A Practical and Flexible Flow Analysis for Higher-Order Languages

J. Michael Ashley*
Computer Science Department
Indiana University
Bloomington, Indiana

Abstract

A flow analysis framework for higher-order, mostly-functional languages is given. The framework unifies and extends previous work on flow analyses for this class of programming languages. The analysis is based on abstract interpretation and is parameterized over the abstraction of literals, two polyvariance operators, and a projection operator. The polyvariance operators regulate the accuracy of the analysis while the projection operator regulates the speed. A preliminary implementation of the analysis is incorporated and used in a production-quality Scheme compiler. The analysis can process any legal Scheme program without modification. While it has been demonstrated that analyses at least as accurate as 0CFA are useful for justifying program transformations, an instantiation of this analysis less precise than 0CFA is used to facilitate loop recognition, eliminate the construction of closures, and optimize procedure calls. This demonstrates that relatively inaccurate analyses can still be useful for justifying transformations.

1 Introduction

A flow analysis for a higher-order, mostly-functional language collects data-flow and control-flow information about programs in the language. The information collected can be used to drive program transformations like partial evaluation [9] and closure conversion [22] as well as compiler optimizations like type recovery [19] and the selection of closure representations [18].

There are several abstract interpretation-based approaches to the flow analysis problem [13, 15, 20, 24] and they collectively identify several important aspects of it:

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

These approaches each identify some but not all of the above aspects. This paper develops a parameterized analysis that unifies them all. The analysis is parameterized over the degree of polyvariance and projection as well as the abstract domain of constants.

The analysis is not just a combination of previous results, however. Compared to previous work the analysis is more accurate on assignment, can potentially recover more information from type tests, and can express a wider range of polyvariance. The analysis is also implemented and used in a production-quality incremental compiler for Scheme. The implemented analysis can process any Scheme program without modification. Furthermore, while most work on projection operators has been theoretical, the implementation incorporates a practical and useful projection operator to accelerate the analysis without losing too much information. The analysis is used to justify the optimization of procedure calls and closure representations as well as to facilitate loop recognition.

The rest of the paper is organized as follows. Section 2 develops the analysis by first giving a collecting operational semantics for a canonical 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. It also gives some preliminary results on the cost of the analysis and its usefulness for justifying optimizations. Section 4 describes related work, and Section 5 gives conclusions.

*Author's current address: Snow Hall 415, Lawrence, KS, 66045. E-mail: *jashley@eecs.ukans.edu*.

2 The analysis

The analysis operates over closed programs in the core language given in Figure 1. It is a subset of Scheme with some representative primitives added. While very simple, the analysis developed around this core language can be extended to all of Scheme or ML, including 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 canonical language. The canonical language is continuation-passing style (CPS) A-normal form. The grammar for this language is given in Figure 2. It is like A-normal form [12], but each continuation of a procedure call and conditional is made explicit through conversion to CPS. The continuation of a conditional is made explicit to avoid code duplication, and the syntax for **pair?** anticipates the continuation being evaluated and bound to a temporary variable.

A core program is translated into canonical form by first converting it to A-normal form and then converting to continuation-passing style. The translation process also assigns a unique tag to each lambda expression, and each cons expression is translated to introduce two new variables that do not occur elsewhere in the program. This extra information is used by the analysis. In the CPS conversion, the primitive **halt** is used to define the initial continuation (**lambda** (t) (**halt** t)). An example translation is given in Figure 3.

After conversion to canonical form, each lambda expression corresponds to a program point of the core program. During run time, control flow can enter a lambda expression's body only through a procedure call, and control can leave only through a tail call. In other words, the body of each lambda expression corresponds to a basic block.

Defining the flow analysis on the canonical language follows the usual abstract interpretation-based methodology [10]. 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.

2.1 The collecting machine

The collecting machine is an instrumented variant of a standard machine to execute canonical programs. It differs from a standard machine by building a cache as it executes a program. The meaning of the program is defined to be the cache that exists when the collecting machine reaches a halt state. The cache expresses the set of execution states that arise at each procedure entry point during the execution of the program. Since continuations are represented as procedures, the cache also expresses information about the execution state on procedure return.

The collecting machine is defined in Figure 4. A value is either a literal, a pair, or a closure. A pair is simply a pair of locations. A closure consists of a tag, a sequence of formals, the code for the closure, and an environment. An environment is a finite function mapping variables to locations. A store is a finite function mapping locations to values. The cache is a set of tag, environment, and state triples, each of which describes an execution state on entry to a procedure. A state in the machine is a four tuple consisting of the next expression to execute, an environment, a store, and a cache. The machine halts if the current machine state does not match any of the transition rules or if the meaning of an auxiliary function is undefined. Given a canonical program A, the initial state of the machine is $\langle A, \emptyset, \emptyset, \emptyset \rangle$, where $\emptyset$ denotes an empty finite function or set.

The transition rules are straightforward. Each of the first three rules evaluates a primitive application and binds the result to a temporary variable. The function $[\![\ ]\!]$ is used to evaluate simple expressions, and the function *new* is used to obtain fresh locations that do not occur in the domain of *Stores*. A **pair?** expression is evaluated by binding a continuation to a temporary variable, evaluating the condition, and then using the function *truish*? to select one branch or the other for continuing the evaluation. An application evaluates its operator to get a closure, extends the environment with new locations, and extends the store to bind those locations to the arguments. It then updates the cache with a snapshot of the current execution state and continues with the evaluation of the closure's code.

As given, the collecting machine in fact implements a flow analysis. After the machine terminates successfully, a post processor can use the cache to determine control- and data-flow graphs for the program. 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

To obtain a computable abstraction of the collecting machine, the collecting machine's execution states must must be collapsed into a finite number of approximate states. The collecting machine may produce an infinite number of execution states, because a variable may be mapped to a possibly infinite number of locations. Thus, the store may be infinite. In the abstract ma-

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

Figure 1: The core Scheme language

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

Figure 2: The canonical Scheme language

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


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

Figure 3: The procedure *copy* and its equivalent in canonical Scheme.

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

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

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

Figure 4: Collecting machine

chine, on the other hand, the locations are divided into a finite number of partitions with one representative location per partition. The store is thus rendered finite, but the value bound to each representative must approximate all of the values bound to the locations in its partition.

Figure 5 defines the abstract machine. An abstract store maps a location to an abstract value. An abstract value is a set of objects, and each object is either an abstract literal, pair, or closure. The domain of possible stores for a given program forms a complete partial order $(\widehat{\mathit{Stores}}, \sqsubseteq)$ where $\sqsubseteq$ is defined as follows. Given two abstract stores $\hat{\sigma}$ and $\hat{\sigma}_1$, $\hat{\sigma}_0 \sqsubseteq \hat{\sigma}_1$ if and only if $\mathit{dom}(\hat{\sigma}_0) \subseteq \mathit{dom}(\hat{\sigma}_1)$ and for all $v \in \mathit{dom}(\hat{\sigma}_0)$, $\hat{\sigma}_0(v) \subseteq \hat{\sigma}_1(v)$. Similarly, $\hat{\sigma}_0 \sqcup \hat{\sigma}_1 = \hat{\sigma}_2$ where $\mathit{dom}(\hat{\sigma}_2) = \mathit{dom}(\hat{\sigma}_0) \cup \mathit{dom}(\hat{\sigma}_1)$ and for all $v \in \mathit{dom}(\hat{\sigma}_2)$, $\hat{\sigma}_2(v) = \hat{\sigma}_0(v) \sqcup \hat{\sigma}_1(v)$. Given a variable v and stores $\hat{\sigma}_0$ and $\hat{\sigma}_1$, $\hat{\sigma}_0(v) \sqcup \hat{\sigma}_1(v)$ is defined to be $\hat{\sigma}_0(v) \cup \hat{\sigma}_1(v)$ if $v \in \mathit{dom}(\hat{\sigma}_0)$ and $v \in \mathit{dom}(\hat{\sigma}_1)$. If $v \notin \mathit{dom}(\hat{\sigma}_0)$ then $\hat{\sigma}_0(v) \sqcup \hat{\sigma}_1(v) = \hat{\sigma}_1(v)$, and the behavior when $v \notin \mathit{dom}(\hat{\sigma}_1)$ is symmetric.

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

The execution rules of the abstract machine are similar to those of the collecting machine. The abstract machine rules must simply take into account that a value is now a set of objects instead of a single object. Simple expressions are evaluated by injecting them into a singleton set using the abstraction function α to map constants in the domain *Constants* to the domain $\widehat{\mathit{Constants}}$. By customizing α the analysis can be instantiated to abstract over different properties of the basic datatypes. A **cons** expression evaluates to a singleton set consisting of one pair. Both the **car** and **set-car!** rules must anticipate their first argument evaluating to more than one pair.

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

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

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

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

Figure 5: Abstract machine

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

The function $\widehat{new}$ guarantees that the machine's environments and abstract stores are finite and is also used to partially regulate the accuracy of the machine. Like *new*, $\widehat{new}$ is a deterministic function that returns a sequence of locations that can be used to create new bindings. $\widehat{new}$ is restricted, however, to deliver only a finite number of locations. Although the range must be finite, $\widehat{new}$ can still affect the accuracy of the machine. Assuming the program has been α-converted, a 0CFA analysis can be obtained by setting *Locations* = *Variables* and setting $\widehat{new}$ to be the identity function on its first argument. A more accurate analysis can be obtained, however, by using a more discriminating function. For example, a 1CFA analysis can be obtained by modifying $\widehat{new}$ to allocate different locations depending on the textual context of the environment extension.

The accuracy of the machine is also affected by the polyvariant function π used by *apply*. π is a partitioning function that maps an abstract store to an element of the finite set *Indices*. Assuming that $\widehat{new}$ is defined appropriately, setting π to the constant function would yield a 0CFA analysis. Similar to $\widehat{new}$, a more accurate analysis can be obtained by a nontrivial partitioning. For example, a binding-time analysis may choose to partition the continuation of an assignment based on whether the variable was assigned a static or dynamic value.

While $\widehat{new}$ and π are used to increase the accuracy of the analysis, the projection operator Θ is used by *apply* to increase the speed. A projection operator Θ is a function from $\widehat{Stores}$ to $\widehat{Stores}$ such that for all $\hat{\sigma} \in \widehat{Stores}$, $\hat{\sigma} \sqsubseteq \Theta(\hat{\sigma})$. By projecting an abstract store to a more general abstract store, the convergence to a stable cache is accelerated. Using a projection operator may cause the analysis to generalize beyond the most accurate solution, *i.e.*, the least fixed point, but in exchange the analysis may be faster.

There are two extremes for projection operators. At one end the projection operator can be just the identify function, and the accuracy of the analysis is not sacrificed at all. At the other end, Θ can be defined such that for all $\hat{\sigma}$, $\Theta(\hat{\sigma}) = \top$. This yields a very fast analysis that collects little useful information. An example of a more useful projection operator is given in Section 3.

2.3 An example

Consider the canonical Scheme program in Figure 3 and assume a 0CFA analysis in which the abstraction function α is the identity function and $\widehat{new}$ is the identity function on its first argument. Upon termination of the abstract machine, a portion of the abstract store on entry to the initial continuation would look like:

$$\begin{aligned} v_6 &= \{\langle v_{11}, v_{12}\rangle, ()\} \\ v_{11} &= \{1, 2\} \\ v_{12} &= \{\langle v_{11}, v_{12}\rangle, ()\} \end{aligned}$$

In particular, the final answer is a list of indeterminable length, and the car of each pair is either 1 or 2. Given that the exact answer is the list (1 2), this abstract value is indeed a safe approximation of the exact answer.

2.4 Correctness

The correctness of the abstract machine has two aspects: termination and safety. In particular, the machine should terminate for all input programs, and upon termination, the machine's cache should be a safe approximation to the collecting machine's cache.

For a given program A and finite domain $\widehat{Constants}$, the machine terminates when the pending set is empty, so the machine terminates only 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 finite and form a complete partial order (CPO) under *sqsubseteq*. Since the program is finite, *Indices* is finite, and the number of abstract stores is finite, there are only a finite number of program points, and Since the store at a program point changes monotonically, only a finite number of programs points can be added to the pending set. Hence, the machine always terminates.

The correctness of the analysis is characterized with the following theorem.

Theorem 1 *Let A be a program in canonical Scheme, and let $\langle A, \rho, \sigma, \gamma\rangle$ and $\langle A, \hat{\rho}, \hat{\sigma}, \hat{\gamma}, \emptyset\rangle$ be halt states of the collecting and abstract machines respectively such that $\langle A, \emptyset, \emptyset, \emptyset\rangle \longrightarrow \langle A, \rho, \sigma, \gamma\rangle$ in the collecting machine and $\langle A, \emptyset, \emptyset, \emptyset, \emptyset\rangle \longrightarrow \langle A, \hat{\rho}', \hat{\sigma}, \hat{\gamma}, \emptyset\rangle$ in the abstract machine. Then $\hat{\gamma}$ is a safe approximation of γ.*

Proof Sketch: The proof proceeds using storage layout relations [21], which is a proof technique for proving that one machine is a *refinement* of another. In this case, the abstract machine is a refinment of the collecting machine. The proof proceeds by induction on the execution steps of the collecting and abstract machines. The induction hypothesis states that on each step, the abstract machine maintains a cache and pending set that approximates the cache of the collecting machine.

2.5 Discussion

Not surprisingly, the abstract machine can be parameterized to emulate the collecting machine. The $\widehat{new}$ function must be modified to mimic *new*, a constant function is used for π, and the identity function is used for the projection operator Θ and the abstraction operator α. Under this instantiation, all abstract values are singleton sets containing precisely one object. The pending set Σ is always empty on procedure entry, but one triple is added to it on procedure application or a conditional test. This one element is immediately removed by *pop*.

The abstract machine does not explicitly use type tests to constrain abstract values. Usually, the constraint 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. When control leaves the lexical context of the conditional, the variable reverts to its former binding. This strategy is easily incorporated into the abstract machine.

Constraining abstract values in the arms of conditionals is useful, but more can be done. Consider these two expressions that may result from macro expansion or procedure inlining.

```
(lambda (x)
  (if (pair? x) #f (error))
  (if (pair? x) (car x) (error)))
```

```
(lambda (x)
  (+ (if (positive? x) x (abs x))
     (if (positive? x) x (abs x))))
```

Assuming nothing is known about how these procedures are used, the redundant type checks in both examples cannot be eliminated using the above strategy for constraining abstract values.

Because the abstract machine retains a store, it is possible to mutate the store to constrain abstract values beyond the lexical scope of conditionals. The precise extent of the constraint is unclear, however, because of the "environment problem" [17, page 67], which refers to the problem of a value being illegally constrained in some contexts because of the fact that multiple environments are folded together to render the analysis computable. Nevertheless, it is at least possible to lengthen the extent of a constrained value until control leaves the procedure in which the value is constrained. In particular, doing so enables the redundant tests in the above two examples to be removed in a type-check elimination pass.

Using both $\widehat{new}$ and π to regulate the accuracy of the analysis is not redundant. The two operators have different properties. $\widehat{new}$ is able to partition a variable's value at the point the variable is bound. This is done by binding the variable to different locations in the environment and storing the segregated values in different locations. Splitting in this way has the advantage that the segregation survives join points. The π function, on the other hand, is able to partition the values of any location at any time. This is more general, but the price is that the segregation does not survive join points.

Both operators are useful. $\widehat{new}$ has been used to segregate **let**-bound variables by type, leading to effective elimination of run-time type checks [15]. It has also been used to segregate the components of pairs to enable good specialization in off-line partial evaluation [9]. In general, $\widehat{new}$ is useful for "opportunistic" polyvariance since segregation happens at the binding point. π, on the other hand, is more useful for "by-need" polyvariance to segregate values closer to the point at which they are used. Since the segregation is more likely to happen only when it is profitable, by-need polyvariance may lead to cheaper but nevertheless effective analyses.

3 Implementation

A preliminary implementation of the analysis is used in the Chez Scheme [7] compiler to justify program optimizations. The compiler processes R^{4}RS Scheme [8] and directly supports multiple return values [1] and a variable arity procedure interface [11].

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 implementation of the analysis therefore handles free variable references by introducing a unique abstract value $\{unknown\}$ to denote an unknown value. In the abstract machine, a free variable reference evaluates to $\{unknown\}$.

When *unknown* is used as a closure in an application, the arguments of the application must 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 every mutable location has the abstract value $\{unknown\}$.

The goal of our initial experiments is to determine whether analyses that are faster but less precise than 0CFA can justify useful optimizations. That 0CFA is useful has been established in previous work [13, 14, 15, 20]. Since the analysis is targeted for both interactive and noninteractive use, however, it is important to know if a faster but less accurate instantiation of the analysis is still useful.

To determine this, the analysis is instantiated to employ the identity function for $\widehat{new}$ and a nontrivial projection operator. The projection operator tracks the number of times the cache has been updated at each program point. If the number exceeds a threshold n for some point, the cache at that point is not updated with the new store. Instead, the new values responsible for the update are considered escaping, and $\{unknown\}$ is substituted in their place. The only exception is that *unknown* is not added to the values of **let**- and **letrec**-bound variables in the source program. This is safe since the values of those variables can never change. If the threshold is n, this projection operator limits the analysis to a worst case of $n + 1$ passes over the program. Setting $n = 0$ results in a one-pass intraprocedural analysis. Setting $n = \infty$ results in a 0CFA analysis.

The information lost when the analysis is forced to generalize is not severe. There are two ways in which the loss is contained. First, only the information about unstable portions of the code is generalized beyond the least fixpoint. For example, suppose $n = 5$ and the following program is analyzed.

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

Benchmark	lines	Description
compiler	30,000	Chez Scheme recompiling itself
texer	3,000	Scheme pretty-printer with TEX output
softscheme	10,000	Wright's soft typer [23] checking a 2,500 line program
similix	7,000	self-application of the Similix [2] partial evaluator
ddd	15,000	hardware derivation system [3] deriving Scheme machine [6]
em-fun	490	EM clustering algorithm in functional style
em-imp	460	EM clustering algorithm in imperative style
dynamic	2,200	a dynamic type inferencer applied to itself
graphs	500	counts the number of directed graphs with particular properties
lattice	200	enumerates the lattice of maps between two lattices
nbody	850	Greengard multipole algorithm for computing gravitational forces
tcheck	260	a polymorphic type inferencer for Scheme

Table 1: Benchmarks

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.

The other way in which the information loss is contained is that when the threshold is exceeded at a program point, information about what is already known at that point is not discarded. Combined with type recovery [16, 19], the flow analysis can still yield useful information for program transformations. For example, suppose $n = 1$ and the following program is analyzed.

```
(letrec ((fac (lambda (x)
                (if (= x 0)
                    1
                    (* x (fac (- x 1)))))))
  (fac 5))
```

Upon termination of the analysis, the abstract value of x is $\{5, \textit{unknown}\}$ when control enters the body of *fac*. In the arms of the conditional, however, the value of x can be constrained to be $\{5, \textit{integer}\}$. The analysis also determines that *fac* is a procedure, and the call (*fac* $(- x\ 1)$) is recursive. An aggressive optimizer might also use the fact that x may be bound to 5 to generate code for a specialized version of *fac* in the case that its input is a fixnum.

The compiler uses the results of the analysis to optimize direct calls, eliminate some closures, and facilitate loop recognition. A direct call is a procedure call where the operator will always evaluate to the same procedure. Direct calls can be optimized by branching directly to the code for the called procedure and in some cases avoiding loading the procedure's closure pointer. Generated code can avoid building a closure if it is not needed. A closure is not needed if it does not have free variables, it does not escape, and all calls to it are direct calls. Loop recognition is done syntactically with the results of the analysis used to verify that all recursive calls to the loop head are in tail position.

For each benchmark in Table 1, Table 2 shows how much optimization was enabled by the flow analysis at $n = 0$ and $n = \infty$. For each of the three optimizations, the number of candidates considered is given along with the number optimized at $n = 0$ and $n = \infty$. For detecting direct calls, the number of candidates is the number of applications in the benchmark code. For closure elimination, the number of candidates is the number of **let**- and **letrec**-bound lambda expressions. For loop detection, the number of candidates is the number of potential loops recognized syntactically.

Even though the analysis is much more accurate at $n = \infty$, the more precise information was not sufficient to enable significantly more optimization. This is a consequence of the chosen optimizations which are particularly sensitive to the accuracy of the analysis. For example, to make a direct call the analysis must be able to determine that exactly one procedure arrives at a call site. It makes no difference if a more accurate analysis can reduce the number of procedures from four to two; the information is still not precise enough. For other optimizations such as type-check elimination, the more accurate information collected at $n = \infty$ versus $n = 0$ would likely justify the elimination of significantly more checks.

Table 3 shows the analysis costs and the benefits of the optimizations for each benchmark. The analysis cost includes the time necessary to rewrite the source program to canonical form, to run the analysis, to annotate the program with the collected flow information, and to convert the program back to the compiler's intermediate form. The stock compiler justifies the op-

	closures eliminated			direct calls			type checks eliminated		
	max	$n = 0$	$n = \infty$	max	$n = 0$	$n = \infty$	max	$n = 0$	$n = \infty$
compiler	1955	536	571	18310	4893	4902	438	311	311
ddd	778	156	186	4967	627	617	147	58	58
similix	865	225	231	7047	1789	1808	100	46	46
softscheme	971	19	21	10001	771	772	57	13	13
texer	134	20	20	1075	304	304	48	39	39

Table 2: The table gives the number of closures eliminated, direct calls recognized, and loops recognized at $n = 0$ and $n = \infty$. The maximum numbers in each case indicate the number of candidates considered.

	compile time (sec)				run-time
	analysis		total		
	$n = 0$	$n = \infty$	$n = 0$	$n = \infty$	speedup
compiler (30,000)	11.4	34.0	42.7	65.7	18%
ddd (15,000)	1.7	1.9	9.9	9.3	4%
similix (7,000)	3.4	8.0	15.2	19.7	1%
softscheme (10,000)	7.9	12.4	51.0	53.9	6%
texer (3,000)	0.7	1.8	2.6	3.8	23%

Table 3: The table measures the compile-time cost of the analysis and the run-time performance increase after enabling the optimization.

timizations using the results of a complex and *ad hoc* analysis. That compiler was modified by eliminating the analysis and restructuring the compiler to incorporate the new flow analysis. At $n = 0$, the flow analysis recovers the same information as the *ad hoc* analysis. The speedups were computed by comparing optimized versions of each benchmark against an unoptimized baseline. The difference in flow information computed by the analysis at $n = 0$ and $n = \infty$ had an immeasurable impact on the speedups of the optimized benchmarks. Hence, the average of the speedups for $n = 0$ and $n = \infty$ is reported.

These preliminary results are very encouraging. On average the cost of the analysis at $n = 0$ was 21% of compile time and at $n = \infty$ it was 38% of compile time. These numbers are not unreasonable considering that the implementation is preliminary and not optimized, and that the stock compiler is tuned for compile-time speed. The projection operator chosen was successful in lowering the analysis cost in every case and in some cases lowering it significantly. Despite such a coarse projection operator, the information collected was still able to justify optimizations. Indeed, the extra information collected at $n = \infty$ had no practical impact on the compiler's ability to optimize the benchmarks. We may conclude then that for some transformations, cheaper analyses can be used profitably.

4 Related work

Shivers [19] and Harrison [13] describe flow analyses for Scheme that are based on abstract interpretation. They differ primarily in the details of the source language and the range of accuracy they can express. Our analysis draws from the advantages of each approach. Like Shivers', we use CPS to make control transfers explicit. Like Harrison's, primitive operations are ordered and bound to temporary variables, and an abstract program state is computed for each basic block of the program. Our analysis subsumes their analyses by expressing a wider range of polyvariance and using projection to accelerate convergence to a fixpoint. Also, their analyses are prototypes that do not handle the full language, while our analysis is completely implemented and can analyze arbitrary Scheme programs.

Jagannathan and Weeks [15] describe an analysis for the polyvariant analysis of higher-order applicative programs that also accommodates side-effects and **call/cc**. Their analysis is parameterized over a polyvariance operator, but it is not parameterized over a projection operator. Also, their characterization of program state is different from ours. In our analysis, the program state is an environment and store. In their analysis, the program state is an environment mapping variables to locations in a global store. Their characterization of program state has consequences on polyvariance and the accuracy of assignment. An assignment to a location pollutes the location's dataflow by adding the assigned value to the dataflow at the point at which the location is bound instead of at the point of assignment. In our analysis, the assigned value replaces the old value of the location in the continuation of the assignment. With respect to polyvariance, their approach limits how program points can be split, since it is not possible to use the π function to split on arbitrary locations at any time. In both cases, our approach implies that more accurate analyses can be expressed.

Other researchers have observed that projection operators can be used in practice to reduce the number of iterations needed to stabilize. Yi and Harrison [24] describe a framework for the automatic generation of abstract interpreters. This framework incorporates a notion of projection that is similar to ours. The difference is that they apply projections to values, and we apply them to the entire computation state. Furthermore, they always project a value to the top of the value lattice, while we permit the operator to project a value to any other value above it in the lattice. Bourdoncle [4, 5] uses widening operators in a theoretical setting to accelerate convergence to a solution. A widening operator ∇ is a substitute for the least upper bound operator. The constraint is that given two points x and y, $x \sqcup y \sqsubseteq x \nabla y$. Both of these approaches the promise the potential to accelerate the analysis without losing too much information. Our work delivers on that promise by exhibiting a practical projection operator that still enables useful optimizations.

5 Conclusions

A problem with incremental compilers is that they must often sacrifice generated code quality for compile-time speed. There are two obvious solutions to the problem. One is to use a "batch mode" compiler when development is complete in order to generate good code. The other is to 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 detect undesirable interactions among optimization passes.

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.

It is not strictly necessary to translate source programs to canonical form before analysis but doing so has several advantages. First, it simplifies the abstract machine considerably. In some sense, the translation step can be seen as deriving a set of constraints, and the abstract machine simply views the constraints as an executable specification. Also, because the machine makes the solution algorithm explicit, an improvement to the algorithm can be proven correct by implementing the improvement as a revised state machine and then proving that the revised machine is a correct refinment of the original machine. Translating to a canonical language is also an advantage in situations where the flow analysis is the core analysis engine for several optimizations in a compiler. Typically, a compiler will alter the abstract syntax tree of the program on each step of the compilation. By compiling to an intermediate form and then performing the flow analysis, the differences in the intermediate representations are isolated from the analysis.

Adding a module system to the language would improve the accuracy of the analysis. Currently, the analyzer assumes no information about the values of free variables in programs. 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.

References

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

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

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

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

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

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

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

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

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

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

[11] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, September 1990.

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

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

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

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

[16] Suresh Jagannathan and Andrew Wright. Effective flow-analysis for avoiding runtime checks. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 207–224. Springer-Verlag, September 1995.

[17] Peter Lee, editor. *Topics in Advanced Language Implementation*. MIT Press, 1991.

[18] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 130–161, 1994.

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

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

[21] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 151–160, 1992.

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

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

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