A Practical and Flexible Flow Analysis for Higher-Order Languages

J. Michael Ashley
University of Kansas, Lawrence, Kansas
and
R. Kent Dybvig
Indiana University, Bloomington, Indiana

A flow analysis collects data-flow and control-flow information about programs. A compiler can use this information to enable optimizations. The analysis described in this article unifies and extends previous work on flow analyses for higher-order languages supporting assignment and control operators. The analysis is abstract interpretation-based and is parameterized over two polyvariance operators and a projection operator. These operators are used to regulate the speed and accuracy of the analysis. An implementation of the analysis is incorporated into and used in a production Scheme compiler. The analysis can process any legal Scheme program without modification. Others have demonstrated that a 0CFA analysis can enable optimizations, but a 0CFA analysis is $O(n^3)$. An $O(n)$ instantiation of our analysis successfully enables the optimization of closure representations and procedure calls. Experiments with the cheaper instantiation show that it is as effective as 0CFA for these optimizations.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: abstract interpretation, higher-order languages

1. INTRODUCTION

A flow analysis for a higher-order language collects data-flow and control-flow information about programs in the language. The information collected can be used to drive program transformations such as partial evaluation [Jones et al. 1993] and closure conversion [Steckler and Wand 1997] as well as compiler optimizations such as type recovery [Shivers 1991] and the selection of closure representations [Shao and Appel 1994].

In lexically-scoped, higher-order languages a procedure is a first-class value represented as a closure. A closure is a data structure that encapsulates a procedure's code and lexical environment. At a call site the operator is an arbitrary expression

This work was partially supported by NSF Grants CCR-9623753, CDA-9401021, and CDA-9312614.
A preliminary version of this article was presented at the 1996 ACM Symposium on Principles of Programming Languages.
Authors addresses: J.M. Ashley, Electrical Engineering and Computer Science Department, Snow Hall 415, Lawrence, KS 66045; R.K. Dybvig, Department of Computer Science, Lindley Hall 215, Bloomington, IN 47405.

that must evaluate to a closure at run time. Since it is an arbitrary expression the operator is generally unknown at compile time.

Flow analysis of higher-order languages is therefore fundamentally different from the analysis of first-order languages. A flow analysis for a first-order language first constructs a control-flow graph from the text of the program and then uses it to compute data-flow information. A flow analysis for a higher-order language, on the other hand, must simultaneously compute control- and data-flow information since the operator at a call site must be determined from data-flow information.

Several abstract interpretation-based approaches have been applied to the problem of flow-analyzing higher-order languages [Ayers 1993; Harrison III 1989; Jagannathan and Weeks 1995; Shivers 1991; Yi and Harrison III 1993]. Each approach addresses one or more of the following important aspects of the problem:

—accurate treatment of mutable data structures,

—use of type tests to constrain abstract values,

—use of polyvariance to increase accuracy, and

—use of projection (widening) to accelerate convergence to a fixpoint.

This article develops a parameterized analysis that addresses all of these aspects. It unifies and extends the prior research to synthesize a framework parameterized over the accuracy and speed of the analysis. Compared to other frameworks, it can describe a wider range of analyses that vary in accuracy and speed.

The analysis' implementation is the first truly practical flow analysis for higher-order programming languages. It is used in a production compiler for Scheme, and the analysis can process any Scheme program without modification. Furthermore, while most work on projection operators has been theoretical, the implementation incorporates a practical projection operator to enforce an $O(n)$ complexity bound on the analysis.

The analysis is used to enable procedure call and closure representation optimizations. Since the operator of a call is generally unknown until run time, procedure call overhead is higher than in first-order languages. The flow analysis can be used to identify the dataflow value of each operator in the program and use that information to generate more efficient code for calls. In some cases a call optimization allows the environment of the enclosing procedure to be reduced. This enables the compiler to reduce the size of or to eliminate the enclosing procedure's closure. The optimizations work together in that call optimization may enable closure optimization which may in turn enable more call optimization.

The optimizations justified by the flow analysis yield speedups of up to 16% on a standard set of benchmarks. Furthermore, we compared two instantiations of the analysis: a 0CFA [Shivers 1988] instantiation and a linear-time *sub-0CFA* instantiation. Our results show that the sub-0CFA analysis is almost as good as the 0CFA analysis for enabling these optimizations.

The rest of the paper is organized as follows. Section 2 develops the analysis by first giving a collecting operational semantics for a normalized Scheme language. The analysis is then given as a simple and intuitive abstraction of the collecting semantics. Section 3 discusses the implementation of the analysis. Section 4 gives results on the cost of two instantiations of the analysis framework and their use-

$$
\begin{array}{rcl}
M & = & P \mid (\texttt{cons}\ M_1\ M_2) \mid (\texttt{car}\ M_1) \mid (\texttt{letrec}\ ((\texttt{v}\ P))\ M_2) \mid \\
 & & (\texttt{call/cc}\ M_1) \mid (\texttt{if}\ (\texttt{pair?}\ M_1)\ M_2\ M_3) \mid (\texttt{begin}\ (\texttt{set!}\ \texttt{v}\ M_1)\ M_2) \mid \\
 & & (M_0\ \ldots\ M_n) \\
P & = & \texttt{c} \mid \texttt{v} \mid (\texttt{lambda}\ (\texttt{v}_1\ \ldots\ \texttt{v}_n)\ M) \\
\texttt{c} & \in & \textit{Constants} \\
\texttt{v} & \in & \textit{Vars}
\end{array}
$$

Fig. 1: The core Scheme language *CS*.

$$
\begin{array}{rcl}
A & = & (\texttt{let}\ ((\texttt{v}\ N))\ A) \mid (\texttt{letrec}\ ((\texttt{v}\ N))\ A) \mid (\texttt{pair?}\ N\ A_1\ A_2) \mid \\
 & & (\texttt{set!}\ \texttt{v}\ N\ A) \mid (N_0\ N_1\ \ldots\ N_n) \mid \texttt{halt} \\
N & = & \texttt{c} \mid \texttt{v} \mid (\texttt{lambda}_\eta\ (\texttt{v}_1\ \ldots\ \texttt{v}_n)\ A) \mid (\texttt{cons}\ \texttt{v}_0\ \texttt{v}_1) \mid (\texttt{car}\ N) \\
\texttt{c} & \in & \textit{Constants} \\
\texttt{v},\texttt{k} & \in & \textit{Vars} \\
\eta & \in & \textit{Tags}
\end{array}
$$

Fig. 2: The normalized Scheme language *NCS*.

fulness for enabling optimizations. Section 5 discusses related work, and Section 6 gives conclusions.

2. THE ANALYSIS

The analysis operates over closed programs in the core language given in Figure 1. It is a subset of Scheme with a restricted `letrec` form and some representative primitives added. While simple, the analysis developed around this core language can be extended to all of Scheme or ML, including the general `letrec` form, multiple return values, variable arity procedures, and programs with free variables.

The language could be given a semantics directly, but it is useful to work instead with a normalized language. The normalized language is a hybrid language similar to continuation-passing style (CPS) and A-normal form [Flanagan et al. 1993]. The grammar for this language is given in Figure 2. Terms are elements of the grammar defined by the nonterminal A. A subexpression in head position is a simple expression defined by the nonterminal N. The normalized language is similar to A-normal form in that intermediate values are named and evaluation order is made explicit. In addition, the car and cdr components of a pair are explicitly named. It resembles continuation-passing style in that the continuations of procedure calls and conditionals are represented as source-level procedures. The effect of introducing continuations at these points is that joins in the control-flow graph are represented uniformly as procedure entry. Finally, lambda expressions are tagged with elements from a set *Tags*, and a `halt` expression is introduced into the language. Both are used in the operational semantics given to *NCS*.

A term in *CS* is converted to an equivalent term in *NCS* via the translation function $\mathcal{C}[\![]\!]$.

$$\mathcal{C}[\![M]\!] = [\![M]\!]\lambda n.(\texttt{let}\ ((\texttt{v}\ n))\ \texttt{halt})$$

The auxiliary function $[\![]\!]$ is defined in Figure 3. In both definitions, all source terms are typeset upright and metaterms are typeset italicized. The definitions are two-level definitions [Nielson and Nielson 1992]. Since they are syntax-translation

$$[\![]\!] \;:\; M \to (N \to A) \to A$$

$$\begin{aligned}
[\![\mathtt{v}]\!]\kappa &= \kappa\mathtt{v}\\
[\![\mathtt{c}]\!]\kappa &= \kappa\mathtt{c}\\
[\![(\mathtt{cons}\ M_0\ M_1)]\!]\kappa &= [\![M_0]\!]\lambda n_0.[\![M_1]\!]\lambda n_1.(\mathtt{let}\ ((\mathtt{v}_0\ n_0))\\
&\qquad\qquad (\mathtt{let}\ ((\mathtt{v}_1\ n_1))\\
&\qquad\qquad\quad \kappa(\mathtt{cons}\ \mathtt{v}_0\ \mathtt{v}_1)))\\
[\![(\mathtt{car}\ M_0)]\!]\kappa &= [\![M_0]\!]\lambda n_0.\kappa(\mathtt{car}\ n_0)\\
[\![(\mathtt{lambda}\ (\mathtt{v}_1\ \ldots\ \mathtt{v}_m)\ M)]\!]\kappa &= \kappa(\mathtt{lambda}_\eta\ (\mathtt{k}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_m)\ [\![M]\!]\lambda n.(\mathtt{k}\ n))\\
[\![(\mathtt{begin}\ (\mathtt{set!}\ \mathtt{v}\ M_0)\ M_1)]\!]\kappa &= [\![M_0]\!]\lambda n_0.(\mathtt{set!}\ \mathtt{v}\ n_0\ [\![M_1]\!]\kappa)\\
[\![(\mathtt{letrec}\ ((\mathtt{v}\ P))\ M)]\!]\kappa &= [\![P]\!]\lambda n.(\mathtt{letrec}\ ((\mathtt{v}\ n))\ [\![M]\!]\kappa)\\
[\![(\mathtt{if}\ (\mathtt{pair?}\ M_0)\ M_1\ M_2)]\!]\kappa &= [\![M_0]\!]\lambda n_0.(\mathtt{let}\ ((\mathtt{k}\ (\mathtt{lambda}_\eta\ (\mathtt{v})\ \kappa\mathtt{v})))\\
&\qquad\qquad (\mathtt{pair?}\ n_0\\
&\qquad\qquad\quad [\![M_1]\!]\lambda n_1.(\mathtt{k}\ n_1)\\
&\qquad\qquad\quad [\![M_2]\!]\lambda n_2.(\mathtt{k}\ n_2)))\\
[\![(\mathtt{call/cc}\ M)]\!]\kappa &= [\![M]\!]\lambda n.(\mathtt{let}\ ((\mathtt{k}_0\ (\mathtt{lambda}_{\eta_0}\ (\mathtt{v})\ \kappa\mathtt{v})))\\
&\qquad\qquad (n\ \mathtt{k}_0\ (\mathtt{lambda}_{\eta_1}\ (\mathtt{k}_1\ \mathtt{v})\ (\mathtt{k}_0\ \mathtt{v}))))\\
[\![(M_0\ M_1\ \ldots\ M_m)]\!]\kappa &= [\![M_0]\!]\lambda n_0.\\
&\qquad\qquad [\![M_1]\!]\lambda n_1.\ldots.\\
&\qquad\qquad\quad [\![M_n]\!]\lambda n_m.(n_0\ (\mathtt{lambda}_\eta\ (\mathtt{v})\ \kappa\mathtt{v})\ n_1\ \ldots\ n_m)
\end{aligned}$$

where variables and tags introduced into the output are fresh.

Fig. 3: A translation function from *CS* to *NCS*.

functions, *all* operators on the right-hand side are dynamic except for the metalevel operator λ and applications of the function $[\![]\!]$ and variable κ. Uses of these operators are always static. Explicit overlining and underlining has therefore been omitted since the category of each operator is syntactically determined.

The function $[\![]\!]$ is a linear-time algorithm that is similar to an A-normalization algorithm [Flanagan et al. 1993]. It takes as arguments an expression M and a continuation argument κ. The continuation takes a simple expression and returns a normalized expression, and when invoked, the continuation completes the translation. Simple expressions are reduced and passed to the continuation. The translated arguments to `cons` are `let`-bound to fresh variables before the reconstructed expression is passed to the continuation, to preserve call-by-value semantics for cons applications. For assignment and `letrec` expressions, the head expression is simplied, and the continuation is processed with κ unchanged. For applications and continuation capture, the head expressions are simplified and the expression residualized introducing an additional argument for the continuation. For conditionals the continuation is residualized to avoid duplicating it. An example translation from *CS* to *NCS* is given in Figure 4.

Defining a flow analysis on *NCS* follows the usual abstract interpretation-based methodology [Cousot and Cousot 1977]. First a collecting operational semantics is defined that assigns an exact meaning to programs. The flow analysis is then given as an abstract operational semantics defined in terms of the collecting semantics.

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

(letrec ((copy (lambda_η1 (k ls)
                 (pair? ls
                   (copy (lambda_η0 (v0)
                           (let ((v1 v0))
                             (let ((v2 (car ls)))
                               (k (cons v2 v1)))))
                         (cdr ls))
                   (k '())))))
  (let ((v3 '()))
    (let ((v4 2))
      (let ((v5 (cons v4 v3)))
        (let ((v6 1))
          (copy (lambda_η1 (v12) (let ((v v12)) halt))
                (cons v6 v5)))))))
```

Fig. 4: The procedure copy and its equivalent in normalized Scheme. The example uses `cdr`, which is treated analogously to `car`. The example has been simplified to eliminate trivial continuations.

2.1 The collecting machine

The collecting machine is an instrumented variant of a CES (code, environment, store) machine [Felleisen 1987] that executes normalized programs. A CES machine is a state transition machine that manipulates the code, environment, and store on each step. The machine does not need to maintain a continuation because *NCS* programs are in continuation-passing style.

The collecting machine differs from a standard machine by building a cache as it executes a program. The machine is defined in Figure 5. The following notation is used in its description. The term $A \multimap\!\!\!\!\rightarrow B$ denotes the set of partial functions from A to B that have finite domains. A finite function $F : A \multimap\!\!\!\!\rightarrow B$ is extended with the notation $F[a := b]$ where a and b are elements of A and B respectively. The term A^* denotes the set of tuples of length 0 or more whose elements are all members of A. Tuples are written $\langle x_0, \ldots, x_n \rangle$, but the commas separating elements and the angle brackets are sometimes omitted for clarity, particularly when tuples are nested. For example, the tuple $\langle a, \langle x_0, \ldots, x_n \rangle, b \rangle$ is by convention written $\langle a, x_0 \ldots x_n, b \rangle$. The empty set and empty finite function are denoted by $\emptyset$.

The machine's state consists of a term A, an environment E, a store S, and a cache C. Environments and stores are each represented as finite functions. An environment maps variables to locations and a store maps locations to values. A value is either a constant, a closure, or a pair. A cache is a record of the machine's execution state. The cache maps an execution context to the store describing that context. A context is determined by a program point η and an environment.

Semantics: Let $M \in CS$, $eval(M) = C$ if

$$\langle \mathcal{C}[\![M]\!], \emptyset, \emptyset, \emptyset \rangle \rightarrow^{+} \langle \mathtt{halt}, E, S, C \rangle$$

Data specifications:

$$\begin{array}{rcll}
T \in State & = & A \times Env \times Store \times Cache & \text{(machine states)} \\
E \in Env & = & Vars \twoheadrightarrow Locs & \text{(environments)} \\
S \in Store & = & Locs \twoheadrightarrow Value & \text{(stores)} \\
C \in Cache & = & Tags \times Env \rightarrow Store & \text{(caches)} \\
Value & = & c \mid \langle \mathbf{cl}\ Tags, Vars^*, A, Env \rangle \mid \langle \mathbf{pr}\ Locs, Locs \rangle & \text{(values)}
\end{array}$$

Transition rules:

$$\begin{array}{rcl}
\langle (\mathtt{let}\ ((\mathtt{v}\ N))\ A), E, S, C \rangle & \rightarrow & \langle A, E[\mathtt{v} := l], S[l := \gamma(N, E, S)], C \rangle \\
& & \text{where } \langle l \rangle = new(\langle \mathtt{v} \rangle, S) \\
\langle (\mathtt{letrec}\ ((\mathtt{v}\ N))\ A), E, S, C \rangle & \rightarrow & \langle A, E', S[l := \gamma(N, E', S)], C \rangle \\
& & \text{where } \langle l \rangle = new(\langle \mathtt{v} \rangle, S) \\
& & E' = E[\mathtt{v} := l] \\
\langle (\mathtt{set!}\ \mathtt{v}\ N\ A), E, S, C \rangle & \rightarrow & \langle A, E, S[E(\mathtt{v}) := \gamma(N, E, S)], C \rangle \\
\langle (\mathtt{pair?}\ N\ A_1\ A_2), E, S, C \rangle & \rightarrow & \langle (pair?(\gamma(N, E, S)) \rightarrow A_1, A_2), E, S, C \rangle \\
\langle (N_0\ N_1\ \ldots\ N_n), E, S, C \rangle & \rightarrow & \langle A, E'', S', C[\langle \eta, E'' \rangle := S'] \rangle \\
& & \text{where } \langle \mathbf{cl}\ \eta, \mathtt{v}_1 \ldots \mathtt{v}_n, A, E' \rangle = \gamma(N_0, E, S) \\
& & \langle l_1, \ldots, l_n \rangle = new(\langle \mathtt{v}_1, \ldots, \mathtt{v}_n \rangle, S) \\
& & E'' = E'[\mathtt{v}_1 := l_1, \ldots, \mathtt{v}_n := l_n] \\
& & S' = S[l_1 := \gamma(N_1, E, S), \\
& & \qquad \ldots, \\
& & \qquad l_n := \gamma(N_n, E, S)]
\end{array}$$

Converting simple expressions to values:

$$\begin{array}{rcl}
\gamma(\mathtt{c}, E, S) & = & c \\
\gamma(\mathtt{v}, E, S) & = & S(E(\mathtt{v})) \\
\gamma((\mathtt{cons}\ \mathtt{v}_0\ \mathtt{v}_1), E, S) & = & \langle \mathbf{pr}\ E(\mathtt{v}_0), E(\mathtt{v}_1) \rangle \\
\gamma((\mathtt{car}\ N), E, S) & = & S(l_0) \text{ where } \langle \mathbf{pr}\ l_0, l_1 \rangle = \gamma(N, E, S) \\
\gamma((\mathtt{lambda}_\eta\ (\mathtt{v}_1\ \ldots\ \mathtt{v}_n)\ A), E, S) & = & \langle \mathbf{cl}\ \eta, \mathtt{v}_1 \ldots \mathtt{v}_n, A, E \rangle
\end{array}$$

Fig. 5: Collecting machine

The domains *Vars* and *Tags* are program dependent and thus finite, since a given program contains only a finite number of variables and lambda expressions.

The transition rules describe a relation between machine states that describes how the machine executes. When the machine is in a state T, execution proceeds to the next state assuming that a transition rule matches T and that the meaning of any auxiliary functions used to compute the subsequent state are well defined. The machine terminates successfully when it halts in a state with the term `halt`.

The transition rules are straightforward. For `let` and `letrec` expressions the right hand side is evaluated and the result bound to a fresh location allocated using the function *new*. The function takes a sequence of variables and a store S. It returns a sequence of locations that are not in the domain of S. Assignment and conditional expressions follow their usual semantics. Both are straightforward transitions since the right-hand side of an assignment and the test expression of a conditional are simple expressions. An assignment updates the store with the value of the right-hand side, and a conditional uses the test expression to select the arm to which control should be transferred. The mechanics of the test are abstracted by the auxiliary function *pair?*, which in an actual implementation would make its determination based on whether or not its argument evaluates to a pair.

Procedure application must perform three actions: determine to where control must be transferred, update the environment and store with bindings for the formal parameters, and update the cache with the current execution state. The operator is evaluated to obtain the closure to be applied. Its lexical environment is extended with bindings for its formal parameters where again the function *new* is used to allocate fresh locations. The store is updated with bindings for the fresh locations using the argument expressions to obtain the actual parameters. Finally, the cache is updated using the closure's tag η, the extended environment as the execution context, and the updated store as the description of the machine's state at that point. If the operator does not evaluate to a closure or an incorrect number of arguments are received then the meaning of the application is undefined.

Simple expressions are evaluated using the auxiliary function γ, which is inductively defined over the syntax of simple expressions. It takes a simple expression, an environment, and a store as arguments. A constant term is mapped to a constant value, a variable is dereferenced, and a lambda expression is mapped to a closure that closes over the current environment. For a `car` expression, the argument is evaluated to obtain a pair and the store dereferenced to obtain the result value. If the argument does not evaluate to a pair then the meaning is undefined. For a `cons` expression, the locations used for the new pair's constituents are the locations to which the argument variables are bound.

As given, the collecting machine in fact implements a flow analysis. After the machine terminates successfully, a postprocessor can use the cache to determine control- and data-flow graphs for the program. Unfortunately, the collecting machine does not terminate for all programs since nonterminating programs lead to an infinite number of execution states. From the collecting machine, however, it is possible to derive an abstract machine that implements a computable but approximate flow analysis.

2.2 The abstract machine

The abstract machine, defined in Figure 6, is a computable abstraction of the collecting machine. It is obtained by collapsing the cache into a finite function. Since for a given program the sets *Tags* and *Vars* are finite, the cache is infinite only when a variable becomes bound to an infinite number of locations during program execution. Thus if the set of locations to which a variable may become bound is restricted to be finite, the cache is guaranteed to be a finite function. Even for terminating computations it is useful to restrict the set of locations in order to keep the cache manageable in a practical implementation.

The set of locations is restricted using the following strategy. For each variable $\mathtt{v}$ in a program, take $L_{\mathtt{v}}$ to be the set of locations to which $\mathtt{v}$ may become bound as the collecting machine executes. The set $L_{\mathtt{v}}$ is divided into a finite number of partitions, and a location is associated with each partition. As an example, take l to be the location identifying one partition of locations $\{l_0, \ldots\}$. As the abstract machine executes, it will construct stores that map l to a value that approximates all of the values to which $l_0, \ldots$ would be bound in stores constructed by the collecting machine. When the abstract machine terminates, the cache will thus be a finite approximation of the cache that would have been computed by the collecting machine.

A value in the abstract machine must approximate a set of values computed by the collecting machine. It is referred to as an *abstract* value to distinguish it from concrete values manipulated by the collecting machine. An abstract value is a set of concrete values, but the set is always finite, because both the set of environments and the set of locations are finite in the abstract machine.

The consequences of using caches that are finite functions and abstract values that are sets of concrete values is reflected in the specification of the abstract machine given in Figure 6. In the specification, $\mathcal{P}(A)$ denotes the powerset of A.

A state in the abstract machine consists of a term, an environment, a store, a cache, and a *pending set*. The pending set records partial computations that remain to be completed by the machine. As in the collecting machine, an environment maps variables to locations and a cache maps a tag and environment to a store. The store is defined differently, however, in that a store maps a location to a *set* of values. Also, the domain $\widehat{Store}$ is lifted so that caches are total functions. For a given program the domain of stores forms a complete partial order $(\widehat{Store}_\perp, \sqsubseteq)$. The operator $\sqsubseteq$ is defined to be $\perp \sqsubseteq S$ and, for all stores S_0 and S_1,

$$S_0 \sqsubseteq S_1 \;\Leftrightarrow\; dom(S_0) \subseteq dom(S_1) \wedge \forall l \in dom(S_0).S_0(l) \subseteq S_1(l)$$

Several auxiliary functions are used in the machine's definition. The function $\widehat{new}$ plays the same role as *new* but is restricted as described above to a finite codomain. This guarantees that the set of locations to which a variable is bound is finite. The abstract environment is passed as an additional argument so that variations on the analysis may select locations using the current analysis context. The function Θ used in the definition of *apply* is an operator used to control the speed of the analysis. The machine implements a 0CFA analysis if $Locs = Vars$, $\widehat{new}$ simply returns its first argument, and Θ is the identity function.

Given a term A in an initial state, the machine terminates when there is no transition out of the current state. It terminates successfully when it ends in a

Semantics: Let $M \in CS$, $\widehat{eval}(M) = \hat{C}$ if

$$\langle \mathcal{C}[\![M]\!], \emptyset, \emptyset, \hat{C}_0, \emptyset\rangle \to^+ \langle \mathtt{halt}, \hat{E}, \hat{S}, \hat{C}, \emptyset\rangle$$
$$\text{where } \hat{C}_0(\eta, \hat{E}) = \bot \text{ for all } \eta \in \mathit{Tags} \text{ and } \hat{E} \in \widehat{Env}$$

Data specifications:

$$\begin{array}{rcll}
\hat{T} & \in & \widehat{State} & = A \times \widehat{Env} \times \widehat{Store} \times \widehat{Cache} \times \widehat{Pending} \quad \text{(machine states)} \\
\hat{E} & \in & \widehat{Env} & = \mathit{Vars} \to\!\!\!\!\circ\; \mathit{Locs} \quad \text{(environments)} \\
\hat{S} & \in & \widehat{Store} & = \mathit{Locs} \to\!\!\!\!\circ\; \mathcal{P}(\mathit{Value}) \quad \text{(stores)} \\
\hat{C} & \in & \widehat{Cache} & = \mathit{Tags} \times \widehat{Env} \to \widehat{Store}_\bot \quad \text{(caches)} \\
\hat{P} & \subseteq & \widehat{Pending} & = A \times \widehat{Env} \times \widehat{Store} \quad \text{(pending sets)}
\end{array}$$

Transition rules:

$$\langle(\mathtt{let}\ ((\mathtt{v}\ N))\ A), \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle A, \hat{E}[\mathtt{v} := l], \hat{S}[l := \hat{\gamma}(N, \hat{E}, \hat{S})], \hat{C}, \hat{P}\rangle$$
$$\text{where } \langle l\rangle = \widehat{new}(\langle \mathtt{v}\rangle, \hat{E}, \hat{S})$$

$$\langle(\mathtt{letrec}\ ((\mathtt{v}\ N))\ A), \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle A, \hat{E}', \hat{S}[l := \hat{\gamma}(N, \hat{E}', \hat{S})], \hat{C}, \hat{P}\rangle$$
$$\text{where } \langle l\rangle = \widehat{new}(\langle \mathtt{v}\rangle, \hat{E}, \hat{S})$$
$$\hat{E}' = \hat{E}[\mathtt{v} := l]$$

$$\langle(\mathtt{set!}\ \mathtt{v}\ N\ A), \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle A, \hat{E}, \hat{S}[\hat{E}(\mathtt{v}) := \hat{\gamma}(N, \hat{E}, \hat{S})], \hat{C}, \hat{P}\rangle$$

$$\langle \mathtt{halt}, \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle A, \hat{E}', \hat{S}', \hat{C}, \hat{P} - \{\langle A, \hat{E}', \hat{S}'\rangle\}\rangle$$
$$\text{where } \langle A, \hat{E}', \hat{S}'\rangle \in \hat{P}$$

$$\langle(\mathtt{pair?}\ N\ A_1\ A_2), \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle \mathtt{halt}, \hat{E}, \hat{S}, \hat{C}, \hat{P}'\rangle$$
$$\text{where } \hat{P}' = (\mathit{pair?}(\hat{\gamma}(N, \hat{E}, \hat{S})) \to \{\langle A_1, \hat{E}, \hat{S}\rangle\}, \emptyset) \cup (\mathit{nonpair?}(\hat{\gamma}(N, \hat{E}, \hat{S})) \to \{\langle A_2, \hat{E}, \hat{S}\rangle\}, \emptyset)$$

$$\langle(N_0\ N_1\ \ldots\ N_n), \hat{E}, \hat{S}, \hat{C}, \hat{P}\rangle \to \langle \mathtt{halt}, \hat{E}, \hat{S}, \hat{C}', \hat{P}'\rangle$$
$$\text{where } \{x_0, \ldots, x_m\} = \hat{\gamma}(N_0, \hat{E}, \hat{S})$$
$$f = \mathit{apply}(\hat{E}, \hat{S}, N_1 \ldots N_n)$$
$$\langle \hat{C}', \hat{P}'\rangle = (f(x_0) \circ \ldots \circ f(x_m))\langle \hat{C}, \hat{P}\rangle$$

Converting simple expressions to values:

$$\begin{aligned}
\hat{\gamma}(\mathtt{c}, \hat{E}, \hat{S}) &= \{c\} \\
\hat{\gamma}(\mathtt{v}, \hat{E}, \hat{S}) &= \hat{S}(\hat{E}(\mathtt{v})) \\
\hat{\gamma}((\mathtt{cons}\ \mathtt{v}_0\ \mathtt{v}_1), \hat{E}, \hat{S}) &= \{\langle \mathbf{pr}\ \hat{E}(\mathtt{v}_0), \hat{E}(\mathtt{v}_1)\rangle\} \\
\hat{\gamma}((\mathtt{car}\ N), \hat{E}, \hat{S}) &= \hat{S}(l_0) \cup \ldots \cup \hat{S}(l_n) \\
&\quad \text{where } \{l_0, \ldots, l_n\} = \{l_0 \mid \langle \mathbf{pr}\ l_0, l_1\rangle \in \hat{\gamma}(N, \hat{E}, \hat{S})\} \\
\hat{\gamma}((\mathtt{lambda}_\eta\ (\mathtt{v}_1\ \ldots\ \mathtt{v}_n)\ A), \hat{E}, \hat{S}) &= \{\langle \mathbf{cl}\ \eta, \mathtt{v}_1 \ldots \mathtt{v}_n, A, \hat{E}\rangle\}
\end{aligned}$$

Applying a closure in an abstract context:

$$\mathit{apply} : (\widehat{Env} \times \widehat{Store} \times N^*) \to \mathit{Value} \to (\widehat{Cache} \times \widehat{Pending}) \to (\widehat{Cache} \times \widehat{Pending})$$

$$\mathit{apply}(\hat{E}, \hat{S}, N_1 \ldots N_n)(\langle \mathbf{cl}\ \eta, \mathtt{v}_1 \ldots \mathtt{v}_n, A, \hat{E}'\rangle)\langle \hat{C}, \hat{P}\rangle = \langle \hat{C}[\langle \eta, \hat{E}''\rangle := \hat{S}''], \hat{P} \cup \hat{P}'\rangle$$
$$\begin{aligned}
\text{where } \langle l_1, \ldots, l_n\rangle &= \widehat{new}(\langle \mathtt{v}_1, \ldots, \mathtt{v}_n\rangle, \hat{E}, \hat{S}) \\
\hat{E}'' &= \hat{E}'[\mathtt{v}_1 := l_1, \ldots, \mathtt{v}_n := l_n] \\
\hat{S}'' &= \Theta(\hat{C}(\eta, \hat{E}'') \sqcup \hat{S}[l_1 := \hat{\gamma}(N_1, \hat{E}, \hat{S}), \ldots, l_n := \hat{\gamma}(N_n, \hat{E}, \hat{S})]) \\
\hat{P}' &= (\hat{C}(\eta, \hat{E}'') = \hat{S}'') \to \emptyset, \{\langle A, \hat{E}'', \hat{S}'\rangle\}
\end{aligned}$$

Fig. 6: Abstract machine

state with the term `halt` and an empty pending set. As the machine executes, it builds a progressively more general cache until it is a safe approximation of the cache computed by the collecting machine. As with the collecting machine, the cache computed by the abstract machine represents the output.

The evaluation function $\hat{\gamma}$ builds an abstract value from a simple expression. For expressions that evaluate to a single concrete value, that value is injected into a set to form an abstract value. The argument of a `car` expression evaluates to an abstract value that may contains multiple pairs. Its value is therefore the union of the car of each pair in the set, and its value is unspecified if there are nonpairs in the abstract value of the argument.

The execution rules of the abstract machine are similar to those of the collecting machine but must also take into account that a value is now a set. The rules for `let` and `letrec` are essentially identical to their counterparts in the collecting machine. For an assignment, the store is updated with the abstract value of the right-hand side. There is no need to union the new value with the old value at the assigned location as the least upper bound operation at the next program point will merge the old value with the new value in the continuation. For a `halt` expression, the next pending computation in P is retrieved and the machine restarted. For a conditional, the test evaluates to an abstract value containing several concrete values. Some may be pairs and some may not be. The function *pair*? adds the true arm to the pending set if the test value contains a pair. Likewise, *nonpair*? does the same for the else arm if the test value contains something other than a pair.

The application rule must anticipate the operator evaluating to a set of closures. The auxiliary function *apply* is a curried function used to apply each closure to its arguments. The environment, store, and arguments are constant for each closure application. They are combined by *apply* to obtain a function f. The function f when applied to a closure returns a new function that maps a cache and pending set to a new cache and pending set. The new function is applied to each closure and the results are composed to build a transformer that takes a cache and pending set and returns a new cache and pending set. If a nonclosure is applied the machine's behavior is unspecified.

When *apply* has received all of its arguments it updates the cache with a new store S' and perhaps adds an element to the pending set. The store S' is obtained in three steps. First, the incoming store is extended with bindings for the formal parameters and projected using the operator Θ. The least upper bound of the result and the program point's old store is then taken. The pending set is updated if the new store S' is different from the old store recorded by the cache, and the added tuple $\langle A, E'', S' \rangle$ indicates that the abstract machine must eventually evaluate the term A in environment E'' and store S'.

2.3 Regulating accuracy and speed

The functions $\widehat{new}$ and Θ regulate the speed and accuracy of the analysis. The function $\widehat{new}$ primarily regulates accuracy by splitting environments. Environments are split by defining $\widehat{new}$ such that different locations are associated with variables based on the analysis context. The projection operator Θ is used mainly to increase the speed. By defining Θ so that for all $\hat{S} \in \widehat{Store}$, $\hat{S} \sqsubseteq \Theta(\hat{S})$ the convergence to a stable cache is accelerated.

While $\widehat{new}$ tends to regulate accuracy and Θ tends to regulate speed, the two functions each have an impact on both accuracy and speed. Using $\widehat{new}$ to create a 1CFA analysis, for instance, yields a more accurate analysis that is $O(2^n)$ in the size of the input program. Using Θ to accelerate the analysis usually results in a solution that is not the least solution, affecting the accuracy of the analysis.

As an example of the machine's output, consider the normalized Scheme program in Figure 4 and assume a 0CFA analysis in which Θ is the identity function and $\widehat{new}$ projects its first argument. Once the machine terminates the cache will associate a store with each program point. A portion of the store on entry to `copy` would be

$$\begin{aligned}
\texttt{ls} &= \{\langle \mathbf{pr}\ v_6, v_5\rangle, \langle \mathbf{pr}\ v_4, v_3\rangle, ()\} \\
v_6 &= \{1\} \\
v_5 &= \{\langle \mathbf{pr}\ v_4, v_3\rangle\} \\
v_4 &= \{2\} \\
v_3 &= \{()\}
\end{aligned}$$

In particular, the arguments bound to `ls` have been collapsed into one set.

A 1CFA analysis can be implemented by modifying $\widehat{new}$ so that it allocates locations based on the shape of the environment at each call site. Suppose the call site in the body of the let binding for `copy` is labeled a and the call site in the definition of `copy` is labeled b. The store on the initial call to `copy` would bind `ls` as follows.

$$\begin{aligned}
\texttt{ls}^a &= \{\langle \mathbf{pr}\ v_6, v_5\rangle\} \\
v_6 &= \{1\} \\
v_5 &= \{\langle \mathbf{pr}\ v_4, v_3\rangle\} \\
v_4 &= \{2\} \\
v_3 &= \{()\}
\end{aligned}$$

The store on all recursive calls to `copy` would bind `ls` as follows.

$$\begin{aligned}
\texttt{ls}^b &= \{\langle \mathbf{pr}\ v_4, v_3\rangle, ()\} \\
v_4 &= \{2\} \\
v_3 &= \{()\}
\end{aligned}$$

For this program the abstract machine creates two cache entries for the program point corresponding to entry into `copy`. A compiler might use the more refined information to inline the procedure `copy` at the initial call site and then perform constant folding and copy propagation based on the exact information available at the initial call site.

The specification in Figure 6 supplies little information to the two functions. A practical implementation will often provide more, and we give two examples to illustrate. In each case, the abstract machine must be modified to transmit the additional information. The first example shows how $\widehat{new}$ may be extended to improve the accuracy of the analysis. The second example shows how Θ may be implemented improve the speed of the analysis. The two examples depend on the following subsets of the domain *Value*.

$$\begin{aligned}
\mathit{Pairs} &= \langle \mathbf{cl}\ \mathit{Tags}, \mathit{Vars}^*, A, \mathit{Env}\rangle \\
\mathit{Closures} &= \langle \mathbf{pr}\ \mathit{Locs}, \mathit{Locs}\rangle
\end{aligned}$$

2.3.1 *Splitting on argument types.* Our splitting strategy is similar to the splitting strategy used by Schism [Consel 1993]. In Schism, the environment is split based on the binding times of a procedure's arguments. Our splitting strategy simply splits based on the types. The domain of locations is $\mathit{Vars} \times \mathit{Types}$ where $\mathit{Types} = \{\mathit{const}, \mathit{pair}, \mathit{closure}\}$. The definition of $\widehat{new}$ must also be modified to accept the values to be bound to the locations. A definition of $\widehat{new}$ that implements the splitting strategy is

$$\widehat{new}(\langle v_1, \ldots, v_n\rangle, \langle x_1, \ldots, x_n\rangle, \hat{E}, \hat{S}) = \langle v_1, \mathit{type}(x_1)\rangle \ldots \langle v_n, \mathit{type}(x_n)\rangle$$
$$\text{where } \mathit{type}(x) = \begin{cases} \mathit{const} & \text{if } x \cap \mathit{Constants} \neq \emptyset \\ \mathit{pair} & \text{if } x \cap \mathit{Pairs} \neq \emptyset \\ \mathit{closure} & \text{if } x \cap \mathit{Closures} \neq \emptyset \end{cases}$$

It is not possible for values to be mixed, *e.g.*, constants in a value associated with a location typed as a pair, since the primitive operations generate singleton sets, and the *apply* function in conjunction with the redefined $\widehat{new}$ function maintains the partitioning as analysis proceeds.

2.3.2 *Collapsing pairs.* As defined, the analysis maintains an abstract pair for each occurence of `cons` evaluated in each analysis context. The operations on pairs do not take advantage of this separation, however. The separation would be used only in a polyvariant analysis in which $\widehat{new}$ made splitting decisions based on the separation. If this is not the case, the efficiency of the analysis could be improved by folding pairs.

The following defines a projection operator Θ that implements this strategy. It takes an additional argument: the locations just added to the store by the *apply* function. The abstract value associated with each location is collapsed by folding the abstract values of the components of all the pairs in the abstract value. A representative pair is selected to represent the folded pairs and the others are removed from the abstract value.

$$\begin{aligned}
\Theta(l_1 \ldots l_n, \hat{S}) &= \mathit{refine}(l_1, \mathit{refine}(\ldots, \mathit{refine}(l_n, \hat{S}))) \\
\mathit{refine}(l, \hat{S}) &= \begin{cases} \hat{S} & \text{if } \mathit{Pairs} \cap \hat{S}(l) = \emptyset \\ \mathit{collapse}(l, \hat{S}) & \text{otherwise} \end{cases} \\
\mathit{collapse}(l, \hat{S}) &= \hat{S}[l := (\hat{S}(l) - \mathit{Pairs}) \cup \{\langle \mathbf{pr}\ l_1, l_2\rangle\}, \\
&\qquad l_1 := \bigcup\{\hat{S}(l_1) \mid \langle \mathbf{pr}\ l_1, l_2\rangle \in \hat{S}(l)\}, \\
&\qquad l_2 := \bigcup\{\hat{S}(l_2) \mid \langle \mathbf{pr}\ l_1, l_2\rangle \in \hat{S}(l)\}] \\
&\qquad \text{for some } \langle \mathbf{pr}\ l_1, l_2\rangle \in \hat{S}(l)
\end{aligned}$$

2.4 Correctness

The abstract machine is correct if it terminates for all programs and, upon termination, its cache is a safe approximation of the collecting machine's cache.

The machine runs until the pending set is empty, so the machine terminates if a

finite number of program points are added to the pending set. Since $\widehat{new}$ produces a finite number of locations, the stores manipulated by the machine are each finite and together form a finite complete partial order under $\sqsubseteq$. Since the store at a program point is updated monotonically, only a finite number of programs points can be added to the pending set. Hence, the machine always terminates.

The safety of the analysis may be established by reasoning about corresponding execution traces of the abstract and collecting machines. A proof would proceed by arguing that the collecting machine's cache at each execution step is safely approximated by the cache in the abstract machine when the abstract machine terminates. This may be shown by induction on the number of steps in the collecting machine's execution trace. A complete proof may be found in a related technical report [Ashley and Dybvig 1998].

3. IMPLEMENTATION

We have incorporated the analysis into the Chez Scheme [Dybvig 1994] compiler and used it to justify certain program optimizations. The compiler processes ANSI Scheme and directly supports multiple return values [Ashley and Dybvig 1994] and a variable arity procedure interface [Dybvig and Hieb 1990].

As defined in Section 2, the analysis can process only closed programs, *i.e.*, programs with no free variables other than recognized primitives. This is unacceptable for a realistic implementation, since this restriction implies that program parts cannot be analyzed in isolation. The actual implementation of the analysis therefore handles free variable references by introducing a unique abstract value $\{unknown\}$ to denote the value of a free variable reference.

When *unknown* is used as a closure in an application, the arguments of the application *escape*. The consequences of the escape depends on the escaped value. If the value is a procedure, the analysis must assume that the procedure is applied to arguments that have the abstract value $\{unknown\}$. For data structures, the values bound to all accessible locations also escape, and furthermore, the analysis must conservatively assume that in the continuation of the call all of the structure's mutable locations have the abstract value $\{unknown\}$. Finally, unknown values must be assumed to be both pairs and nonpairs in the handling of conditionals.

The abstract machine's behavior when operators are applied to illegal values is unspecified, but the implementation must make a commitment. There are two choices. If at run-time the type error would result in a reentrant continuation the implementation must assume the expression could evaluate to any value, *i.e.*, $\{unknown\}$. If it is not reentrant, the implementation may assume the abstract value of the expression is the empty set. In our implementation, error continuations are not reentrant and the latter strategy is used.

The formal specification of the analysis does not explicitly use type tests to constrain abstract values but this is done in the implementation. This is accomplished by rebinding the tested variable to a new location in each arm of the conditional and filling the locations with the constrained values of the variable [Heintze 1994]. When control leaves the lexical context of the conditional, the variable reverts to its former binding.

While the operational specification of the analysis may be seen as an interpreter, the implementation stages the interpreter so that code is compiled to closures and

the closures then executed. The implementation accepts the Chez Scheme compiler's intermediate representation as input and dynamically generates closures that are executed to obtain the solution.

Closures are generated dynamically since it would be impractical to generate them in a single preprocessing step. Closure generation is dependent on the abstract context in which an expression is analyzed. In a polyvariant analysis the number of potential contexts can be quite large, even exponential in the size of the program, but during analysis only a small fraction of the contexts may actually be encountered. Hence, it is more practical to generate closures lazily.

The analysis as specified in Section 2 is sensitive to where assignments occur in the control flow of a program. In other words, the abstract value of a variable at a control point prior to an assignment may be completely different from the value at the control point following the assignment. This is more precise than an analysis that is control-flow insensitive to when assignments occur. In a control-flow insensitive analysis, the most approximate value of a variable is used at all points in the control-flow graph. Set-based analysis [Heintze 1994] is an example of a flow-insensitive analysis. A flow-sensitive analysis may be beneficial in languages that use assignment more frequently, *e.g.*, object-oriented languages. Scheme, however, does not encourage assignment, and assignments occur relatively infrequently.

Both flow-sensitive and flow-insensitive versions of the analysis framework are implemented. The flow-insensitive version is straightforward to implement efficiently and does not differ much from the constraint solver for a set-based analysis [Flanagan and Felleisen 1995; Heintze 1994]. The flow-sensitive version can also be implemented efficiently by using a global store for immutable locations and incrementally extending stores where updates occur. A combination of caching and lazily computing least upper bounds then yields a fast implementation.

The two implementations have comparable performance, and for the benchmarks reported the information collected was essentially identical for purposes of enabling optimizations. The flow-insensitive implementation, however, is significantly less complex than the flow-sensitive version. We therefore use the flow-insensitive version in the compiler.

4. EVALUATION

Instantiations of the analysis framework can be used to enable compiler optimizations. Procedure inlining, for example, has been justified by 0CFA and polynomial-1CFA instantiations [Ashley 1997]. We also use two instantiations to enable a set of call optimizations described below. One instantiation is 0CFA, but a 0CFA analysis is $O(n^3)$. This upper bound prevents its unconditional use in a production compiler. In order to balance compile-time speed with the quality of generated code, it is important to know if a faster but less accurate analysis is still useful for enabling optimizations.

To determine this, the framework is also instantiated to obtain an analysis that is faster but less accurate than 0CFA. This is accomplished by retaining the same $\widehat{new}$ function used in the 0CFA instantiation but using a nontrivial projection operator Θ. The projection operator tracks the number of times the cache has been updated at each program point. If the number exceeds a threshold n at some point, the cache at that point is updated with a store projected from the new store. The

projected store is similar to the new store except that the values responsible for the update are considered escaping and *unknown* is substituted in their place. If the threshold is n, this projection operator limits the analysis to a worst case of $n+1$ passes over the program. Setting $n = \infty$ results in a 0CFA analysis for any program input to the analysis.

As an example consider the following program with $n = 1$.

```
(let ((f (lambda (x) x)))
  (f 2)
  (f #t))
```

During analysis the cache describing the store on entry to the procedure `f` will be updated twice. On the first update the location bound to `x` will have the abstract value $\{2\}$. On the second update it will be $\{2, \mathit{unknown}\}$ since the threshold will be exceeded and the value causing the update, *i.e.*, the boolean `#t`, will be replaced with *unknown*.

Limiting the number of passes over a program yields a class of analyses we call *sub-0CFA*. Therefore setting n to some natural number yields an approximation of 0CFA. For any given program, however, there exists an analysis in the sub-0CFA class that is identical to 0CFA since the analysis always terminates in a finite number of iterations. While any analysis with a finite threshold is a sub-0CFA analysis, letting $n = 1$ yields an inexpensive analysis we call *sub-0CFA* in our evaluation.

In exchange for a linear-time analysis, the projection operator may force some program points to be generalized. The precision lost in the generalization is contained by the analysis in two ways. First, when a program point is generalized, the information that is already known at that point is not discarded. Only new information is replaced with the abstract value $\{\mathit{unknown}\}$. Second, only unstable program points are generalized. For example, suppose $n = 5$ and the following program is analyzed.

```
((lambda (f g) (f 1) (g 2))
 (lambda (x) ...)
 (lambda (y) ...))
```

Also assume that the code for `f` stabilizes in three iterations and the code for `g` stabilizes in ten iterations. The flow analysis will be forced to generalize the information about `g`, but it will not have to generalize the information about `f` since the control- and data-flow for `f` does not depend on `g`.

Procedure representation and procedure call are more complex in Scheme than in first-order, statically-typed languages. Because procedures are first-class, the evaluation of a lambda expression yields a *closure* consisting of the procedure's code and the environment in which the expression was evaluated. In addition, the operator at a call site is generally unknown so performing a procedure call is more elaborate than usual. It involves steps that include

(1) evaluating the operator into a reserved closure pointer (`cp`) register,
(2) performing a type-check to ensure the operator is a procedure,
(3) retrieving the code pointer from the operator's closure,
(4) jumping to the code referenced by the code pointer, and

(5) ensuring that the correct number of arguments have been passed.

The compiler uses the results of the analysis to optimize procedure calls and avoid run-time closure construction. Procedure calls can be optimized in several ways. The type check can be eliminated if the operator will always evaluate to a procedure. The code pointer in the closure does not need to be referenced if the operator always evaluates to procedures built from the same lambda expression. Instead, the compiler can emit a direct jump to the applied procedure's code, bypassing the argument-count check on the callee's side. A direct call to a procedure that does not access its free variables can be further optimized by not loading the procedure into the `cp` register. This last optimization also implies that the expression evaluated to yield the procedure can be considered useless code and eliminated if it cannot cause a side effect.

Closure construction can be avoided if the closure has no free variables that will be referenced. Since the compiler is incremental and code is dynamically linked, a closure with no referenced free variables can be constructed once at compile time and linked into the generated code stream. The closure can be eliminated entirely if it satisfies the additional property that its code pointer is never needed. As implied by the above optimization, the code pointer is unneeded if all calls to the procedure are direct calls.

In the example below, both calls to `f` are recognized as direct calls.

```
(letrec ((f (lambda (x) (f x))))
  (car (f 0)))
```

No closure need be constructed, the argument count and type checks can be avoided, the `cp` register does not need to be loaded, and all control transfers are direct jumps to the destination.

The following example computes factorial of 5 in continuation-passing style.

```
(letrec ((f (lambda (x k)
              (if (= x 0)
                  (k 1)
                  (f (- x 1) (lambda (v) (k (* x v))))))))
  (f 5 (lambda (x) x)))
```

Again, the procedure bound to `f` has no free variables, so it can be eliminated and calls to it are direct calls. Applications of the continuation can also be optimized by eliminating the type checks and argument count checks.

The final example below illustrates two more optimizations.

```
(let ((f (lambda (x) ((car x) 5))))
  (f (cons (lambda (a) a) '())))
```

First, the lambda expression (`lambda (a) a`) has no free variables and its value can therefore be constructed at compile time. The value of (`car x`) is a procedure constructed from a known lambda expression, so the call is a direct call, the `cp` register does not need to be loaded, the type and argument count checks can be avoided, and the jump is a direct jump. In addition, since the expression (`car x`) cannot cause a side effect, the evaluation of this expression can be omitted at run time.

Benchmark	lines	Description
texer	3,000	A Scheme pretty-printer with TEX output
similix	7,000	Self-application of the Similix [Bondorf 1993] partial evaluator
ddd	15,000	A hardware derivation system [Bose 1991] deriving a Scheme machine [Burger 1994]
conform	450	A program that manipulates lattices and partial orders
dynamic	2,200	A dynamic type inferencer applied to itself
earley	650	Earley's algorithm for generating parsers for context-free grammars
em-fun	490	EM clustering algorithm in functional style
em-imp	460	EM clustering algorithm in imperative style
graphs	500	A program that counts the number of directed graphs with particular properties
interpret	1000	A Scheme interpreter evaluating the takl [Gabriel 1985] benchmark
lattice	200	A program that enumerates the lattice of maps between two lattices
matrix	550	A program that tests whether a matrix is maximal among all matrices obtained by reordering rows and columns
maze	800	A hexagonal maze generator
nbody	850	A program that computes gravitational forces using the Greengard multipole algorithm
splay	950	A program that builds splay trees

Table I. Benchmarks

The *Chez Scheme* compiler was modified to use the flow analysis to enable these optimizations. How well the analysis enables the optimizations was measured by running a series of benchmarks with the sub-0CFA and 0CFA analyses. The benchmarks are described in Table I. All were run without modification. The first three benchmarks consist of numerous top-level definitions that are separately analyzed and compiled. The remaining benchmarks are self-contained and are therefore block-analyzed and compiled.

Table II shows how long each benchmark takes to run with optimizations disabled, how much time it takes the sub-0CFA and 0CFA analyses to process the benchmark, and the speedups obtained with the optimizations enabled. All times are in seconds and are collected on an Intel 80686 with a 256K level 2 cache running Linux kernel 2.0.18. The data indicates that there is usually not much difference in analysis time between sub-0CFA and 0CFA. On the other hand, for some benchmarks 0CFA takes significantly longer, and for other benchmarks 0CFA is actually faster. 0CFA is sometimes faster because of a small constant overhead associated with the sub-0CFA projection operator and the handling of an increased number of unknown values. The run-time speedups obtained by the optimizations are in the range of immeasurable to 16%. Furthermore, there is no significant difference between the sub-0CFA and 0CFA analyses in terms of how much optimization they can enable. Speedups for the first three benchmarks were relatively poor, but the fact that they were subject to separate compilation explains the results.

The effects of the two analyses are also measured quantitatively. We instrumented the compiler to collect compile-time information about calls generated and run-time information about calls executed. Likewise, compile-time and run-time information about closure construction is collected. This data is given in Tables III and IV. For each benchmark, data is given first for the sub-0CFA analysis and then the 0CFA

benchmark	unoptimized run time	analysis time		run-time speedup	
		$n = 1$	$n = \infty$	$n = 0$	$n = \infty$
texer	1.18	0.37	0.39	6%	5%
similix	10.73	1.36	1.25	1%	1%
ddd	15.76	0.38	0.46	0%	0%
conform	0.22	0.05	0.05	9%	9%
dynamic	0.25	0.36	0.34	0%	0%
earley	0.08	0.06	0.07	12%	12%
em-fun	46.72	0.08	0.08	5%	5%
em-imp	27.09	0.08	0.08	5%	5%
graphs	65.86	0.04	0.04	6%	7%
interpret	1.10	0.55	3.12	0%	0%
lattice	40.36	0.03	0.04	9%	9%
matrix	38.81	0.06	0.08	7%	7%
maze	8.12	0.06	0.06	9%	9%
nbody	34.30	0.21	0.20	16%	16%
splay	0.27	0.06	0.06	14%	14%

Table II. The table measures for each benchmark its running time with optimizations disabled, the compile-time costs of the analyses, and the run-time performance increase after each analysis is used to drive optimizations. Times are given in seconds.

analysis. The static data gives information about operations generated by the compiler. The dynamic data gives information about operations executed at run time. The analyses affect neither the number of call operations for which code is generated nor the number of operations actually performed. Rather, they affect only how the operations were optimized. The distribution of the optimizations are given in the bar graphs.

The data on procedure calls is divided into four categories.

noload	calls where the jump was a direct branch to the destination and the `cp` register did not need to be loaded
direct	calls where the jump was a direct branch and the `cp` register was loaded with the called procedure
nocheck	calls where the jump had to go through the code pointer but the type check could be omitted
unoptimized	calls that could not be optimized

The distribution graphs showing this data are interesting in two ways. First, the cheaper sub-0CFA analysis identifies the same noload and direct call sites as the 0CFA analysis, and these calls usually account for most of the dynamic calls in a program. Second, the 0CFA analysis improves on the sub-0CFA analysis precisely by identifying those sites where the operator is definitely one of a set of procedures. In the benchmark suite, the programs lattice, graphs, nbody, and splay all benefit from the extra precision of 0CFA, but the observable difference in run-times is insignificant. In general, a program that makes extensive use of higher-order procedures, *e.g.*, a program in continuation-passing style, will benefit from the extra precision of the 0CFA analysis.

The data on closures falls into three categories:

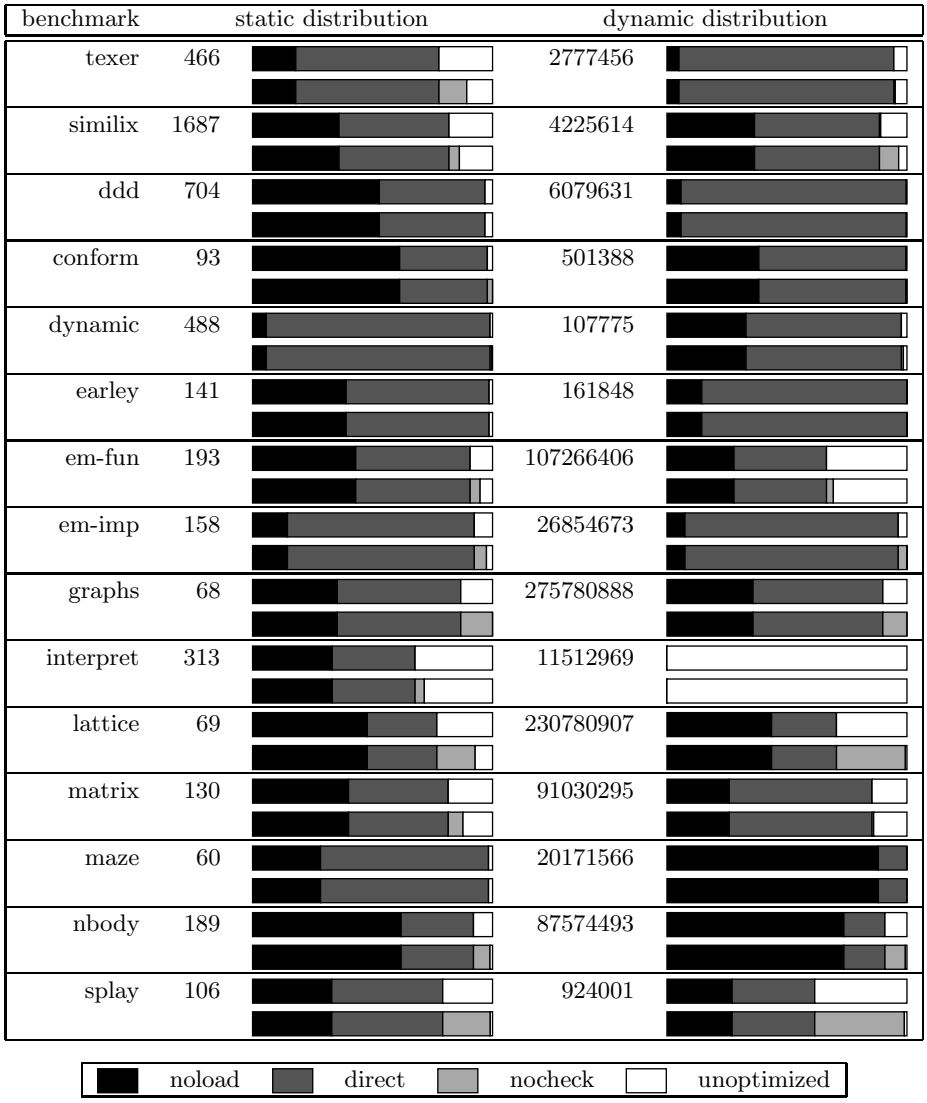

Table III. Static and dynamic distribution of optimized procedure calls. The data for sub-0CFA is given above the data for 0CFA.

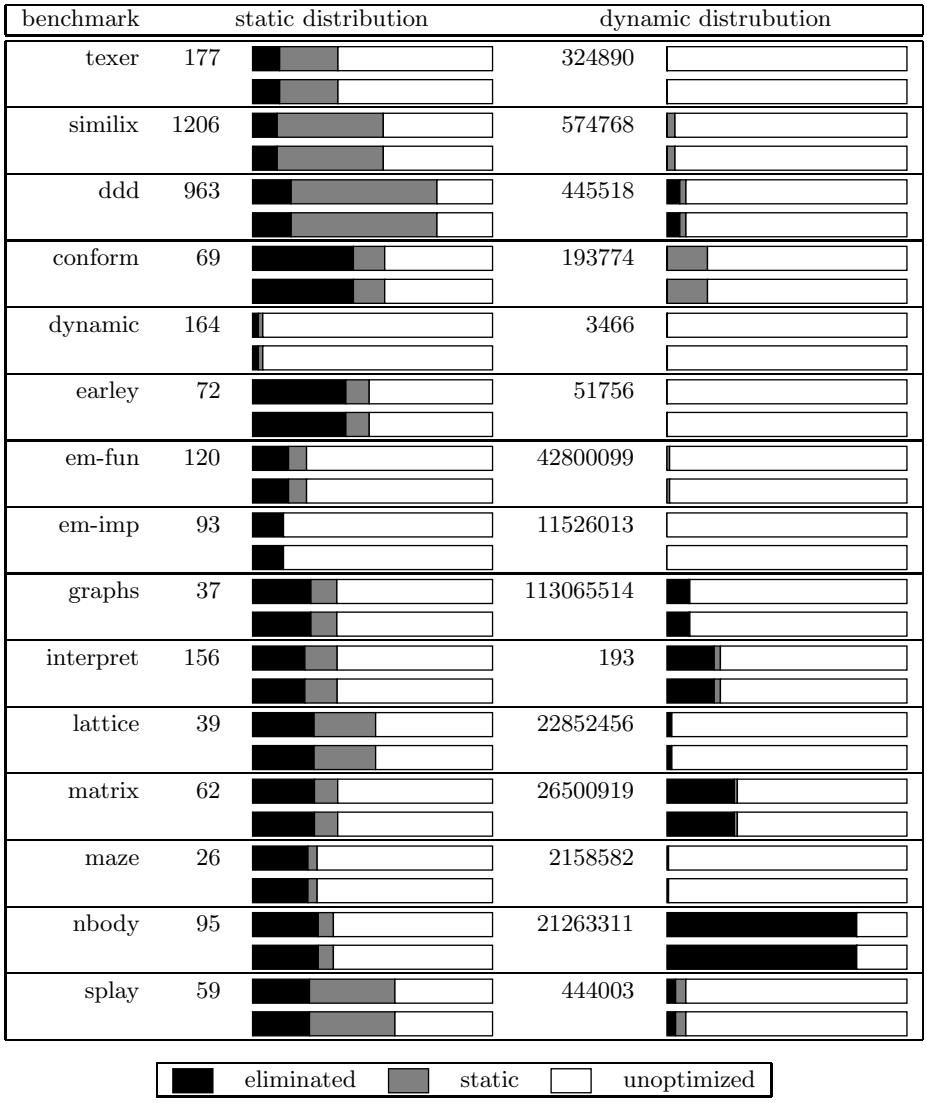

Table IV. Static and dynamic distribution of optimized closure construction. The data for sub-0CFA is given above the data for 0CFA.

eliminate	closures that could be entirely eliminated from the program
static	closures that could be constructed once at compile time and linked into the compiled code
unoptimized	closures that could not be optimized and had to be constructed at run time.

In the dynamic distribution graphs, the areas marked as eliminated or static represent the proportion of the closures that would have been constructed at run time had they not been eliminated or constructed at compile time.

With closure optimization the differences between the compile-time and run-time distributions is significant. A significant fraction of the lambda expressions in the benchmarks could be optimized, but those lambda expressions were not repeatedly evaluated during execution. The lambda expressions that were optimized were typically procedures defined in outer lexical contours and then applied repeatedly. The benchmark nbody is the exception, where closure elimination reduced closure allocation significantly at run time. The primary benefit of closure optimization is not the savings in allocation costs, however. Rather, closure optimization enables the compiler to turn direct calls into noload calls.

5. RELATED WORK

Basing the analysis on a collecting semantics for the language is similar to the approach taken by Young [Young 1987]. In his approach, a denotational collecting semantics is defined that collects information about the value of an expression. The static analysis is then expressed as an abstraction of the collecting semantics. Our semantics differs in that both the collecting and abstract semantics are defined operationally.

Shivers [Shivers 1991] and Harrison [Harrison III 1989] describe flow analyses for Scheme that are based on abstract interpretation. They differ primarily in the details of the source language and the range of accuracy they can express. Our analysis draws from the advantages of each approach. Like Shivers' analysis, we use CPS to make control transfers explicit. Like Harrison's analysis, however, code is kept in direct-style within a basic block and `let` bindings are used to order operations. Our analysis extends their analyses by expressing a wider range of polyvariance and using projection to accelerate convergence to a fixpoint. Also, their analyses were prototypes, while our analysis is completely implemented and running in a production compiler.

Jagannathan and Weeks [Jagannathan and Weeks 1995] describe a polyvariant analysis for higher-order languages that also accommodates side effects and first-class continuations. Their analysis is parameterized over a polyvariance operator, but it is not parameterized over a projection operator. Also, their characterization of program state is different from ours. In our analysis, the program state is an environment and store that is extended at each program point. In their analysis, the program state is an environment mapping variables to locations in a global store. With respect to polyvariance, our $\widehat{new}$ operator splits program points in the same way their analysis does by splitting the environment.

Serrano and Feeley [Serrano and Feeley 1996] describe a static analysis for storage use applications that also handles escaping values. Their treatment is the same

as ours. Escaping closures are applied to unknown arguments, and the mutable locations of data structures are given an unknown value. Our treatment of escaping values is handled implicitly in the implementation while their formal specification includes the treatment of escaping values.

Boucher and Feeley [Boucher and Feeley 1996] introduce the notion of *abstract compilation* as a technique for accelerating static analyses. Like our staging of the abstract machine, their approach is also inspired by partial evaluation and involves eliminating the interpretive component of the analysis. Their analysis is 0CFA while our framework can express a range of accuracy. Since their analysis is 0CFA, compilation can proceed in a preprocessing step before the compiled analysis is executed. Since our analysis may be polyvariant, however, we interleave compilation and execution so that compilation occurs on demand.

Some monovariant flow analyses are specified declaratively [Heintze 1994; Palsberg and Schwartzbach 1995]. The analyses are implemented by deriving a set of constraints from an input program and then solving those constraints. The advantage of the approach is that it maintains a clean separation between constraint generation and constraint satisfaction. This separation is impractical in a polyvariant analysis, however, where the number of potential abstract contexts can be quite large relative to the number of abstract contexts actually encountered during analysis. Our analysis is a hybrid solution where the specification is operational, but the implementation dynamically compiles code for abstract contexts as they are encountered and executes the generated closures on demand. Furthermore, the behavior of the generated closures is similar to the behavior of a constraint solver.

Other researchers have observed that projection operators can be used in practice to reduce the number of iterations needed to stabilize. Yi and Harrison [Yi and Harrison III 1993] describe a framework for the automatic generation of abstract interpreters. This framework incorporates a notion of projection that is similar to ours. The difference is that they apply projections to values, and we apply them to the entire computation state. Furthermore, they always project a value to the top of the value lattice, while we permit the operator to project a value to any other value above it in the lattice. Cousot and Cousot [Cousot and Cousot 1977] and Bourdoncle [Bourdoncle 1992; 1993] use widening operators in a theoretical context as a means of building computable analyses over lattices with infinite height and accelerating analyses over lattices with finite height. A widening operator ∇ is a substitute for the least upper bound operator. The constraint is that given two points x and y, $x \sqcup y \sqsubseteq x \nabla y$. Both of these approaches demonstrate the potential to accelerate the analysis without losing too much information. Our work realizes this potential by exhibiting a practical projection operator that still enables useful optimizations.

6. CONCLUSIONS

Compilers must balance compilation speed and the speed of generated code. Software developers want a fast compiler for the development cycle but want the compiler to generate the best code possible when development is finished. There are two common solutions to the problem. One is to use two compilers: a fast compiler during development and a highly-optimizing "batch mode" compiler when development is complete. The other is to use a single compiler but selectively turn on and

off optimizations. The disadvantage of the first solution is that two compilers must be written and maintained. The disadvantage of the second is that it is harder to verify the compiler's correctness since it is difficult to test all optimization switch permutations.

The flexibility of our analysis suggests the alternative strategy of *always* doing the flow analysis necessary for optimizations. During development, however, the flow analysis is run with a coarse projection operator. As a result, the analyzer is fast, but the compiler sometimes does not have enough information to perform optimizations. When the development cycle has ended, the coarse projection operator is replaced with the identity operator, allowing the analysis to collect more precise information and perform stronger optimizations.

This strategy is attractive despite the fact that 0CFA and sub-0CFA have similar costs on many of the reported benchmarks. In an interactive setting it is essential to provide guarantees about compile times, and some classes of programs, *e.g.*, programs in continuation-passing style, are expensive to analyze with 0CFA. A more subtle problem with 0CFA is that small changes to the program may significantly increase or decrease analysis times. This fluctuation can be frustrating to the programmer. Sub-0CFA guarantees a complexity bound for all programs submitted to the analyzer and avoids these problems.

The efficient implementation of the flow analysis requires that the interpreter be staged into compilation and execution steps. The compilation and execution stages are similar to the constraint-generation and constraint-solution phases of a constraint-based analysis. Despite the similarity in implementation, it is unclear how to specify a polyvariant flow analysis declaratively as a constraint-based analysis. Constraint-based analyses reported in the literature are monovariant and assume that constraint generation occurs as a preprocessing step before constraint satisfaction. Our implementation interleaves constraint generation and satisfaction to avoid generating constraints that might erode the accuracy of the best solution. Reflecting this operational aspect of the analysis in a declarative specification of the analysis is an open problem.

Neither the specification nor implementation of the analysis framework is optimized specifically for polyvariant instantiations of the analysis. Ashley and Consel [Ashley and Consel 1994] describe optimizations for polyvariant analyses that could be applied to our implementation. The optimizations involve eliminating program points that become useless as the analysis works towards a solution.

Adding a module system to the language would improve the accuracy of the analysis. Currently, the analyzer assumes no information about external references. While correct, this is not very satisfactory. With a module system, the analyzer could save the abstract values of exported bindings for later use during the analysis of an importing module. The analysis of the importing module could then infer more accurate information.

7. ACKNOWLEDGEMENTS

Marc Feeley, Suresh Jagannathan, and Andrew Wright provided some of the benchmarks we used in our experiments. Mitch Wand's and the anonymous referees' comments helped improve the presentation significantly.

REFERENCES

Ashley, J. M. 1997. The effectiveness of flow analysis for inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*. ACM, 99–111.

Ashley, J. M. and Consel, C. 1994. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems 16,* 5, 1431–1448.

Ashley, J. M. and Dybvig, R. K. 1994. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. ACM, 140–149.

Ashley, J. M. and Dybvig, R. K. 1998. A practical and flexible flow analysis for higher-order languages (extended version). Tech. Rep. DL-1998-02, University of Kansas Design Laboratory. May.

Ayers, A. E. 1993. Abstract analysis and optimization of Scheme. Ph.D. thesis, MIT.

Bondorf, A. 1993. *Similix Manual, System Version 5.0*. DIKU, University of Copenhagen, Denmark.

Bose, B. 1991. DDD—A transformation system for Digital Design Derivation. Tech. Rep. 331, Indiana University, Computer Science Department. May.

Boucher, D. and Feeley, M. 1996. Abstract compilation: A new implementation paradigm for static analysis. In *Proceedings of the 1996 International Conference on Compiler Construction*. ACM.

Bourdoncle, F. 1992. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming 2,* 4 (Oct.), 407–436.

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

Burger, R. G. 1994. The Scheme machine. Tech. Rep. 413, Indiana University, Computer Science Department. Aug.

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

Cousot, P. and Cousot, R. 1977. 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*. ACM, 238–252.

Dybvig, R. K. 1994. *Chez Scheme System Manual, Rev. 2.4*. Cadence Research Systems, Bloomington, Indiana.

Dybvig, R. K. and Hieb, R. 1990. A new approach to procedures with variable arity. *Lisp and Symbolic Computation 3,* 3 (Sept.), 229–244.

Felleisen, M. 1987. The calculi of lambda-v-cs-conversion: a syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana University, Bloomington, Indiana.

Flanagan, C. and Felleisen, M. 1995. Set-based analysis for full Scheme and its use in soft-typing. Tech. Rep. 253, Rice University. Oct.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. ACM, 237–247.

Gabriel, R. P. 1985. *Performance and Evaluation of LISP Systems*. MIT Press series in computer systems. MIT Press.

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

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

Jagannathan, S. and Weeks, S. 1995. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, 393–407.

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

Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press.

Palsberg, J. and Schwartzbach, M. I. 1995. Safety analysis versus type inference. *Information and Computation 118,* 1, 128–141.

Serrano, M. and Feeley, M. 1996. Storage use analysis and its applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. ACM, 50–61.

Shao, Z. and Appel, A. W. 1994. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. ACM, 130–161.

Shivers, O. 1988. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. ACM, 164–174.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University. CMU-CS-91-145.

Steckler, P. A. and Wand, M. 1997. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems 19,* 1, 48–86.

Yi, K. and Harrison III, W. L. 1993. Automatic generation and management of interprocedural program analyses. In *Proceedings of the 20th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, 246–259.

Young, J. H. 1987. The theory and practice: Semantic program analysis for higher-order functional programming languages. Ph.D. thesis, Yale University.