Set-Based Analysis for Full Scheme and Its Use in Soft-Typing

Cormac Flanagan
Matthias Felleisen

Rice COMP TR95-254

October 1995

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Set-Based Analysis for Full Scheme and Its Use in Soft-Typing

Cormac Flanagan* Matthias Felleisen*

Department of Computer Science,
Rice University,
Houston, TX 77251-1982

Abstract

Set-Based Analysis is an efficient and accurate program analysis for higher-order languages. It exploits an intuitive notion of approximation that treats program variables as sets of values. We present a new derivation of set-based analysis, based on a reduction semantics, that substantially simplifies previous formulations. Most importantly, the derivation easily extends from a functional core language to include imperative features such as assignments and first-class continuations, and supports the first correctness proof of set-based analysis for these imperative features. The paper includes an implementation of the derived analysis for a Scheme-like language, and describes a soft-typing algorithm that eliminates type-checks based on the information produced by the analysis.

*Supported in part by NSF grant CCR 91-22518 and a sabbatical at Carnegie Mellon University.

Contents

1 Introduction

Advanced compiler optimizations heavily rely on static information about the values that program variables may assume. Computing such information for higher-order languages such as Scheme [2] and ML [16] is particularly complex, since a static control-flow graph of a program is not readily available at compile time [19, 22]. Heintze [10, 11] pioneered a *set-based* approach to program analysis that controls this complexity in two ways. First, the approach is based on a single notion of approximation that treats program variables as sets of values. Second, the approach splits the analysis into two phases: a *specification* phase and a *solution* phase. During the specification phase, the analysis derives *constraints* on the sets of values that program variables may assume. These constraints approximate the dataflow relationships of the analyzed program. During the solution phase, the analysis produces sets of values that satisfy these constraints. The result is a valid approximation of the value sets for the program variables.

In contrast to the simplicity of the analysis, Heintze's *derivation* of set-based analysis is complex. The derivation of the specification phase involves a number of intermediate steps. Specifically, Heintze uses the "natural" semantics framework to define a *set-based* "natural" semantics, from which he reads off "safeness" conditions on set environments. He then shows how to derive set constraints from the set-based semantics and proves that a solution of these constraints is a safe set environment. The derivation of the solution phase is also complex. It involves solving constraints on the infinite sets of values that program variables may assume. Heintze represents these infinite sets using a restricted form of set constraints, which can be regarded as regular tree grammars, and his solution phase produces a regular tree grammar representation of the solution of the set constraints.

Not only is Heintze's derivation complex, it is also limited to the analysis of purely functional languages. Extending it beyond a simple functional core is difficult, mostly due to the use of "natural" semantics, which cannot easily accommodate destructive data structure manipulations and non-local control operators. Although suitable set constraints for these imperative features can be produced in an ad-hoc manner, there is no semantic foundation for these constraints. Thus existing implementations of set-based analysis for languages like Scheme and ML [10, 11, 8, 7], which include assignments and non-local control operators, are not semantically well-founded.

We present a simpler derivation of set-based analysis than Heintze's. It avoids the complexities and limitations outlined above, and explains the underlying notion of approximation in a clear manner. Our approach is based on a reduction semantics that explicitly models the allocation of objects in a global heap [5].[1] We avoid Heintze's intermediate steps by deriving our set constraints and proving their correctness *directly* from the reduction semantics. Our semantics also records sufficient information to associate each run-time value with the corresponding syntactic value in the source program. We exploit this information to express our set constraints in terms of finite sets of syntactic values in the source program, thus avoiding the need to reason about constraints on infinite sets of values.

Since our heap-based semantics represents values as general "graphs", it is straightforward to extend both the semantics and the analysis to a language with assignments. Similarly, because the complete control state is available for each step of the evaluation,

[1] Also see [17] for an application of this idea to model and verify garbage collection algorithms.

the extension to a language with non-local control constructs is also straightforward. Our derivation supports the first soundness proof of set-based analysis for a language containing these imperative features.

The presentation of our results proceeds as follows. The second section presents the syntax and semantics of a simple functional language, and the third section describes the set-based analysis of that language. The fourth and fifth sections shows how the analysis extends to assignments and first-class continuations, respectively. The sixth section describes an application of set-based analysis known as *soft-typing*, and Section 7 discusses related work. The appendix includes a simple implementation of the derived analysis for a Scheme-like language containing both assignments and first-class continuations.

2 An Idealized Intermediate Language

2.1 Syntax

Our motivation in performing program analyses is to produce information for the optimization phases of a compiler. Compilers typically convert source programs into a simple intermediate representation, and then proceed to analyze and optimize the intermediate representation of the program. Therefore, we formulate the analysis for an intermediate representation of an idealized functional language. Specifically, we use the subset of A-normal forms [9] of a λ-calculus-like language with a **let** construct and conditional expressions:[2] see Figure 1. The language also includes the primitives **cons**, **car**, and **cdr** for list manipulation, which will serve to illustrate the treatment of primitive operations, and a basic constant `nil` denoting the empty list.

The key property of terms in A-normal form is that each intermediate value is explicitly named and that the order of execution follows the lexical nesting of **let**-expressions. The use of A-normal forms thus simplifies the formulation of the semantics and facilitates the compile-time analysis of programs since every intermediate value is named [20].

Values are either simple values or heap-allocated values. Simple values are either variables or locations. Locations do not occur in source programs; they are only created during program evaluation. Heap-allocated values include procedures $(\lambda x.\, M)$, pairs of values (**cons** v v) and basic constants. Procedures and **let**-expressions are *binding constructs*, and we assume that all binding constructs in a program introduce distinct variables. A variable occurrence is *free* if it is not bound by an enclosing binding construct, and a term is *closed* if it contains no free variables. We use Λ_a^0 to denote the set of location-free, closed terms, or *programs*.

Notation We use the following notations throughout the paper: $\mathcal{P}_{\text{fin}}$ denotes the finitary power-set constructor; $f : A \longrightarrow B$ denotes that f is a total function from A to B; and $M \in P$ denotes that the term M occurs in the program P.

[2] Our language follows the Scheme and Lisp tradition of using conditional expressions instead of a pattern-matching construct. The use of conditional expressions, predicates and selectors enables more opportunities for functional abstraction than the pattern matching approach, but imposes more work on the analysis phase.

Programs:

$$
\begin{array}{rcllr}
M, N & \in & \Lambda_a & ::= & v \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Expressions)} \\
 & & & & \mid (\textbf{let}\ (x\ h)\ M) \\
 & & & & \mid (\textbf{let}\ (x\ (\textbf{car}\ v))\ M) \\
 & & & & \mid (\textbf{let}\ (x\ (\textbf{cdr}\ v))\ M) \\
 & & & & \mid (\textbf{let}\ (x\ (\textbf{if}\ v\ M\ M))\ M) \\
 & & & & \mid (\textbf{let}\ (x\ (\textbf{apply}\ v\ v))\ M) \\
 & & & & \mid (\textbf{let}\ (x\ M)\ M) \\
v & \in & Sval & ::= & x \mid l \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Simple Values)} \\
x & \in & Vars & = & \{x, y, z, \ldots\} \qquad\qquad\qquad\qquad\qquad\quad \text{(Variables)} \\
l & \in & Loc & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{(Locations)} \\
h & \in & Hval & ::= & (\lambda x. M) \mid (\textbf{cons}\ v\ v) \mid c \qquad\qquad \text{(Heap Values)} \\
c & \in & Const & = & \{\texttt{nil}, \ldots\} \qquad\qquad\qquad\qquad \text{(Basic Constants)}
\end{array}
$$

FIGURE 1: The A-normalized Intermediate Language

2.2 Semantics

We define the semantics of programs via an abstract machine (see Figure 2) that explicitly models the allocation of objects into a global heap [17, 5]. A heap is a set of bindings from locations to heap values. The set *Loc* of locations contains an infinite subset Loc_x reserved for each variable x. The function *new* takes a heap and a variable x, and returns a new location from Loc_x that is not already allocated in the heap. The operation $M[x \leftarrow l_x]$ denotes the substitution of the location l_x for all occurrences of x within M, and the operation $h[x \leftarrow l_x]$ is defined analogously.

Each state of the machine consists of a heap and program term, except for the special state `error`. The machine evaluates programs by stepping through a sequence of states according to a set of transition rules. Each transition rule of the machine defines the semantics of a particular class of expressions. For example, the transition rule (*car*) evaluates expressions of the form $(\textbf{let}\ (x\ (\textbf{car}\ l_y))\ M)$. If the heap value at location l_y is a pair, then the transition rule extracts the first component of that pair, stores that component at a new location l_x in the heap, and replaces all occurrences of x within M by l_x. Alternatively, if the heap value at location l_y is not a pair, then the transition rule (*car*) raises an error. Each of the remaining transition rules has a similar intuitive explanation.

The definition of these transition rules relies on the notion of an *evaluation context*. An evaluation context E is a term with a "hole" [ ] in place of the next subterm to be evaluated. For example, in the term $(\textbf{let}\ (x\ M)\ N)$, the next expression to be evaluated lies within M, and thus the definition of evaluation contexts includes the clause $(\textbf{let}\ (x\ E)\ M)$.

The evaluator *eval* is a function from programs to results. A result is either a basic constant, or one of the tags `procedure` or `cons`, indicating that the program returned an abstraction or a pair respectively, or the tag `error`, indicating that the program raised an error, or $\perp$, indicating that the program diverges.

Evaluator:

$$eval : \Lambda_a^0 \longrightarrow Const \cup \{\texttt{procedure}, \texttt{cons}, \texttt{error}, \bot\}$$

$$eval(P) = \begin{cases} c & \text{if } \langle \emptyset, P \rangle \longmapsto^* \langle H, l \rangle \text{ and } H(l) = c \\ \texttt{procedure} & \text{if } \langle \emptyset, P \rangle \longmapsto^* \langle H, l \rangle \text{ and } H(l) = (\lambda x.\, M) \\ \texttt{cons} & \text{if } \langle \emptyset, P \rangle \longmapsto^* \langle H, l \rangle \text{ and } H(l) = (\mathbf{cons}\ l_1\ l_2) \\ \texttt{error} & \text{if } \langle \emptyset, P \rangle \longmapsto^* \texttt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists S_i \text{ such that } \langle \emptyset, P \rangle \longmapsto^i S_i \end{cases}$$

Domains:

$$\begin{array}{rcllr} S & \in & State & ::= \langle H, M \rangle \mid \texttt{error} & \text{(States)} \\ H & \in & Heap & ::= \{l_1 = h_1, \ldots, l_n = h_n\} & \text{(Heaps)} \\ l & \in & Loc & ::= \bigcup_{x \in Vars} Loc_x & \text{(Locations)} \\ l_x & \in & Loc_x & & \text{(x-Locations)} \\ E & \in & EvalCtxt & ::= [\,] \mid (\mathbf{let}\ (x\ E)\ M) & \text{(Evaluation Contexts)} \end{array}$$

Transition Function:

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ c)\ M)\,]\rangle & \\ \quad \longmapsto \langle H \cup \{l_x = c\}, E[\,M[x \leftarrow l_x]\,]\rangle & (\mathit{bind\text{-}c}) \\ \quad\quad \text{where } l_x = new(H, x) & \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\lambda w.\, N))\ M)\,]\rangle & \\ \quad \longmapsto \langle H \cup \{l_x = (\lambda w.\, N)\}, E[\,M[x \leftarrow l_x]\,]\rangle & (\mathit{bind\text{-}lam}) \\ \quad\quad \text{where } l_x = new(H, x) & \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\mathbf{cons}\ l_y\ l_z))\ M)\,]\rangle & \\ \quad \longmapsto \langle H \cup \{l_x = (\mathbf{cons}\ l'_y\ l'_z), l'_y = H(l_y), l'_z = H(l_z)\}, evcxt M[x \leftarrow l_x]\rangle & (\mathit{bind\text{-}cons}) \\ \quad\quad \text{where } l_x = new(H, x),\ l'_y = new(H, y),\ l'_z = new(H, z) & \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\mathbf{car}\ l_y))\ M)\,]\rangle & \\ \quad \longmapsto \begin{cases} \langle H \cup \{l_x = H(l_{z_1})\}, E[\,M[x \leftarrow l_x]\,]\rangle & \text{if } H(l_y) = (\mathbf{cons}\ l_{z_1}\ l_{z_2}) \\ \quad \text{where } l_x = new(H, x) & \\ \texttt{error} & \text{if } H(l_y) \neq (\mathbf{cons}\ l_{z_1}\ l_{z_2}) \end{cases} & (\mathit{car}) \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\mathbf{cdr}\ l_y))\ M)\,]\rangle & \\ \quad \longmapsto \text{analogous to } (\mathit{car}) & (\mathit{cdr}) \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\mathbf{if}\ l_y\ M_1\ M_2))\ M)\,]\rangle & \\ \quad \longmapsto \begin{cases} \langle H, E[\,(\mathbf{let}\ (x\ M_1)\ M)\,]\rangle & \text{if } H(l_y) \neq \texttt{nil} \\ \langle H, E[\,(\mathbf{let}\ (x\ M_2)\ M)\,]\rangle & \text{if } H(l_y) = \texttt{nil} \end{cases} & (\mathit{if}) \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ (\mathbf{apply}\ l_y\ l_z))\ M)\,]\rangle & \\ \quad \longmapsto \begin{cases} \langle H \cup \{l_w = H(l_z)\}, E[\,(\mathbf{let}\ (x\ N[w \leftarrow l_w])\ M)\,]\rangle & \text{if } H(l_y) = (\lambda w.\, N) \\ \quad \text{where } l_w = new(H, w) & \\ \texttt{error} & \text{if } H(l_y) \neq (\lambda w.\, N) \end{cases} & (\mathit{apply}) \end{array}$$

$$\begin{array}{ll} \langle H, E[\,(\mathbf{let}\ (x\ l_y)\ M)\,]\rangle & \\ \quad \longmapsto \langle H \cup \{l_x = H(l_y)\}, E[\,M[x \leftarrow l_x]\,]\rangle & (\mathit{bind\text{-}v}) \\ \quad\quad \text{where } l_x = new(H, x) & \end{array}$$

FIGURE 2: The CS-Machine

3 Set-Based Analysis

The goal of set-based analysis is to produce information about the sets of values that program variables may assume during an execution. In general, the exact value set for each program variable is not computable, and therefore some notion of approximation is needed. Set-based analysis is based on an intuitive approximation that treats program variables as sets of values, and ignores the information regarding which specific value a variable represents at a given point in an execution. Since each value associated with a variable corresponds to a distinct location in our semantics, we express this notion of approximation by merging all locations allocated for each variable into a single *abstract location*. That is, we use each variable x as an *abstract location* representing all the locations in Loc_x that may be allocated for x. Thus, if P is the program being analyzed, then we use the set $Vars_P$ of variables occurring in P as the set of abstract locations for the analysis of P.

We extend this notion of representation from variables to heap values as follows: Let $Hval_P$ be the set of heap values that occur in the program P. Each heap value produced during the execution of P is obtained by substitution from a syntactic heap value in $Hval_P$. Therefore, we use each heap value h in $Hval_P$ as an *abstract value* representing all *substitution instances* of h, *i.e.*, all heap values obtained from h by substituting appropriate locations for the free variables of h. We call the resulting set of heap values the *substitution closure* of the heap value h. The substitution closure of a set of heap values is defined in a straightforward manner.

Definition 3.1. (*Substitution Closure*)

- For $h \in Hval$ with $FV[h] = \{x_1, \ldots, x_n\}$, the *substitution closure* of h is

$$Cl(h) = \{h[x_1 \leftarrow l_{x_1}, \ldots, x_n \leftarrow l_{x_n}] \mid l_{x_i} \in Loc_{x_i}\}$$

- For $A \subseteq Hval$, the *substitution closure* of A is

$$Cl(A) = \bigcup_{h \in A} Cl(h)$$

∎

Our analysis produces a finite table, called a *set environment*, that maps each program variable to a set of syntactic heap values from $Hval_P$. A set environment $\mathcal{E}$ is *valid* for the program P if, for each variable x, each heap value associated with x during an execution is in the substitution closure of $\mathcal{E}(x)$.

Definition 3.2. (*Set environments, Validity*) Let P be a program.

- A mapping $\mathcal{E} : Vars_P \rightarrow \mathcal{P}_{\text{fin}}(Hval_P)$ is a *set environment* for P.
- The relation $P \models \mathcal{E}$ (read "*P validates $\mathcal{E}$*", or "*$\mathcal{E}$ is valid for P*") holds if $\langle \emptyset, P \rangle \longmapsto^* \langle H, M \rangle$ implies that $H \models \mathcal{E}$.
- The relation $H \models \mathcal{E}$ holds if for all bindings $(l_x = h) \in H$, $h \in Cl(\mathcal{E}(x))$.

∎

We develop an analysis that produces valid set environments in two steps. To develop the *specification* phase of the analysis, we derive constraints on the sets of values that program variables may assume and prove the soundness of these constraints with respect to the abstract machine semantics. For the *solution* phase of the analysis we develop an algorithm that solves these constraints.

3.1 Deriving Set Constraints

Given a program P, a set constraint is of the form:

$$\frac{A_P}{B}$$

where A_P and B are statements concerning set environments, and A_P also depends on the program. A set environment $\mathcal{E}$ *satisfies* this constraint if whenever A holds for $\mathcal{E}$ and P, then B also holds for $\mathcal{E}$. We show how, for each kind of term in P, the analysis must add certain constraints to the global set of program constraints:

- $(\mathbf{let}\ (x\ (\lambda w.\, N))\ M)$

  During execution, this source term will correspond to a run-time term $(\mathbf{let}\ (x\ (\lambda w.\, N'))\ M')$, where N' and M' are substitution instances of N and M respectively. To record that x may be bound to a substitution instance of $(\lambda w.\, N)$, we require that $\mathcal{E}$ satisfies the following constraint:

$$\frac{(\mathbf{let}\ (x\ (\lambda w.\, N))\ M) \in P}{(\lambda w.\, N) \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}lam}})$$

- $(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M)$

  During execution, this source term will correspond to a run-time term $(\mathbf{let}\ (x\ (\mathbf{cons}\ l_y\ l_z))\ M')$, where M' is a substitution instance of M. To record that x may be bound to a heap value $(\mathbf{cons}\ l'_y\ l'_z)$, we require that $\mathcal{E}$ satisfies the following constraint:

$$\frac{(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M) \in P}{(\mathbf{cons}\ y\ z) \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}lam}})$$

- $(\mathbf{let}\ (x\ h)\ M)$

  During execution, this source term will correspond to a run-time term $(\mathbf{let}\ (x\ h')\ M')$, where h' and M' are substitution instances of h and M respectively. To record that x may be bound to a substitution instance of h, we require that $\mathcal{E}$ satisfies the following constraint:

$$\frac{(\mathbf{let}\ (x\ h)\ M) \in P}{h \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}cons}})$$

- (**let** $(x$ (**car** $y))\ M)$

  During execution, this term in the source program will correspond to the run-time term (**let** $(x$ (**car** $l_y))\ M')$, where l_y is some location for y and M' is a substitution instance of M. If the heap maps the location l_y to a pair (**cons** $l_{z_1}\ l_{z_2}$), then the evaluation of this expression will associate x with the heap value at location l_{z_1}. If we assume that $\mathcal{E}$ approximates the value sets of the program variables y and z_1, then $\mathcal{E}(y)$ includes (**cons** $z_1\ z_2$), and $\mathcal{E}(z_1)$ approximates the heap value at l_{z_1}. Therefore, we ensure that $\mathcal{E}$ accounts for the bindings created during the evaluation of this expression by demanding that $\mathcal{E}$ satisfies the following constraint:

$$\frac{(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M) \in P \qquad (\mathbf{cons}\ z_1\ z_2) \in \mathcal{E}(y)}{\mathcal{E}(z_1) \subseteq \mathcal{E}(x)} \qquad (C^P_{car})$$

  No additional constraints are required for the error case, since whenever y does not denote a pair, an `error` state is immediately produced and no further bindings are created.

- (**let** $(x$ (**apply** $y\ z))\ M)$

  Assume that $\mathcal{E}(y)$ includes $(\lambda w.\, N)$. Then the evaluation of this expression will associate the formal parameter w with one of the values of the argument variable z. Moreover, at the corresponding function return, the value of the "return" variable of the procedure may flow to the variable x. Thus, if $\mathit{FinalVar}[\,N\,]$ is a function that determines the innermost ("result") variable of N, then the crucial constraint for application expressions is as follows:

$$\frac{(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P \qquad (\lambda w.\, N) \in \mathcal{E}(y)}{\mathcal{E}(z) \subseteq \mathcal{E}(w) \qquad \mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(x)} \qquad (C^P_{apply})$$

  The definition of *FinalVar* is straightforward.

  Definition 3.3. (*FinalVar*)

$$\begin{array}{rcl} \mathit{FinalVar} : \Lambda_a & \longrightarrow & \mathit{Vars} \\ \mathit{FinalVar}[\,x\,] & = & x \\ \mathit{FinalVar}[\,(\mathbf{let}\ (x\ \cdots)\ M)\,] & = & \mathit{FinalVar}[\,M\,] \end{array}$$

  ∎

Examining each of the other classes of program terms in a similar manner results in a set of nine program-based *set constraints*: see Figure 3.

Set Constraints for a Sample Program To illustrate the derivation of set constraints, we consider the following program *cons-it*. The program defines a function *cons-it* that returns a cons-cell containing two copies of its argument, and the program then applies this function to the numbers 1 and 2, producing results $pair_1$ and $pair_2$ respectively, and finally returns the *car* of $pair_2$:

$$\frac{(\textbf{let}\ (x\ c)\ M) \in P}{c \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}c}})$$

$$\frac{(\textbf{let}\ (x\ (\lambda w.\, N))\ M) \in P}{(\lambda w.\, N) \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}lam}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{cons}\ y\ z))\ M) \in P}{(\textbf{cons}\ y\ z) \in \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}cons}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{car}\ y))\ M) \in P \qquad (\textbf{cons}\ z_1\ z_2) \in \mathcal{E}(y)}{\mathcal{E}(z_1) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{car}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{cdr}\ y))\ M) \in P \qquad (\textbf{cons}\ z_1\ z_2) \in \mathcal{E}(y)}{\mathcal{E}(z_2) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{cdr}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{if}\ y\ M_1\ M_2))\ M) \in P \qquad h \in \mathcal{E}(y) \qquad h \neq \texttt{nil}}{\mathcal{E}(\mathit{FinalVar}[\,M_1\,]) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{if\text{-}true}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{if}\ y\ M_1\ M_2))\ M) \in P \qquad \texttt{nil} \in \mathcal{E}(y)}{\mathcal{E}(\mathit{FinalVar}[\,M_2\,]) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{if\text{-}false}})$$

$$\frac{(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M) \in P \qquad (\lambda w.\, N) \in \mathcal{E}(y)}{\mathcal{E}(z) \subseteq \mathcal{E}(w) \qquad \mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{apply}})$$

$$\frac{(\textbf{let}\ (x\ N)\ M) \in P}{\mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(x)} \qquad (C^P_{\mathit{bind\text{-}v}})$$

FIGURE 3: Set Constraints on $\mathcal{E}$ with respect to P.

$$\begin{array}{l}
(\textbf{let}\ (\mathit{cons\text{-}it}\ (\lambda x.\ (\textbf{let}\ (r\ (\textbf{cons}\ x\ x))\ r)))\\
\quad (\textbf{let}\ (t_1\ 1)\\
\quad\quad (\textbf{let}\ (\mathit{pair}_1\ (\mathit{cons\text{-}it}\ t_1))\\
\quad\quad\quad (\textbf{let}\ (t_2\ 2)\\
\quad\quad\quad\quad (\textbf{let}\ (\mathit{pair}_2\ (\mathit{cons\text{-}it}\ t_2))\\
\quad\quad\quad\quad\quad (\textbf{let}\ (a\ (\textbf{car}\ \mathit{pair}_2))\\
\quad\quad\quad\quad\quad\quad a))))))
\end{array}$$

Instantiating the set constraints for this program produces the following constraints:

From $C^P_{bind\text{-}lam}$:	$(\lambda x.(\mathbf{let}\ (r\ (\mathbf{cons}\ x\ x))\ r)) \in \mathcal{E}(cons\text{-}it)$	(1)
From $C^P_{bind\text{-}cons}$:	$(\mathbf{cons}\ x\ x) \in \mathcal{E}(r)$	(2)
From $C^P_{bind\text{-}c}$:	$1 \in \mathcal{E}(t_1)$	(3)
From $C^P_{bind\text{-}c}$:	$2 \in \mathcal{E}(t_2)$	(4)
From C^P_{apply} :	$\dfrac{(\lambda w.N) \in \mathcal{E}(cons\text{-}it)}{\mathcal{E}(t_1) \subseteq \mathcal{E}(w) \qquad \mathcal{E}(FinalVar[N]) \subseteq \mathcal{E}(pair_1)}$	(5)
From C^P_{apply} :	$\dfrac{(\lambda w.N) \in \mathcal{E}(cons\text{-}it)}{\mathcal{E}(t_2) \subseteq \mathcal{E}(w) \qquad \mathcal{E}(FinalVar[N]) \subseteq \mathcal{E}(funpair_2)}$	(6)
From C^P_{car} :	$\dfrac{(\mathbf{cons}\ z_1\ z_2) \in \mathcal{E}(pair_2)}{\mathcal{E}(z_1) \subseteq \mathcal{E}(a)}$	(7)

Constraints (1) through (4) are *atomic constraints*, and we satisfy these constraints by simply adding the appropriate bindings to $\mathcal{E}$. To satisfy the *inference constraints* (5) through (7), we extend $\mathcal{E}$ as required by these constraints.

From (1), (3), (5) :	$1 \in \mathcal{E}(x)$	(8)
From (1), (2), (5) :	$(\mathbf{cons}\ x\ x) \in \mathcal{E}(pair_1)$	(9)
From (1), (4), (6) :	$2 \in \mathcal{E}(x)$	(10)
From (1), (2), (6) :	$(\mathbf{cons}\ x\ x) \in \mathcal{E}(pair_2)$	(11)
From (7), (8), (11) :	$1 \in \mathcal{E}(a)$	(12)
From (7), (10), (11) :	$2 \in \mathcal{E}(a)$	(13)

At this stage $\mathcal{E}$ is the set environment:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathcal{E}(cons\text{-}it) & = & \{(\lambda x.(\mathbf{let}\ (r\ (\mathbf{cons}\ x\ x))\ r))\} & \mathcal{E}(t_1) & = & \{1\} \\
\mathcal{E}(r) & = & \{(\mathbf{cons}\ x\ x)\} & \mathcal{E}(t_2) & = & \{2\} \\
\mathcal{E}(pair_1) & = & \{(\mathbf{cons}\ x\ x)\} & \mathcal{E}(x) & = & \{1,2\} \\
\mathcal{E}(pair_2) & = & \{(\mathbf{cons}\ x\ x)\} & \mathcal{E}(a) & = & \{1,2\}
\end{array}$$

A quick examination of the set constraints shows that this set environment satisfies the constraints for the program *cons-it*. Furthermore, since each binding in $\mathcal{E}$ is required by these constraints, $\mathcal{E}$ is the smallest set environment satisfying the set constraints. Of course, $\mathcal{E}$ does not represent the *exact* set of variable bindings created during an execution. For example, the variable a only gets bound to the constant 1 during an execution, but according to $\mathcal{E}$, the set of possible heap-values of a also includes the constant 2.

3.2 Soundness of the Set Constraints

Proving the soundness of the derived constraints requires showing that if a set environment $\mathcal{E}$ satisfies the set constraints for a given program P, then $\mathcal{E}$ must be valid for P.

Assume that $\mathcal{E}$ satisfies the set constraints relative to P. To prove that $\mathcal{E}$ is valid for P, we need to show that for any n, $\langle\, \emptyset,\, P \,\rangle \longmapsto^n \langle\, H,\, M \,\rangle$ implies that $H \models \mathcal{E}$. The natural approach is to proceed by induction on n. However, as part of the proof, we will need to consider intermediate transitions from states other than $\langle\, \emptyset,\, P\,\rangle$. Therefore, we strengthen the induction hypothesis to:

$$S \models_P \mathcal{E} \text{ and } S \longmapsto S' \text{ implies } S' \models_P \mathcal{E}$$

where the invariant relation $S \models_P \mathcal{E}$ (read "*S validates $\mathcal{E}$ with respect to P*") is an appropriately chosen relation that supports the proof of the induction hypothesis.[3] We say that the relation $S \models_P \mathcal{E}$ holds for $S = \langle\, H,\, M\,\rangle$ if and only if $H \models \mathcal{E}$ (see Definition 3.2) and $M \models_P \mathcal{E}$. The new relation $M \models_P \mathcal{E}$ (read "*M validates $\mathcal{E}$ with respect to P*") requires that M is either derived via substitution from a term in P, or that M is a **let**-expression (**let** $(x\ N)\ L$) whose sub-terms satisfy the same relation, and that the set environment $\mathcal{E}$ accounts for the potential binding of x to the values $\mathit{FinalVar}[\,N\,]$ can assume.

Definition 3.4. (*Substitution Closure of a Term, $M \models_P \mathcal{E}$, FinalVar, $S \models_P \mathcal{E}$*)

- For $M \in \Lambda_a$, the *substitution closure* of M is

$$\mathit{Cl}(M) = \{M[x_1 \leftarrow l_{x_1}, \ldots, x_n \leftarrow l_{x_n}] \mid \{x_1, \ldots, x_n\} \subseteq FV[M] \text{ and } l_{x_i} \in \mathit{Loc}_{x_i}\}$$

- $M \models_P \mathcal{E}$ holds if either:
  1. $M \in \mathit{Cl}(N)$ for some $N \in P$, or
  2. $M = (\textbf{let}\ (x\ N)\ L)$, $N \models_P \mathcal{E}$, $L \models_P \mathcal{E}$ and $\mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(x)$.

- The definition of *FinalVar* is extended to run-time terms (which may contain locations) as follows:

$$\begin{array}{rcl} \mathit{FinalVar} : \Lambda_a & \longrightarrow & \mathit{Vars} \\ \mathit{FinalVar}[\,x\,] & = & x \\ \mathit{FinalVar}[\,l_x\,] & = & x \\ \mathit{FinalVar}[\,(\textbf{let}\ (x\ \cdots)\ M)\,] & = & \mathit{FinalVar}[\,M\,] \end{array}$$

- $S \models_P \mathcal{E}$ holds if either:
  1. $S = \langle\, H,\, M\,\rangle$, $H \models \mathcal{E}$ and $M \models_P \mathcal{E}$, or
  2. $S =$ `error`.

∎

Our chosen invariant relation supports the proof of the induction hypothesis: if the invariant holds for a given state S, then the invariant also holds for the successor of S.

Lemma 3.5 *If $S \models_P \mathcal{E}$, $S \longmapsto S'$ and $\mathcal{E}$ satisfies the constraints relative to P, then $S' \models_P \mathcal{E}$.*

Proof: By case analysis of the transition rule used for $S \longmapsto S'$.

[3] The relation $S \models_P \mathcal{E}$ corresponds to the *concretization function* γ used in abstract interpretation.

- Suppose $S \longmapsto S'$ via the transition rule (*bind-lam*). Then:

$$\begin{aligned} S &= \langle\, H,\, E[\; (\mathbf{let}\ (x\ (\lambda w.\, N))\ M)\;]\,\rangle \\ S' &= \langle\, H \cup \{l_x = (\lambda w.\, N)\},\, E[\; M[x \leftarrow l_x]\;]\,\rangle \end{aligned}$$

  By Lemma 3.6(1), $(\mathbf{let}\ (x\ (\lambda w.\, N))\ M) \models_P \mathcal{E}$. Since a lambda-expression is not a term, Definition 3.4 implies that $(\mathbf{let}\ (x\ (\lambda w.\, N))\ M)$ is a substitution instance of some term $(\mathbf{let}\ (x\ (\lambda w.\, N'))\ M')$ in P, where $N \in Cl(N')$ and $M \in Cl(M')$. Constraint $(C^P_{\mathit{bind\text{-}lam}})$ implies that $(\lambda w.\, N') \in \mathcal{E}(x)$. Therefore $(\lambda w.\, N) \in Cl(\mathcal{E}(x))$ and $H \cup \{l_x = (\lambda w.\, N)\} \models \mathcal{E}$.
  From $M \in Cl(M')$ we know that $M[x \leftarrow l_x] \in Cl(M')$ and therefore $M[x \leftarrow l_x] \models_P \mathcal{E}$. Also:

$$\mathit{FinalVar}[\,(\mathbf{let}\ (x\ (\lambda w.\, N))\ M)\,] = \mathit{FinalVar}[\,M\,] = \mathit{FinalVar}[\,M[x \leftarrow l_x]\,]$$

  by Lemma 3.6(3). Hence, by Lemma 3.6(2), $E[\; M[x \leftarrow l_x]\;] \models_P \mathcal{E}$. Thus $S' \models_P \mathcal{E}$, and the claim holds for this case.

- The case where $S \longmapsto S'$ via the transition rule (*bind-c*) holds by a similar argument.

- Suppose $S \longmapsto S'$ via the transition rule (*bind-cons*). Then:

$$\begin{aligned} S &= \langle\, H,\, E[\; (\mathbf{let}\ (x\ (\mathbf{cons}\ l_y\ l_z))\ M)\;]\,\rangle \\ S' &= \langle\, H \cup \{l_x = (\mathbf{cons}\ l'_y\ l'_z)\},\, E[\; M[x \leftarrow l_x]\;]\,\rangle \end{aligned}$$

  By Lemma 3.6(1), $(\mathbf{let}\ (x\ (\mathbf{cons}\ l_y\ l_z))\ M) \models_P \mathcal{E}$. Definition 3.4 implies that $(\mathbf{let}\ (x\ (\mathbf{cons}\ l_y\ l_z))\ M)$ is a substitution instance of some term $(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M')$ in P, where $M \in Cl(M')$. Constraint $(C^P_{\mathit{bind\text{-}cons}})$ implies that $(\mathbf{cons}\ y\ z) \in \mathcal{E}(x)$. Therefore $(\mathbf{cons}\ l'_y\ l'_z) \in Cl(\mathcal{E}(x))$ and $H \cup \{l_x = (\mathbf{cons}\ l'_y\ l'_z)\} \models \mathcal{E}$.
  Also, $E[\; M[x \leftarrow l_x]\;] \models_P \mathcal{E}$ by the same reasoning as for the previous case. Hence $S' \models_P \mathcal{E}$.

- Suppose $S \longmapsto S'$ via the transition rule (*car*). Then S is the state:

$$S = \langle\, H,\, E[\; (\mathbf{let}\ (x\ (\mathbf{car}\ l_y))\ M)\;]\,\rangle$$

  If $H(l_y)$ is not a pair, then $S' = \texttt{error}$ and the claim trivially holds.
  Next, consider the case where $H(l_y)$ is the pair $(\mathbf{cons}\ l_{z_1}\ l_{z_2})$. Then:

$$S' = \langle\, H \cup \{l_x = H(l_{z_1})\},\, E[\; M[x \leftarrow l_x]\;]\,\rangle$$

  Since $H \models \mathcal{E}$, $H(l_y) \in Cl(\mathcal{E}(y))$, and thus $(\mathbf{cons}\ z_1\ z_2) \in \mathcal{E}(y)$. By Lemma 3.6(1), $(\mathbf{let}\ (x\ (\mathbf{car}\ l_y))\ M) \models_P \mathcal{E}$, and therefore Definition 3.4 implies that $(\mathbf{let}\ (x\ (\mathbf{car}\ l_y))\ M)$ is a substitution instance of some term $(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M')$ in P, where $M \in Cl(M')$. Constraint (C^P_{car}) implies that $\mathcal{E}(z_1) \subseteq \mathcal{E}(x)$, and therefore $H(l_{z_1}) \in Cl(\mathcal{E}(z_1)) \subseteq Cl(\mathcal{E}(x))$. Thus $H \cup \{l_x = H(l_{z_1})\} \models \mathcal{E}$.
  Also, $E[\; M[x \leftarrow l_x]\;] \models_P \mathcal{E}$ by the same reasoning as for the previous case. Hence $S' \models_P \mathcal{E}$.

- The case where $S \longmapsto S'$ via the transition rule (*cdr*) holds by a similar argument.
- Suppose $S \longmapsto S'$ via the transition rule (*if*). Then S is the state:

  $$\langle\, H,\, E[\,(\mathbf{let}\ (x\ (\mathbf{if}\ l_y\ M_1\ M_2))\ M)\,]\,\rangle$$

  Assume first that $H(l_y) \neq \texttt{nil}$. Then:

  $$S' = \langle\, H,\, E[\,(\mathbf{let}\ (x\ M_1)\ M)\,]\,\rangle$$

  Since $S \models_P \mathcal{E}$, there exists $h \in \mathcal{E}(y)$ with $h \neq \texttt{nil}$. By Lemma 3.6(1), $(\mathbf{let}\ (x\ (\mathbf{if}\ l_y\ M_1\ M_2))\ M) \models_P \mathcal{E}$, and therefore Definition 3.4 implies that the term $(\mathbf{let}\ (x\ (\mathbf{if}\ l_y\ M_1\ M_2))\ M)$ is a substitution instance of some term $(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1'\ M_2'))\ M')$ in P, where $M_1 \in \mathit{Cl}(M_1')$ and $M \in \mathit{Cl}(M')$. By constraint $(C^P_{\mathit{if\text{-}true}})$, $\mathcal{E}(\mathit{FinalVar}[\,M_1'\,]) \subseteq \mathcal{E}(x)$ and hence By Lemma 3.6(4), $\mathcal{E}(\mathit{FinalVar}[\,M_1\,]) \subseteq \mathcal{E}(x)$. Thus $(\mathbf{let}\ (x\ M_1)\ M) \models_P \mathcal{E}$, and $S' \models_P \mathcal{E}$.

  The case where $H(l_y) = \texttt{nil}$ holds by a similar argument.
- Suppose $S \longmapsto S'$ via the transition rule (*apply*). Then S is the state:

  $$S = \langle\, H,\, E[\,(\mathbf{let}\ (x\ (\mathbf{apply}\ l_y\ l_z))\ M)\,]\,\rangle$$

  Assume first that $H(l_y) = (\lambda w.\, N)$. Then:

  $$S' = \langle\, H \cup \{l_w = H(l_z)\},\, E[\,(\mathbf{let}\ (x\ N[w \leftarrow l_w])\ M)\,]\,\rangle$$

  Since $H \models \mathcal{E}$, $H(l_y) \in \mathit{Cl}(\mathcal{E}(y))$ and thus $(\lambda w.\, N') \in \mathcal{E}(y)$ with $N \in \mathit{Cl}(N')$. By Lemma 3.6(1), $(\mathbf{let}\ (x\ (\mathbf{apply}\ l_y\ l_z))\ M) \models_P \mathcal{E}$, which implies that there exists $(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M') \in P$ with $M \in \mathit{Cl}(M')$. By constraint (C^P_{apply}), $\mathcal{E}(z) \subseteq \mathcal{E}(w)$ and $\mathcal{E}(\mathit{FinalVar}[\,N'\,]) \subseteq \mathcal{E}(x)$. Hence $H(l_z) \in \mathit{Cl}(\mathcal{E}(w))$ and thus $H \cup \{l_w = H(l_z)\} \models \mathcal{E}$.

  Furthermore, by Lemma 3.6(4), $\mathit{FinalVar}[\,N'\,] = \mathit{FinalVar}[\,N\,]$. Hence $\mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(x)$, which implies that $(\mathbf{let}\ (x\ N[w \leftarrow l_w])\ M) \models_P \mathcal{E}$. Thus $S' \models_P \mathcal{E}$.

  If $H(l_y)$ is not a procedure, then $S' = \texttt{error}$ and the claim trivially holds.
- Suppose $S \longmapsto S'$ via the transition rule (*bind-v*). Then:

  $$\begin{aligned} S &= \langle\, H,\, E[\,(\mathbf{let}\ (x\ l_y)\ M)\,]\,\rangle \\ S' &= \langle\, H \cup \{l_x = H(l_y)\},\, E[\,M[x \leftarrow l_x]\,]\,\rangle \end{aligned}$$

  By Lemma 3.6(1), $(\mathbf{let}\ (x\ l_y)\ M) \models_P \mathcal{E}$. By Definition 3.4, either $\mathcal{E}(y) = \mathcal{E}(\mathit{FinalVar}[\,l_y\,]) \subseteq \mathcal{E}(x)$, or else $(\mathbf{let}\ (x\ y)\ M') \in P$, in which case by constraint $(C^P_{\mathit{bind\text{-}v}})$, we also have that $\mathcal{E}(y) \subseteq \mathcal{E}(x)$.

  Since $H(l_y) \in \mathit{Cl}(\mathcal{E}(y)) \subseteq \mathit{Cl}(\mathcal{E}(x))$, $H \cup \{l_x = H(l_y)\} \models \mathcal{E}$. Therefore, $S' \models_P \mathcal{E}$.

∎

The previous proof depends on the following lemma.

Lemma 3.6

1. *If $E[\ M\ ] \models_P \mathcal{E}$ then $M \models_P \mathcal{E}$.*
2. *Suppose $E[\ M\ ] \models_P \mathcal{E}$ and $\mathcal{E}$ satisfies the constraints relative to P. Then for all M' such that $M' \models_P \mathcal{E}$ and $\mathit{FinalVar}[\ M'\ ] = \mathit{FinalVar}[\ M\ ]$, we have that $E[\ M'\ ] \models_P \mathcal{E}$.*
3. *For $N \in \Lambda_a$, $\mathit{FinalVar}[\ N\ ] = \mathit{FinalVar}[\ N[x \leftarrow l_x]\ ]$.*
4. *If $N \in \mathit{Cl}(N')$, then $\mathit{FinalVar}[\ N\ ] = \mathit{FinalVar}[\ N'\ ]$.*

Proof: The first two claims can be proven by induction on the structure of E; the third follows by induction on the structure of N. Claim 4 is a consequence of claim 3. ∎

Since the invariant trivially holds for the initial state $\langle\ \emptyset,\ P\ \rangle$, it follows that the set constraints are *sound*, *i.e.*, if $\mathcal{E}$ satisfies the set constraints relative to P, then $\mathcal{E}$ is a valid set environment for P.

Theorem 3.7 (Soundness of Constraints) *If $\mathcal{E}$ satisfies the constraints relative to P, then $P \models \mathcal{E}$.*

Proof: Suppose $\mathcal{E}$ satisfies the constraints relative to P, and that

$$\langle\ \emptyset,\ P\ \rangle \longmapsto^* \langle\ H,\ M\ \rangle$$

By the definition of the invariant relation, $\langle\ \emptyset,\ P\ \rangle \models_P \mathcal{E}$. Therefore Lemma 3.5 implies that $\langle\ H,\ M\ \rangle \models_P \mathcal{E}$, which implies by definition that $H \models \mathcal{E}$. The latter is true for any state derivable from the initial state, hence $P \models \mathcal{E}$. ∎

In summary, any set environment satisfying the set constraints with respect to a program P is a conservative approximation to the set of bindings created during the execution of P.

3.3 From Set Constraints to Set-Based Analysis

The class of set environments for a given program P, denoted SetEnv_P, forms a complete lattice under the natural pointwise partial ordering $\sqsubseteq$ defined by:

$$\mathcal{E}_1 \sqsubseteq \mathcal{E}_2 \text{ if and only if } \forall x \in \mathit{Vars}_P.\ \mathcal{E}_1(x) \subseteq \mathcal{E}_2(x)$$

The least set environment $\bot_{\mathcal{E}}$ in the lattice is the trivial mapping: $\bot_{\mathcal{E}}(x) = \emptyset$. The size of the sets Vars_P and Hval_P is bounded by $|P|$, where $|P|$ denotes the number of expressions in P. Therefore, the height of the lattice SetEnv_P is bounded by $|P|^2$.

Each set constraint C_x^P (where x ranges over *bind-c*, *car*, …, *bind-v*) can be interpreted as a monotonic function, also denoted by C_x^P, on this lattice. We define the function $\mathcal{C}_P$ to be the least upper bound of the functions $C_{\mathit{bind\text{-}c}}^P, \ldots, C_{\mathit{bind\text{-}v}}^P$:

$$\begin{aligned} \mathcal{C}_P &: SetEnv_P \longrightarrow SetEnv_P \\ \mathcal{C}_P(\mathcal{E}) &= C^P_{bind\text{-}c}(\mathcal{E}) \sqcup \cdots \sqcup C^P_{bind\text{-}v}(\mathcal{E}) \end{aligned}$$

Since the combined function $\mathcal{C}_P$ is also monotonic, the least fixpoint given by $\bigsqcup_{n \in \mathbf{N}} {\mathcal{C}_P}^n(\bot_{\mathcal{E}})$ exists. Moreover, since the height of the lattice is bounded by $|P|^2$, this fixpoint is reached within $|P|^2$ steps. We define set-based analysis as the function that returns this least fixpoint.

Definition 3.8. (*sba*) The function $sba : \Lambda_a \longrightarrow SetEnv_P$ is given by:

$$sba(P) = \bigsqcup_{n \leq |P|^2} {\mathcal{C}_P}^n(\bot_{\mathcal{E}})$$

∎

Since the fixpoints of $\mathcal{C}_P$ are exactly the solutions to the set constraints, it follows that $sba(P)$ is the least solution to the set constraints, and hence that $sba(P)$ is a valid set environment for P.

Theorem 3.9 (Correctness of Set-Based Analysis) $P \models sba(P)$

Proof: Since $sba(P)$ is a solution to the set constraints, it follows by Theorem 3.7 that $P \models sba(P)$. ∎

Appendix A contains a complete algorithm that calculates the set-based analysis of a program by generating and solving set constraints for that program.

4 Set-Based Analysis of Assignments

Realistic programming languages such as Scheme or ML provide a variety of imperative facilities in addition to a functional core language. These imperative constructs typically include destructive assignments and non-local control operators. In this section, we consider how to extend set-based analysis to a language with assignments. The following section deals with first-class continuations.

Let $\Lambda_a^!$ be the extension of Λ_a with assignments:

$$M \in \Lambda_a^! ::= \ldots \mid (\mathbf{let}\ (x\ (\mathbf{set!}\ v\ v))\ M)$$

The evaluation of an assignment expression $(\mathbf{let}\ (x\ (\mathbf{set!}\ y\ z))\ M)$ *assigns* y to the value of z, and also binds x to this value, before continuing with the evaluation of M.

4.1 Semantics

We define the semantics of $\Lambda_a^!$ by extending the CS-machine with an additional transition rule for assignment expressions:

$$\begin{array}{ll} \langle\, H \cup \{l_y = h_y\},\ E[\ (\mathbf{let}\ (x\ (\mathbf{set!}\ l_y\ l_z))\ M)\ ]\,\rangle & \\ \longmapsto\ \ \langle\, H \cup \{l_y = h_z, l_x = h_z\},\ E[\ M[x \leftarrow l_x]\ ]\,\rangle & (set!) \\ \qquad \text{where } h_z = H(l_z) \text{ and } l_x = new(H \cup \{l_y = h_y\}, x) & \end{array}$$

The evaluator for the extended language is defined in the usual manner (cmp. Figure 2).

4.2 Set-Based Analysis

4.2.1 Deriving Set Constraints

The derivation of a set-based analysis for $\Lambda_a^!$ follows the development of Section 3. It yields one additional constraint, which accounts for the effect of assignment expressions. The new constraint states that for each expression of the form $(\textbf{let}\ (x\ (\textbf{set!}\ y\ z))\ M) \in P$ in the analyzed program, the values of z may flow to both x and y:

$$\frac{(\textbf{let}\ (x\ (\textbf{set!}\ y\ z))\ M) \in P}{\mathcal{E}(z) \subseteq \mathcal{E}(y) \qquad \mathcal{E}(z) \subseteq \mathcal{E}(x)} \qquad (C^P_{set!})$$

4.2.2 Soundness of the Set Constraints

We prove the soundness of the set constraints for $\Lambda_a^!$ by showing that every transition step preserves the invariant relation $S \models_P \mathcal{E}$ from Definition 3.4, which is naturally extended to $\Lambda_a^!$.

Lemma 4.1 *If $S \models_P \mathcal{E}$, $S \longmapsto S'$ and $\mathcal{E}$ satisfies the constraints relative to P, then $S' \models_P \mathcal{E}$.*

Proof:

By case analysis of the transition rule used for $S \longmapsto S'$. The only case we need to consider concerns assignments, since all the other cases are analyzed in Lemma 3.5, and can be adopted without change.

Suppose $S \longmapsto S'$ via the transition rule (*set!*). Then:

$$\begin{aligned} S &= \langle\, H \cup \{l_y = h_y\},\ E[\ (\textbf{let}\ (x\ (\textbf{set!}\ l_y\ l_z))\ M)\ ]\,\rangle \\ S' &= \langle\, H \cup \{l_y = h_z, l_x = h_z\},\ E[\ M[x \leftarrow l_x]\ ]\,\rangle \end{aligned}$$

where $h_z = H(l_z)$.

A straightforward adaptation of Lemma 3.6 to the language $\Lambda_a^!$ implies that $(\textbf{let}\ (x\ (\textbf{set!}\ l_y\ l_z))\ M) \models_P \mathcal{E}$. Therefore, this term is a substitution instance of some term $(\textbf{let}\ (x\ (\textbf{set!}\ y\ z))\ M')$ in P, where $M \in Cl(M')$. From constraint $(C^P_{set!})$, $\mathcal{E}(z) \subseteq \mathcal{E}(y)$ and $\mathcal{E}(z) \subseteq \mathcal{E}(x)$. Hence $h_z \in Cl(\mathcal{E}(y))$ and $h_z \in Cl(\mathcal{E}(x))$, which implies that $H \cup \{l_y = h_z, l_x = h_z\} \models \mathcal{E}$.

Furthermore, $M \in Cl(M')$ implies that $M[x \leftarrow l_x] \in Cl(M')$, and therefore $E[\ M[x \leftarrow l_x]\ ] \models_P \mathcal{E}$. Hence $S' \models_P \mathcal{E}$, and the claim holds for this case. ∎

The rest of the verification of set-based analysis for $\Lambda_a^!$ is adapted *mutatis mutandis*, which illustrates the flexibility of our semantic framework.

5 Set-Based Analysis of First-Class Continuations

In this section we extend our derivation of set-based analysis to a language that contains first-class continuations in addition to assignments and the functional core language[4]. Let

[4] Assignments and first-class continuations can be added in any order, or independently, to the functional core language.

$\Lambda_a^{!+c}$ be the following extension of $\Lambda_a^!$:

$$M \;\in\; \Lambda_a^{!+c} \;::=\; \ldots \mid (\mathbf{let}\ (x\ (\mathbf{letcc}\ y\ N))\ M)$$

The evaluation of a **letcc**-expression ($\mathbf{letcc}\ y\ N$) *captures* the current evaluation context (or *continuation*) surrounding that expression, binds y to a location containing this continuation, and continues with the evaluation of N. An invocation of a captured continuation causes the current evaluation context to be discarded and replaced by the captured continuation.

5.1 Semantics

We define the semantics of $\Lambda_a^{!+c}$ by appropriately extending the CS-machine. A *continuation* is a non-empty evaluation context, and we store captured continuations in the heap. Thus heaps bind locations to continuations in addition to binding locations to heap values:[5]

$$H \;\in\; \mathit{Heap} \;::=\; \{l_1 = h_1, \ldots, l_n = h_n, l_1' = E_1, \ldots, l_m' = E_m\} \quad \text{where } E_i \neq [\,]$$

We extend the CS-machine with an additional transition rule for capturing continuations, and modify the transition rule (*apply*) to correctly apply continuations as well as procedures: see Figure 4.

$$\begin{array}{ll} \langle H, E[\ (\mathbf{let}\ (x\ (\mathbf{letcc}\ y\ N))\ M)\ ]\rangle & \\ \longmapsto\ \ \langle H \cup \{l_y = E[\ (\mathbf{let}\ (x\ [\,])\ M)\ ]\},\ E[\ (\mathbf{let}\ (x\ N[y \leftarrow l_y])\ M)\ ]\rangle & (\mathit{letcc}) \\ \qquad \text{where } l_y = \mathit{new}(H, y) & \end{array}$$

$$\begin{array}{l} \langle H, E[\ (\mathbf{let}\ (x\ (\mathbf{apply}\ l_y\ l_z))\ M)\ ]\rangle \\ \longmapsto \begin{cases} \langle H \cup \{l_w = H(l_z)\},\ E[\ (\mathbf{let}\ (x\ N[w \leftarrow l_w])\ M)\ ]\rangle & \text{if } H(l_y) = (\lambda w.\, N) \\ \quad \text{where } l_w = \mathit{new}(H, w) & \\ \langle H \cup \{l_w = H(l_z)\},\ E'[\ N[w \leftarrow l_w]\ ]\rangle & \text{if } H(l_y) = E'[\ (\mathbf{let}\ (w\ [\,])\ N)\ ] \\ \quad \text{where } l_w = \mathit{new}(H, w) & \\ \texttt{error} & \text{otherwise} \end{cases} \quad (\mathit{apply}) \end{array}$$

FIGURE 4: Additional Transition Rules for First-Class Continuations

The evaluator for the extended language is defined in the usual manner (cmp. Figure 2). If the evaluation of a program returns a captured continuation, then the extended evaluator simply returns the tag `procedure`.

5.2 Set-Based Analysis

The analysis of the language $\Lambda_a^{!+c}$ requires an extended notion of set environments, since a heap now maps locations to both heap values and continuations. As before, we use elements of Hval_P to represent heap values, and we need to choose a finite set of *abstract continuations*

[5] For technical reasons that will become clear soon, we do not include continuations in the set of heap values.

to represent the continuations that may be captured during an execution. For set-based analysis, the crucial property of a continuation $E[\ (\textbf{let}\ (y\ [\ ])\ M)\ ]$ is that the variable y receives the argument value passed to the continuation. Therefore, we use the *abstract continuation* $(\textbf{cont}\ y)$ to represent all continuations of the form $E[\ (\textbf{let}\ (y\ [\ ])\ M)\ ]$. Let $Cont_P$ denote the set $\{(\textbf{cont}\ y) \mid y \in Vars_P\}$ of abstract continuations used for the analysis of the program P. A set environment is now a finite table that maps each program variable to a subset of $Hval_P \cup Cont_P$. A set environment $\mathcal{E}$ is *valid* for P if for each heap value h associated with a variable x during an execution, $h \in Cl(\mathcal{E}(x))$, and for each continuation $E[\ (\textbf{let}\ (y\ [\ ])\ M)\ ]$ associated with x during an execution, $(\textbf{cont}\ y) \in \mathcal{E}(x)$.

Definition 5.1. ($Cont_P$*, Set environments, Validity,* $H \models \mathcal{E}$) Let P be a program.

- $Cont_P = \{(\textbf{cont}\ y) \mid y \in Vars_P\}$.
- A mapping $\mathcal{E} : Vars_P \to \mathcal{P}_{\mathrm{fin}}(Hval_P \cup Cont_P)$ is a *set environment* for P.
- A set environment $\mathcal{E}$ is *valid* for P if $\langle\, \emptyset,\ P\,\rangle \longmapsto^* \langle\, H,\ M\,\rangle$ implies that $H \models \mathcal{E}$.
- The relation $H \models \mathcal{E}$ (read "H *validates* $\mathcal{E}$") holds if and only if:
  - for all bindings $(l_x = h) \in H$, $h \in Cl(\mathcal{E}(x))$, and
  - for all bindings $(l_x = E[\ (\textbf{let}\ (y\ [\ ])\ M)\ ]) \in H$, $(\textbf{cont}\ y) \in \mathcal{E}(x)$.

∎

5.2.1 Deriving Set Constraints

The set constraints for $\Lambda_a^{!+c}$ extend those for $\Lambda_a^{!}$. The additional constraints ensure that the set environment $\mathcal{E}$ accounts for the bindings created during the capture and invocation of continuations.

- $(\textbf{let}\ (y\ (\textbf{letcc}\ x\ N))\ M)$

  The evaluation of this expression during an execution captures some run-time evaluation context $E[\ (\textbf{let}\ (y\ [\ ])\ M)\ ]$ surrounding the **letcc**-expression, and binds x to this continuation. In addition, if the evaluation of N terminates, the value of the "return" variable of N may flow to the variable y. Thus, the crucial constraint for **letcc** expressions is as follows:

$$\frac{(\textbf{let}\ (y\ (\textbf{letcc}\ x\ N))\ M) \in P}{(\textbf{cont}\ y) \in \mathcal{E}(x) \qquad \mathcal{E}(\mathit{FinalVar}[\,N\,]) \subseteq \mathcal{E}(y)} \qquad (C^P_{letcc})$$

- $(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M)$

  Suppose that $\mathcal{E}(y)$ includes $(\textbf{cont}\ w)$. Then, the variable y may denote a continuation of the form $E[\ (\textbf{let}\ (w\ [\ ])\ N)\ ]$ during an execution, and the evaluation of this application expression may pass the value of the argument variable z to the variable w. To ensure that the set environment $\mathcal{E}$ accounts for bindings created by such transitions, we require that $\mathcal{E}$ satisfies the following constraint:

$$\frac{(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M) \in P \qquad (\textbf{cont}\ w) \in \mathcal{E}(y)}{\mathcal{E}(z) \subseteq \mathcal{E}(w)} \qquad (C^P_{apply\text{-}k})$$

Because continuations never return, no constraints concerning $\mathcal{E}$ are required for the return case.

5.2.2 Soundness of the Set Constraints

Proving the soundness of the set constraints for the extended language $\Lambda_a^{!+c}$ requires showing that if a set environment $\mathcal{E}$ satisfies the set constraints for a given program P, then $\mathcal{E}$ must be valid for P. As before, the proof proceeds based on an induction hypothesis:

$$S \models_P \mathcal{E} \text{ and } S \longmapsto S' \text{ implies } S' \models_P \mathcal{E}$$

where $S \models_P \mathcal{E}$ is an appropriately chosen invariant relation. For the analysis of the language $\Lambda_a^{!+c}$, we need to strengthen the invariant relation previously used for both Λ_a and $\Lambda_a^{!}$. The strengthened invariant relation ensures that only "proper" continuations are stored in the heap.

Definition 5.2. ($H \models_P \mathcal{E}$, $E \models_P \mathcal{E}$, $S \models_P \mathcal{E}$)

- $H \models_P \mathcal{E}$ holds if $H \models \mathcal{E}$ and for all $(l_x = E) \in H$, $E \models_P \mathcal{E}$.
- $E \models_P \mathcal{E}$ holds for $E = (\textbf{let}\ (x\ E')\ L)$ if $L \models_P \mathcal{E}$ and either:
  - $E' = [\ ]$, or
  - $E' \models_P \mathcal{E}$ and $\mathcal{E}(\mathit{FinalVar}[\,E'\,]) \subseteq \mathcal{E}(x)$.
- $S \models_P \mathcal{E}$ holds if either:
  1. $S = \langle\, H, M\,\rangle$, $H \models_P \mathcal{E}$ and $M \models_P \mathcal{E}$, or
  2. $S =$ `error`.

∎

Our chosen invariant relation supports the proof of the induction hypothesis: if the invariant holds for a given state S, then the invariant also holds for the successor of S.

Lemma 5.3 *If $S \models_P \mathcal{E}$, $S \longmapsto S'$ and $\mathcal{E}$ satisfies the constraints relative to P, then $S' \models_P \mathcal{E}$.*

Proof: By case analysis of the rule used for $S \longmapsto S'$. The only cases we need to consider concern continuation capture and invocation, since the other cases hold by the arguments used in Lemma 3.5 or Lemma 4.1, and can be adopted without change.

- Suppose $S \longmapsto S'$ via the transition rule (*letcc*). Then:

$$\begin{array}{rcl} S & = & \langle\, H,\, E[\, (\textbf{let}\ (x\ (\textbf{letcc}\ y\ N))\ M)\,]\,\rangle \\ S' & = & \langle\, H',\, E[\, (\textbf{let}\ (x\ N[y \leftarrow l_y])\ M)\,]\,\rangle \end{array}$$

  where $H' = H \cup \{l_y = E[\, (\textbf{let}\ (x\ [\,])\ M)\,]\}$.
  Lemma 3.6(1) implies that $(\textbf{let}\ (x\ (\textbf{letcc}\ y\ N))\ M) \models_P \mathcal{E}$. Therefore, the term $(\textbf{let}\ (x\ (\textbf{letcc}\ y\ N))\ M)$ is a substitution instance of some term $(\textbf{let}\ (x\ (\textbf{letcc}\ y\ N'))\ M')$ in P, where $N \in Cl(N')$ and $M \in Cl(M')$. By the constraint (C^P_{letcc}), $(\textbf{cont}\ x) \in \mathcal{E}(y)$, which implies that $H' \models \mathcal{E}$. By Lemma 5.4, $E[\, (\textbf{let}\ (x\ [\,])\ M)\,] \models_P \mathcal{E}$, and hence $H' \models_P \mathcal{E}$.
  The constraint (C^P_{letcc}) also implies that $\mathcal{E}(\mathit{FinalVar}[\,N'\,]) \subseteq \mathcal{E}(x)$, and therefore by Lemma 3.6, parts 3 and 4, $\mathcal{E}(\mathit{FinalVar}[\,N[y \leftarrow l_y]\,]) \subseteq \mathcal{E}(x)$. Hence $(\textbf{let}\ (x\ N[y \leftarrow l_y])\ M) \models_P \mathcal{E}$, $S' \models_P \mathcal{E}$, and the claim holds for this case.

- Suppose $S \longmapsto S'$ via the transition rule (*apply*). Then S is the state:

$$\langle\, H,\, E[\, (\textbf{let}\ (x\ (\textbf{apply}\ l_y\ l_z))\ M)\,]\,\rangle$$

  Assume that $H(l_y) = E'[\, (\textbf{let}\ (w\ [\,])\ N)\,]$. Then:

$$S' = \langle\, H \cup \{l_w = H(l_z)\},\, E'[\, N[w \leftarrow l_w]\,]\,\rangle$$

  Since $H \models \mathcal{E}$, we know that $(\textbf{cont}\ w) \in \mathcal{E}(y)$. Lemma 3.6(1) implies that $(\textbf{let}\ (x\ (\textbf{apply}\ l_y\ l_z))\ M) \models_P \mathcal{E}$, and hence $(\textbf{let}\ (x\ (\textbf{apply}\ l_y\ l_z))\ M)$ is a substitution instance of some term $(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M') \in P$, where $M \in Cl(M')$. By constraint $(C^P_{apply\text{-}k})$, $\mathcal{E}(z) \subseteq \mathcal{E}(w)$. Thus $H(l_z) \in Cl(\mathcal{E}(w))$ and $H \cup \{l_w = H(l_z)\} \models \mathcal{E}$.
  Furthermore, since $E'[\, (\textbf{let}\ (w\ [\,])\ N)\,] \models_P \mathcal{E}$, by Lemma 5.4 we have that $E'[\, N[w \leftarrow l_w]\,] \models_P \mathcal{E}$. Hence $S' \models_P \mathcal{E}$, and the claim holds for this case.

∎

The previous proof depends on the following lemma, and on a straightforward extension of Lemma 3.6 to $\Lambda_a^{!+c}$.

Lemma 5.4

1. *If* $E[\, (\textbf{let}\ (x\ (\textbf{letcc}\ y\ N))\ M)\,] \models_P \mathcal{E}$, *then* $E[\, (\textbf{let}\ (x\ [\,])\ M)\,] \models_P \mathcal{E}$.
2. *If* $E'[\, (\textbf{let}\ (w\ [\,])\ N)\,] \models_P \mathcal{E}$, *then* $E'[\, N[w \leftarrow l_w]\,] \models_P \mathcal{E}$.

   Proof: The proof of both parts proceeds by induction on the structure of E. ∎

Theorem 5.5 (Soundness of Constraints for $\Lambda_a^{!+c}$)
For $P \in \Lambda_a^{!+c}$, *if* $\mathcal{E}$ *satisfies the constraints relative to* P, *then* $P \models \mathcal{E}$.

Proof: Follows from Lemma 5.3, based on the argument used in Theorem 3.7. ∎

The rest of the verification of set-based analysis is again straightforward.

6 Application: Soft Typing from Set-Based Analysis

The information produced by set-based analysis facilitates a variety of optimizations. Examples include partial evaluation [15], escape analysis [18], redundant test elimination [11], closure analysis [21], touch-optimization [8, 7], dead-code elimination, constant-folding and code hoisting. Here, we illustrate how set-based analysis applies to soft typing [4, 28, 1, 12, 14].

6.1 Soft Typing

The programming language Scheme [2] is a dynamically-typed language, which imposes no restrictions on the values that variables or expressions in a program may assume. Dynamic-typing leads to a simple and flexible language suitable for rapid software development, but does not provide any compile-time guarantee that primitive operations are only applied to arguments of the appropriate type. Instead, each primitive operation typically performs a *type-check* on its arguments at run-time to ensure that they are of the appropriate type. Thus, for example, the operation **car** must first check that its argument is a pair, before it can extract the first component of that pair. Performing these type-checks can add a significant overhead to the execution time. For example, Wright [25:p. 105] reports that this cost is 22% of the execution time under Chez Scheme at `optimize-level 2`. Since Chez Scheme eliminates some run-time checks at `optimize-level 2`, a naïve compiler would spend an even greater fraction of the execution time performing these type-checks.

One of the goals[6] of soft-typing is to reduce the overhead of dynamic typing by removing unnecessary type-checks wherever provably possible, without changing the semantics of programs. In this section, we develop a soft-typing algorithm that exploits the information produced by set based analysis for this purpose.

6.2 Non-Checking Primitives

The current language does not provide primitives that do not perform type-checks on their arguments. To express and verify a soft-typing algorithm that replaces the type-checking primitives **car**, **cdr** and **apply** by non-checking primitives, we extend the language $\Lambda_a^{!+c}$ with non-checking forms of the type-checking primitive operations, denoted **car**, **cdr** and **apply**, respectively:

$$\begin{array}{lcl} M & ::= & \ldots \\ & & \mid (\textbf{let}\ (x\ (\underline{\textbf{car}}\ y))\ M) \\ & & \mid (\textbf{let}\ (x\ (\underline{\textbf{cdr}}\ y))\ M) \\ & & \mid (\textbf{let}\ (x\ (\underline{\textbf{apply}}\ y\ z))\ M) \end{array}$$

As their name indicates, a non-checking operation behaves in the same manner as the original version as long as its arguments are of the appropriate type.[7] If the argument is not of the appropriate type, the behavior of the non-checking variant is unspecified, and any arbitrary state may be produced. The extended language is called $\underline{\Lambda_a^{!+c}}$.

We define the semantics of the extended language $\underline{\Lambda_a^{!+c}}$ by extending the CS-machine with the additional transition rules described in Figure 5. The evaluator for the extended

[6] Unlike other approaches to soft-typing [4, 28, 1], we are not concerned with inferring types for program

$$\langle H, E[\ (\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ l_y))\ M)\ ]\rangle$$
$$\longmapsto \begin{cases} \langle H \cup \{l_x = H(l_{z_1})\}, E[\ M[x \leftarrow l_x]\ ]\rangle & \text{if } H(l_y) = (\mathbf{cons}\ l_{z_1}\ l_{z_2}) \\ \quad \text{where } l_x = new(H, x) & \\ \text{unspecified} & \text{if } H(l_y) \neq (\mathbf{cons}\ l_{z_1}\ l_{z_2}) \end{cases} \qquad (\underline{car})$$

$$\langle H, E[\ (\mathbf{let}\ (x\ (\underline{\mathbf{cdr}}\ l_y))\ M)\ ]\rangle$$
$$\longmapsto \text{analogous to } (\underline{car}) \qquad (\underline{cdr})$$

$$\langle H, E[\ (\mathbf{let}\ (x\ (\underline{\mathbf{apply}}\ l_y\ l_z))\ M)\ ]\rangle$$
$$\longmapsto \begin{cases} \langle H \cup \{l_w = H(l_z)\}, E[\ (\mathbf{let}\ (x\ N[w \leftarrow l_w])\ M)\ ]\rangle & \text{if } H(l_y) = (\lambda w. N) \\ \quad \text{where } l_w = new(H, w) & \\ \text{unspecified} & \text{if } H(l_y) \neq (\lambda w. N) \end{cases} \qquad (\underline{apply})$$

FIGURE 5: Transition Rules for Non-Checking Primitives

language, $\underline{eval}$, is defined in the usual way (cmp. Figures 2). Unlike $eval$, the evaluator $\underline{eval}$ is no longer a total function. There are programs in $\underline{\Lambda_a^{!+c}}$ for which the behavior of the evaluator $\underline{eval}$ is unspecified because of the application of a non-checking operation to an argument of inappropriate type. Still, the two evaluators agree on programs in $\Lambda_a^{!+c}$.

Lemma 6.1 *For $P \in \Lambda_a^{!+c}$, $\underline{eval}(P) = eval(P)$.*

6.3 The Soft Typing Algorithm

Our soft typing algorithm uses the results of set based analysis to replace the type-checking operations **car**, **cdr** and **apply** by the corresponding non-checking operation whenever possible, *without* changing the semantics of programs. For example, suppose that a program contains (**let** (x (**car** y)) M) and that set-based analysis proves that y is only bound to pairs. Then we can replace the expression with (**let** (x ($\underline{\mathbf{car}}$ y)) M), which the machine can execute without performing a type-check on y. In general, if the set environment produced by set-based analysis shows that the arguments of **car**, **cdr** or **apply** are always of the correct type, the soft-typing algorithm replaces the operation with its non-checking version. The soft-typing algorithm $\mathcal{T}$, parameterized over a set environment $\mathcal{E}$, is defined in Figure 6. If $\mathcal{E}$ is valid for the program being optimized, then the soft-typing algorithm $\mathcal{T}_\mathcal{E}$ preserves the meaning of that program. For every transition step of the source program P there exists a corresponding transition step for the soft-typed program $\mathcal{T}_\mathcal{E}(P)$. We extend $\mathcal{T}_\mathcal{E}$ in a compatible manner to states to aid in the proof of this correspondence.

Lemma 6.2 (Step Correspondence) *Let P be a program with a set environment $\mathcal{E}$, and let S be a state for which $\mathcal{E}$ is valid.*

1. *Suppose $S \longmapsto S'$. Then $\mathcal{T}_\mathcal{E}(S) \longmapsto \mathcal{T}_\mathcal{E}(S')$.*

variables and presenting these types to the programmer.

[7] We only consider procedures and not continuations to be an appropriate first argument to $\underline{\mathbf{apply}}$ in order to avoid the need for a type-dispatch on this argument within $\underline{\mathbf{apply}}$.

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{E}} : \Lambda_a^{!+c} & \longrightarrow & \underline{\Lambda_a^{!+c}} \\
\mathcal{T}_{\mathcal{E}}[x] & = & x \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ c)\ M)] & = & (\textbf{let}\ (x\ c)\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\lambda y.\,N))\ M)] & = & (\textbf{let}\ (x\ (\lambda y.\,\mathcal{T}_{\mathcal{E}}(N)))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{cons}\ y\ z))\ M)] & = & (\textbf{let}\ (x\ (\textbf{cons}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{car}\ y))\ M)] & = & \left\{ \begin{array}{ll} (\textbf{let}\ (x\ (\underline{\textbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq \{(\textbf{cons}\ z_1\ z_2)\} \\ (\textbf{let}\ (x\ (\textbf{car}\ y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq \{(\textbf{cons}\ z_1\ z_2)\} \end{array} \right. \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{cdr}\ y))\ M)] & = & \text{analogous to } \textbf{car} \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{if}\ y\ M_1\ M_2))\ M)] & = & (\textbf{let}\ (x\ (\textbf{if}\ y\ \mathcal{T}_{\mathcal{E}}[M_1]\ \mathcal{T}_{\mathcal{E}}[M_2]))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M)] & = & \left\{ \begin{array}{ll} (\textbf{let}\ (x\ (\underline{\textbf{apply}}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq \{(\lambda w.\,N)\} \\ (\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq \{(\lambda w.\,N)\} \end{array} \right. \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ N)\ M)] & = & (\textbf{let}\ (x\ \mathcal{T}_{\mathcal{E}}[N])\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{set!}\ y\ z))\ M)] & = & (\textbf{let}\ (x\ (\textbf{set!}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let}\ (x\ (\textbf{letcc}\ y\ N))\ M)] & = & (\textbf{let}\ (x\ (\textbf{letcc}\ y\ \mathcal{T}_{\mathcal{E}}[N]))\ \mathcal{T}_{\mathcal{E}}[M])
\end{array}
$$

FIGURE 6: The soft-Typing Algorithm $\mathcal{T}$

2. *Suppose* $\mathcal{T}_{\mathcal{E}}(S) \longmapsto S''$. *Then there exists* S' *such that* $S \longmapsto S'$ *and* $\mathcal{T}_{\mathcal{E}}(S') = S''$.

   Proof:

   The proof of both parts proceeds by induction on n and by case analysis of $S \longmapsto S'$ and $\mathcal{T}_{\mathcal{E}}(S) \longmapsto S''$ respectively. ∎

The Step Correspondence Lemma implies that a soft-typed program exhibits the same behavior as the corresponding source program.

Theorem 6.3 *For any program* $P \in \Lambda_a^{!+c}$, $\underline{eval}(\mathcal{T}_{sba(P)}(P)) = eval(P)$.

Proof: Both the fact that $\underline{eval}$ is well-defined on $\mathcal{T}_{sba(P)}(P)$ and that the equality holds follow from Lemma 6.2. ∎

In summary, the soft-typing algorithm removes redundant type-checks from programs based on the information produced by set-based analysis. This optimization algorithm is provably-correct with respect to the extended evaluator $\underline{eval}$. Any implementation that realizes $\underline{eval}$ correctly can therefore make use of this optimization.

7 Related Work

The main contribution of our paper is an elegant and flexible *derivation* of set-based analysis. It substantially simplifies previous formulations [10, 11, 8, 7], and explains the underlying notion of approximation in a clear manner. Unlike the derivation and formalization of other approaches to program analysis, our derivation is not restricted to purely functional languages. Instead, it extends in a natural and straightforward manner to imperative features such as destructive assignments (Section 4) and non-local control operators (Section 5). This extensibility is essential for analyzing programs in languages with higher-order functions as well as imperative features. In this section, we compare our *derivation* with a number of comparable program analyzes, including an advanced type inference algorithm [1] and two recent variants [23, 13] of Shivers' 0CFA [22]

Aiken *et al.* [1] develop a type inference algorithm for a purely functional language. They reduce the type inference problem to a system of type inclusion constraints, which are similar to our set constraints. They do not present a proof of their type soundness lemma, but indicate that this proof is based on the denotational semantics of their language. It is well-known that extending type soundness proofs based on denotational semantics to a language with imperative features is complex and error-prone, even for a simple Hindley-Milner type system: see Tofte's [24] discussion of Damas's [3] faulty proof of a type soundness theorem. Extending the derivation and soundness proofs of Aiken *et al.*'s rich type system would be even more difficult and complex.

Stefanescu and Zhou [23] present an analysis for an intermediate closure-converted language. They start from an operational semantics in which each state is a set of bindings from locations to either values or expressions. A state transition arbitrarily chooses a location-expression binding and reduces that expression. Their analysis is parameterized by a function that maps each location created during an execution into a finite set of abstract locations. They derive a system of equations, similar to our set constraints, that relates the sets of abstract values that these abstract locations may assume. The result of their analysis is the least solution to the system of equations.

Their semantics is non-intuitive primarily because it uses bindings for two distinct purposes. States contain both location-value bindings, which correspond to our heap, and location-expression bindings, which essentially represents a flat version of our A-normal form control string. This flat representation does not maintain the control information provided by our control string; instead, their semantics evaluates available redexes in an indeterminate manner. This indeterminate behavior is a major obstacle to extending their approach to a language with any imperative features. The addition of destructive assignments is further complicated by the requirement of their indeterminate semantics that all transitions can only increase the set of bindings in a state, and cannot modify existing bindings.

Jagannathan and Weeks [13] develop an analysis for a higher-order language, starting from an unusual operational semantics that records a *flow graph* in which nodes represent location-value bindings and edges represent the flow of values between locations in a program. Like Stefanescu and Zhou, they parameterize their analysis over relations that describe which locations should be merged. The result of their analysis is the least fixpoint of an abstract transition function.

The complexity of their approach is primarily a result of their decision to analyze the source language, instead of a simplified intermediate form. Expressions in their language can be nested in arbitrarily complex fashions. Evaluating these expressions in a leftmost-outermost manner requires a complex strategy to locate the next redex in the program. Furthermore, the evaluation of nested expressions produces intermediate values not associated with program variables. To record these intermediate values, Jagannathan and Weeks *label* each program expression with a unique label, and then associate each intermediate value with the label of the corresponding expression. Thus, their semantics and their analysis record label-value associations in addition to the conventional variable-value associations. Our choice of an intermediate A-normal form representation avoids both of these complexities. Because the order of execution in our language follows the lexical nesting of **let**-expressions [9], control transfer is accomplished in a simple manner. Also, since every intermediate value in our language is named, we only need to record variable-value associations, and avoid the additional complexity of labeling each expression in the program [20].

Jagannathan and Weeks further complicate their rewriting semantics using labels of expressions within their state representation instead of the actual expressions themselves. Therefore, a state does not completely represent an intermediate stage of a computation and cannot be further evaluated without knowledge of the original source program. In addition, their state representation does not contain any explicit control information. Instead, they use the subtle trick of extracting control information from the edges of their flow-graph. Specifically, when the body of a function terminates, they retrieve the label of the expression to which the result of the function body "flows", and then transfer control to that expression. It is not obvious how this technique extends to complex control operators such as exceptions or **call/cc**.

A final disadvantage of Jagannathan and Weeks derivation is that their analysis is defined via a complex abstract transition function. This monolithic definition does not provide the insight and understanding that the specification of set-based analysis in terms of set constraints provides.

Wright and Felleisen [26] describe a simple and flexible strategy for proving the type soundness of Hindley-Milner-style polymorphic type systems. Their approach is based on a reduction semantics that easily extends to encompass imperative constructs, and adapts subject reduction theorems from combinatory logic to programming languages. Our derivation of set-based analysis exploits similar techniques. Specifically, our derivation is also based on a reduction semantics, and Lemma 3.5 can be viewed as a *subject reduction lemma*: it states that the reduction of a state preserves correctness with respect to any set environment satisfying the set constraint. Like Wright and Felleisen's approach, our derivation extends to additional imperative facilities in a straightforward manner. We expect that this flexibility is a result of the subject reduction technique common to both.

8 Conclusion

Static information about program behavior is crucial for the generation of efficient code for advanced languages. Unfortunately, the derivation of an analysis to compute such information for higher-order languages is subtle and complex, and is closely tied to the formulation of the semantics of the language. Many analyses that were originally developed for a purely functional language can be difficult if not impossible to extend to languages with additional imperative features. This combination of complexity and inflexibility is a major obstacle to exploiting these analyses in realistic compilers.

Our derivation of set-based analysis avoids these complexities and limitations. We formulate our derivation for the language of A-normal forms, since this language constitutes an ideal intermediate representation for compilers [9]. The simplicity of our derivation follows both from our use of the A-normal form language, and from our text-based reduction semantics, which explicitly models the allocation of objects into a global heap, and which records information that associates each run-time value with the corresponding syntactic value. Our use of A-normal forms simplifies our reduction semantics, since the order of execution follows the lexical nesting of **let**-expressions. In addition, A-normal forms are ideally suited to the derivation of analyses, since all intermediate values in the language are named, and there is no need for the additional complexity of labeling each expression in the program. We derive our analysis directly from the reduction semantics, thus avoiding the intermediate steps of Heintze's derivation, and we exploit the information recorded by our semantics to express our set constraints on a finite lattice of set environments, instead of on the infinite lattice required by Heintze's approach.

Extending our derivation to accommodate additional language facilities is straightforward. We have seen how the derivation extends to languages with destructive assignments and first class continuations, thus providing the first soundness proof of set-based analysis for these imperative features. Additional features such as exceptions or communication channels are also easily incorporated. The primary reason for the extensibility of our derivation is the flexibility of our semantics. Since the semantics allocates all values into a global heap, it easily extends to include assignments or cycle-creating constructs such as **letrec**. The semantics also exposes the entire state of the evaluation at each intermediate step, which facilitates the inclusion of non-local control operators such as first-class continuations or exceptions.

Our derivation can also be modified to produce a more accurate (but more expensive) analysis. One possible approach is to apply Heintze's technique of *polyvariance* [11], which is equivalent to creating different "versions" of certain functions in the source program by preceding the analysis by a series of β_v "expansion" steps.

A more general approach requires modifying our semantics to expose additional details. Specifically, it involves extracting the *environment* and *continuation* components of the control string, and expressing them as separate components of each intermediate state. The resulting semantics is essentially a CESK-machine [6, 5] specialized to A-normal forms. Based on this semantics we can follow our derivation to develop a number of different constraint-based analyses, with different cost-performance tradeoffs.[8] It is also possible to develop a general analysis framework that is parameterized in a fashion similar to Jagannathan and Weeks's framework [13]. A particular instance of this framework is created by specifying relations that describe which locations created during program execution should be merged. Instances of this framework include set-based analysis, as well as a number of more precise analyses.

A Set-Based Analysis Algorithm

A complete set-based analysis algorithm for a Scheme-like language containing both assignments and first-class continuations is included in Figures 7 through 9. The algorithm is written in Scheme extended with a special form **match** for pattern matching [27].

The function *SBA* traverses an expression to derive the set constraints for that expression. Certain set constraints cannot be satisfied immediately. Consider, for example, the expression (**let** (x (**set!** y z)) M). The constraint $C^P_{set!}$ requires that all values in $\mathcal{E}(z)$

[8] See [8, 7] for a derivation of set-based analysis from a parallel CEK-machine.

```
;; inputs:   a term M
;; outputs:  calculates FinalVar[ M ]
;; effects:  derives the set constraints for M,
;;           and invokes a constraint solver on these constraints

(define SBA
  (lambda (M)
    (match M
      [(? variable? x) x]
      [`(let (,x ,exp) ,body)
       (begin
         (match exp
           [(? constant? c) (new-value! x c)]
           [`(cons ,y1 ,y2) (new-value! x `(cons ,y1 ,y2))]
           [`(lambda ,y ,N)
            (let ([finalvar-N (SBA N)])
              (new-value! x `(lambda ,y ,N ,finalvar-N)))]
           [`(car ,y) (new-constraint! y `(propagate-car-to ,x))]
           [`(cdr ,y) (new-constraint! y `(propagate-cdr-to ,x))]
           [`(if ,y ,M1 ,M2)
            (let ([finalvar-M1 (SBA M1)][finalvar-M2 (SBA M2)])
              (new-constraint! y `(conditional-prop #t ,finalvar-M1 ,x))
              (new-constraint! y `(conditional-prop #f ,finalvar-M2 ,x)))]
           [`(apply ,y ,z) (new-constraint! y `(application ,x ,z))]
           [`(set! ,y ,z)
            (new-constraint! z `(propogate-to ,y))
            (new-constraint! z `(propogate-to ,x))]
           [`(letcc ,y ,N)
            (let ([finalvar-N (SBA N)])
              (new-constraint! finalvar-N `(propogate-to ,x))
              (new-value! y `(cont ,x)))]
           [_  ; Must be a term
            (let ([finalvar-exp (SBA exp)])
              (new-constraint! finalvar-exp `(propagate-to ,x)))])
         (SBA body))])))
```

FIGURE 7: The Set-Based Analysis Algorithm

must be in $\mathcal{E}(x)$, but the set $\mathcal{E}(z)$ may not yet be known. Therefore, the function *SBA* associates a constraint `(propagate-to ,x) with the variable z. This constraint is applied (via the function *interpret-constraint*) to each value that is added to $\mathcal{E}(z)$, and propagates that value to $\mathcal{E}(x)$.

The functions *new-value!* and *new-constraint!* associate values and constraints, respectively, with variables. The function *interpret-constraint* ensures that each constraint on a variable is satisfied for all possible values of that variable, by creating additional values or new constraints as necessary.

The worst-case time complexity of the algorithm is $O(|P|^3)$, which can be verified as

```
;; inputs:   a variable and a value
;; outputs:  void
;; effects:  adds the value to ℰ(var), and applies the appropriate constraints to that value
(define new-value!
  (lambda (var value)
    (unless (in-ℰ? var value)
       (add-to-ℰ! var value)
       (foreach-in-𝒞 var (lambda (constraint) (interpret-constraint constraint value))))))

;; inputs:   a variable and a constraint
;; outputs:  void
;; effects:  associates the constraint with var, and applies the constraint to all values in ℰ(var)
(define new-constraint!
  (lambda (var constraint)
    (unless (in-𝒞? var constraint)
       (add-to-𝒞! var constraint)
       (foreach-in-ℰ var (lambda (value) (interpret-constraint constraint value))))))

;; inputs:   a constraint and a value
;; outputs:  void
;; effects:  ensures that the constraint is satisfied, given that the constraint
;;           and the value are associated with the same variable
(define interpret-constraint
  (lambda (constraint value)
    (match (cons constraint value)
       [`((propagate-to ,x) . ,v) (new-value! x v)]
       [`((propagate-car-to ,x) . (cons ,y1 ,y2)) (new-constraint! y1 `(propagate-to ,x))]
       [`((propagate-cdr-to ,x) . (cons ,y1 ,y2)) (new-constraint! y2 `(propagate-to ,x))]
       [`((application ,result ,arg) . (lambda ,para ,_ ,finalvar))
        (new-constraint! finalvar `(propagate-to ,result))
        (new-constraint! arg `(propagate-to ,para))]
       [`((application ,result ,arg) . (cont ,x)) (new-constraint! arg `(propagate-to ,x))]
       [`((conditional-prop ,test ,from ,to) . ,value)
        (when (eq? (null? value) test) (new-constraint! from `(propagate-to ,to)))]
       [_ (void)])))
```

FIGURE 8: The Set-Based Analysis Algorithm: Auxiliary Functions 1

follows: We assume that the functions *in-ℰ?*, *add-to-ℰ!*, *in-𝒞?* and *add-to-𝒞!* operate in constant time, and that the functions *foreach-in-ℰ* and *foreach-in-𝒞* operate in time linear in the number of elements in the appropriate set.

Each call of the functions *new-value!* and *new-constraint!* takes constant time, excluding the time spent in the body of the respective **unless** expressions. The number of variables, the number of values, and the number of constraints are all $O(|P|)$. Therefore the test conditions of each **unless** expression can succeed at most $O(|P|^2)$ times, and function *interpret-constraint* is called at most $O(|P|^3)$ times. Each call of *interpret-constraint* terminates in constant time. Therefore, the entire algorithm takes $O(|P|^3)$ time.

;; The following three functions maintain a mapping $\mathcal{E}$ from program
;; variables to sets of values

;; inputs: a variable and a value
;; outputs: returns true if $\mathcal{E}(var)$ contains the value
;; effects: none
(**define** *in-$\mathcal{E}$?* (**lambda** (*var value*) ...))

;; inputs: a variable and a value
;; outputs: void
;; effects: adds *value* to $\mathcal{E}(var)$
(**define** *add-to-$\mathcal{E}$!* (**lambda** (*var value*) ...))

;; inputs: a variable and a function
;; outputs: none
;; effects: applies *fn* to each value in $\mathcal{E}(var)$
(**define** *foreach-in-$\mathcal{E}$* (**lambda** (*var fn*) ...))

;; The following three functions maintain a mapping $\mathcal{C}$ from program
;; variables to sets of constraints

;; inputs: a variable and a constraint
;; outputs: returns true if $\mathcal{C}(var)$ contains the constraint
;; effects: none
(**define** *in-$\mathcal{C}$?* (**lambda** (*var constraint*) ...))

;; inputs: a variable and a constraint
;; outputs: void
;; effects: adds *constraint* to $\mathcal{C}(var)$
(**define** *add-to-$\mathcal{C}$!* (**lambda** (*var constraint*) ...))

;; inputs: a variable and a function
;; outputs: none
;; effects: applies *fn* to each constraint in $\mathcal{E}(var)$
(**define** *foreach-in-$\mathcal{C}$* (**lambda** (*var fn*) ...))

FIGURE 9: The Set-Based Analysis Algorithm: Auxiliary Functions 2

Acknowledgements We thank Nevin Neintze for discussions on set-based analysis and for access to his implementation of set-based analysis for ML. We also thank Chris Colby and Stephanie Weirich for comments on an earlier version of this paper.

References

[1] AIKEN, A., WIMMERS, E. L., AND LAKSHMAN, T. K. Soft typing with conditional types. In *Proceedings of the ACM Sigplan Conference on Principles of Programming Languages* (1994), pp. 163–173.

[2] Clinger, W., and Jonathan Rees, e. The revised[4] report on the algorithmic language scheme. *ACM Lisp Pointers 4*, 3 (July 1991).

[3] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[4] Fagan, M. *Soft Typing.* PhD thesis, Rice University, 1990.

[5] Felleisen, M., and Friedman, D. P. A calculus for assignments in higher-order languages. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages* (Munich, West Germany, Jan. 1987), pp. 314–345.

[6] Felleisen, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[7] Flanagan, C., and Felleisen, M. Well-founded touch optimization for futures. Rice University Computer Science TR94-239.

[8] Flanagan, C., and Felleisen, M. The semantics of future and its use in program optimizations. In *Proceedings of the ACM Sigplan Conference on Principles of Programming Languages* (1995), pp. 209–220.

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

[10] Heintze, N. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[11] Heintze, N. Set-based analysis of ML programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (1994), pp. 306–317.

[12] Henglein, F. Dynamic typing: syntax and proof theory. *Science of Computer Programming 22* (1994), 197–230.

[13] Jagannathan, S., and Weeks, S. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages* (1995), pp. 393–407.

[14] Jagannathan, S., and Wright, A. Effective flow analysis for avoiding run-time checks. , 1995.

[15] Malmkjaer, K., Heintze, N., and Danvy, O. ML partial evaluation using set-based analysis. Tech. Rep. CMU-CS-94-129, Carnegie Mellon University, 1994.

[16] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, 1990.

[17] Morrisett, G., Felleisen, M., and Harper, R. Abstract models of memory management. In *Functional Programming and Computer Architecture* (1995). To appear.

[18] Park, Y. G., and Goldberg, B. Escape analysis on lists. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (June 1992), pp. 116–127.

[19] Rozas, G. J. Liar, an Algol-like compiler for scheme. Master's thesis, Massachusetts Institute of Technology, January 1984.

[20] Sabry, A., and Felleisen, M. Is continuation-passing useful for data flow analysis. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1994), pp. 1–12.

[21] Shao, Z., and Appel, A. Space-efficient closure representations. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1994), pp. 150–161.

[22] Shivers, O. *Control-flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie-Mellon University, 1991.

[23] Strfanescu, D., and Zhou, Y. An equational framework for the flow analysis of higher order functional programs. In *LFP* (1994), pp. 318–327.

[24] Tofte, M. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1987.

[25] Wright, A. *Practical Soft Typing for Scheme.* PhD thesis, Rice University, 1994.

[26] Wright, A., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991.

[27] Wright, A. K., and Duba, B. F. Pattern matching for Scheme. Unpublished manuscript, 1993. Available at `"ftp://cs.rice.edu/public/wright/match.ps.Z"`.

[28] Wright, A. and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (1994), pp. 250–262.