difficulty arises from the power of lambda: it provides both environment and control structure. In 0CFA, the environment structure collapses into the degenerate case exploited by the early algorithm.

9 Conclusion

My chief purpose in writing this paper is to show a formal description of the Scheme control-flow analysis problem. This description is useful for several reasons:

- It leads us to useful computable approximations.
- The semantic description should give the reader a detailed, rigorous understanding of the Scheme control-flow analysis problem and its approximate solutions.
- Because it is a formal description, grounded in the semantics of Scheme, it can serve as a basis for proving formal properties of the analysis, its connection to the standard Scheme semantics, and the correctness of program optimisations based on it. We would like to prove that the semantics are all well-defined; that the approximate CFA semantics is a conservative approximation of the exact CFA semantics; and that the approximate semantics is computable. Such proofs are beyond the scope of this paper, but they can be found in my dissertation [10].
- It is a description which helps the Scheme compiler writer to develop and implement useful program optimisations, such as type recovery. Hopefully, compiler writers can use this description to design and implement their own optimisations.

Control-flow analysis is an important tool for developing analyses and optimisations for higher-order programming languages such as Scheme. As such, it is too important to exist without a solid theoretical foundation. The aim of this paper is to sketch out the structure of that foundation.

10 Acknowledgments

Refereeing conference papers is a tedious job. The PEPM referees gave my extended abstract a careful reading and provided many thoughtful suggestions that improved the final paper. I am grateful for their efforts. John Reynolds steered me through the (forthcoming) proofs of semantic correctness. Dana Scott helped me with a difficult domain construction. My advisor, Peter Lee, substantially improved an early draft of this paper. Olivier Danvy supplied several useful references and his usual infectious enthusiasm.

References

[1] Andrew 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.

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *ESOP '90*, Neil Jones (editor). Springer-Verlag, May 1990.

[3] 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 Annual Symposium on Principles of Programming Languages*, pages 238–252. Association of Computing Machinery, 1977.

[4] Paul Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986.

[5] Paul Hudak and Adrienne Bloss. Variations on strictness analysis. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, August 1986.

[6] 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.

[7] John Reynolds, School of Computer Science, CMU. Personal communication.

[8] Peter Sestoft. Replacing function parameters by global variables. Master's Thesis, University of Copenhagen, 1988. Student report 88-7-2, DIKU. A conference-length version of this thesis appears in the *FPCA '89 Conference Proceedings*, pages 39–53, September 1989.

[9] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988. Also available as Technical Report ERGO-88-60, CMU School of Computer Science, Pittsburgh, Penn.

[10] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. Dissertation, CMU. (Forthcoming)

[11] Olin Shivers. Data-flow analysis and type recovery in Scheme. Technical Report CMU-CS-90-115. CMU School of Computer Science, Pittsburgh, Penn., March 1990. Also to appear in *Topics in Advanced Language Implementation*, Peter Lee (editor), MIT Press.

[12] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1991. To appear in *SIGPLAN Notices*. Also available as Technical Report CMU-CS-91-119, CMU School of Computer Science, Pittsburgh, Penn. An early version was available as Technical Report ERGO-90-090.

[13] Olin Shivers. Super-β: Copy, constant, and lambda propagation in Scheme. Working note #3, May 1990.

[14] Olin Shivers. Useless-variable elimination. Working note #2, April 1990.

[15] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Technical Report 474, MIT AI Lab, May 1978.

[16] Jonathan Young and Paul Hudak. Finding fixpoints on function spaces. Research Report 505, Yale University, Department of Computer Science. December 1986.