Well-Founded Touch Optimization for Futures

Cormac Flanagan
Matthias Felleisen

Rice COMP TR94-239

October 1994

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

Well-Founded Touch Optimization for Futures

Cormac Flanagan* Matthias Felleisen*

Department of Computer Science,
Rice University,
Houston, TX 77251-1892

Abstract

The **future** annotations of MultiLisp provide a simple method for taming the implicit parallelism of functional programs, but require *touch* operations within all placeholder-strict primitives to ensure proper synchronization between threads. These *touch* operations contribute substantially to program execution times. We use an operational semantics of **future** developed in a previous paper to derive a program analysis algorithm and an optimization algorithm based on the analysis that removes provably-redundant *touch* operations.Experiments with the Gambit compiler indicate that this optimization substantially reduces program execution times.

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

Contents

1 Futures, Touches and Transparency

Programs in functional languages offer numerous opportunities for executing program components in parallel. In a call-by-value language, for example, the evaluation of every function application could spawn a parallel thread for each argument expression. However, if such a strategy were applied indiscriminately, the execution of a program would generate far too many parallel threads. The overhead of managing these threads would clearly outweigh any benefits from parallel execution.

The **future** annotations of MultiLisp and its Scheme successors [1, 9] provide a simple method for taming the implicit parallelism of functional programs. If a programmer believes that the parallel evaluation of some expression outweighs the overhead of creating a separate task, he may annotate the expression with the keyword **future**. An annotated functional program has the same observable behavior as the original program, but the run-time system may choose to evaluate the **future** expression in parallel to the rest of the program. If it does, the evaluation will proceed as if the annotated expression had immediately returned. Instead of a proper value though, it returns a placeholder, which contains enough information for retrieving the actual result of the annotated expression when needed. When a program operation requires specific knowledge about the value of some subcomputation but finds a placeholder, the run-time system performs a *touch* operation, which synchronizes the appropriate parallel threads, and eventually retrieves the necessary information.

The standard way of implementing *touch* operations on placeholders requires a modification of all program operations that need to know specific aspects of values. For example, procedure application must know that the value of the first sub-expression is a procedure; an **if**-expression demands a proper value in the test position, not only a placeholder; and, addition can only add its inputs if they are numbers. Hence, these operations must be modified so that they first check whether the appropriate arguments are placeholders or proper values and must possibly perform some synchronization.

Past research on **future**s has concentrated on the efficient implementation of the underlying task creation [5, 14, 20, 21, 22] and on the extension of the concept to higher-order control constructs [16, 23]. Little effort has gone into the development of a semantic characterization of the idea or the use of such a semantic framework for the optimization of task creation or coordination. In contrast, the driving force behind our effort is the desire to develop semantically well-founded optimizations for the execution of **future**s. The specific example we choose to consider is the development of an algorithm that safely eliminates as many *touch* operations as possible. Other optimizations will be the subject of future efforts.

In a previous paper [7], we developed a series of semantics for an idealized functional language with **future**. The last semantics is particularly suited to the develelopment of analysis and optimization algorithms, since it exposes appropriate details regarding program executions. We now use that semantics to derive a program analysis algorithm, and a provably-correct *touch* optimization algorithm based on the analysis. An implementation of the optimization on the Gambit Scheme compiler [5] produced significant speedups on a standard set of benchmarks. We believe that this development can be extended to larger languages and other implementation techniques.

The presentation of our results proceeds as follows. The second section introduces a simple, functional language with **future**s, and recalls the low-level parallel abstract machine for the language [7]. The third section discusses the cost of *touch* operations and presents a provably correct algorithm for eliminating unnecessary *touch* operations. The latter is based on the set-based analysis algorithm of the fourth section. The fifth section describes the implementation of these algorithms on the Gambit compiler and compares the modified compiler to the original compiler on a standard set of benchmarks. The sixth section discusses related work.

2 Review: The Language and Its Parallel Semantics

Given the goal of developing a semantics that is useful for proving the soundness of optimizations, we develop the definitional semantics for **future**s for an intermediate representation of an idealized functional language. Specifically, we use the subset of A-normal forms [8] of an extended λ-calculus-like language that includes conditionals and a **future** construct: see Figure 1.

$$
\begin{array}{lclcll}
M & \in & \Lambda_a & ::= & x & \text{(Terms)} \\
 & & & & \mid (\mathbf{let}\ (x\ V)\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{future}\ M))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{if}\ y\ M\ M))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) & \\
V & \in & \mathit{Value} & ::= & c \mid x \mid (\lambda x.\, M) \mid (\mathbf{cons}\ x\ y) & \text{(Values)} \\
x & \in & \mathit{Vars} & = & \{x, y, z, \ldots\} & \text{(Variables)} \\
c & \in & \mathit{Const} & = & \{\mathtt{true}, \mathtt{false}, 0, 1, \ldots\} & \text{(Constants)}
\end{array}
$$

FIGURE **1** The A-normalized Language Λ_a

In a previous paper [7], we developed a series of abstract machines, each specifying the semantics of **future**s at a different level of abstraction. The last of these machines, called the $P(CEK)$-machine, is particularly suited to the development of program analyses, since contains explicit binding information relating program variables to their values. The machine is also suited to the development of a *touch* optimization algorithm, since it exposes the use of explicit placeholder objects and performs *touch* operations on these objects. We use this machine, defined in Figures 2 and 3, as the basis for the development of this paper.

Notation We use the following notations throughout the paper: $\mathcal{P}$ denotes the power-set constructor; $f : A \longrightarrow B$ denotes that f is a total function from A to B; $f : A \longrightarrow_p B$ denotes that f is a partial function from A to B; $M \in P$ denotes that the term M occurs in the program P; and $\langle \mathbf{ar}?\ x, M, E\rangle$ refers to either a normal activation record $\langle \mathbf{ar}\ x, M, E\rangle$ or a tagged activation record $\langle \mathbf{ar}\dagger\ x, M, E\rangle$.

3 Touch Optimization

The $P(CEK)$-machine performs *touch* operations on arguments in placeholder-strict positions of all program operations. These implicit *touch* operations guarantee the transparency of placeholders, which makes **future**-based parallelism so convenient to use. Unfortunately, these compiler-inserted *touch* operations impose a significant overhead on the execution of annotated programs. For example, an annotated doubly-recursive version of *fib* performs 1.3 million *touch* operations during the computation of (*fib* 25).

first mention of ph-strict

Due to the dynamic typing of Scheme, the cost of each *touch* operation depends on the program operation that invoked it. If a program operation already performs a type dispatch to ensure that

Evaluator:

$$eval_{pcek} : \Lambda_a^0 \longrightarrow Answers \cup \{\texttt{error}, \bot\}$$

$$eval_{pcek}(P) = \begin{cases} unload_{pcek}[E(x)] & \text{if } \langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} \langle x, E, \epsilon\rangle \\ \texttt{error} & \text{if } \langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} \texttt{error} \\ \bot & \text{if } \forall i \in N\ \exists S_i \in State_{pcek}, n_i, m_i \in N \text{ such that} \\ & \quad m_i > 0,\ S_0 = \langle P, \emptyset, \epsilon\rangle \text{ and } S_i \longmapsto^{n_i, m_i}_{pcek} S_{i+1} \end{cases}$$

Data Specifications:

$$\begin{array}{lllll} S & \in & State_{pcek} & ::= & \langle M, E, K\rangle \mid \texttt{error}(\textbf{f-let}\ (p\ S)\ S) & \text{(States)} \\ M & \in & \Lambda_a & & & (A\text{-nf Language}) \\ E & \in & Env_{pcek} & ::= & Vars \longrightarrow_p Value_{pcek} & \text{(Environments)} \\ V & \in & Value_{pcek} & ::= & PValue_{pcek} \mid Ph\text{-}Obj_{pcek} & \text{(Run-Time Values)} \\ W & \in & PValue_{pcek} & ::= & c \mid x \mid Cl_{pcek} \mid Pair_{pcek} & \text{(Proper Values)} \\ & & Cl_{pcek} & ::= & \langle(\lambda x.\, M), E\rangle & \text{(Closures)} \\ & & Pair_{pcek} & ::= & (\textbf{cons}\ V\ V) & \text{(Pairs)} \\ & & Ph\text{-}Obj_{pcek} & ::= & \langle \textbf{ph}\ p\ \circ\rangle \mid \langle \textbf{ph}\ p\ V\rangle & \text{(Placeholder Values)} \\ K & \in & Cont_{pcek} & ::= & \epsilon \mid \langle \textbf{ar}\ x, M, E\rangle.K \mid \langle \textbf{ar}\dagger\ x, M, E\rangle.K & \text{(Continuations)} \\ F & \in & FinalState_{pcek} & ::= & \langle x, E, \epsilon\rangle \mid \texttt{error} & \text{(Final States)} \\ A & \in & Answers & ::= & c \mid \texttt{procedure} \mid (\textbf{cons}\ A\ A) & \text{(Answers)} \end{array}$$

Auxiliary Functions:

$$\begin{array}{rcl} unload_{pcek} : Value^0_{pcek} & \longrightarrow & Answers \\ unload_{pcek}[c] & = & c \\ unload_{pcek}[\langle(\lambda x.\, M), E\rangle] & = & \texttt{procedure} \\ unload_{pcek}[(\textbf{cons}\ V_1\ V_2)] & = & (\textbf{cons}\ V_1'\ V_2') \\ & & \text{where } V_i' = unload_{pcek}[V_i] \\ unload_{pcek}[\langle \textbf{ph}\ p\ V\rangle] & = & unload_{pcek}[V] \end{array} \qquad \begin{array}{rcl} touch_{pcek} : Value_{pcek} & \longrightarrow & PValue_{pcek} \cup \{\circ\} \\ touch_{pcek}[\langle \textbf{ph}\ p\ \circ\rangle] & = & \circ \\ touch_{pcek}[\langle \textbf{ph}\ p\ V\rangle] & = & touch_{pcek}[V] \\ touch_{pcek}[W] & = & W \end{array}$$

Placeholder Substitution $S[p := V]$:

$$M[p := V] = M \text{ with all occurrences of } \langle \textbf{ph}\ p\ \circ\rangle \text{ updated to } \langle \textbf{ph}\ p\ V\rangle$$

$$(\textbf{f-let}\ (p'\ S_1)\ S_2)[p := V] = \begin{cases} (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2) & \text{if } p = p' \\ (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2[p := V]) & \text{if } p \neq p' \end{cases}$$

FIGURE **2** The $P(CEK)$-machine: Evaluator and Data Specifications

its arguments have the appropriate type, *e.g.*, **car**, **cdr**, **apply**, *etc*, then a *touch* operation is free. Put differently, an implementation of (**car** x) in pseudo-code is:

(**if** (*pair?* x) (**unchecked-car** x)
(*error* 'car "Not a pair"))

Extending the semantics of **car** to perform a *touch* operation on placeholders is simple:

(**if** (*pair?* x) (**unchecked-car** x)
(**let** ([y (*touch* x)]) (**if** (*pair?* y) (**unchecked-car** y) (*error* 'car "Not a pair"))))

The *touch*ing version of **car** incurs an additional overhead only in the error case or when x is a placeholder. For the interesting case when x is a pair, no overhead is incurred. Since the vast

Transition Rules:

$$
\begin{array}{rcll}
\langle(\mathbf{let}\ (x\ c)\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow c], K\rangle & (\mathit{bind\text{-}const})\\
\langle(\mathbf{let}\ (x\ y)\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow E(y)], K\rangle & (\mathit{bind\text{-}var})\\
\langle(\mathbf{let}\ (x\ (\lambda y.\, N))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow \langle(\lambda y.\, N), E\rangle], K\rangle & (\mathit{bind\text{-}lam})\\
\langle(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow (\mathbf{cons}\ E(y)\ E(z))], K\rangle & (\mathit{bind\text{-}cons})\\
\langle x, E, \langle \mathbf{ar}\ y, M, E'\rangle.K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E'[y \leftarrow E(x)], K\rangle & (\mathit{return})
\end{array}
$$

$$
\langle(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M), E, K\rangle \longmapsto^{1,1}_{pcek}
\begin{cases} \langle M, E[x \leftarrow V_1], K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = (\mathbf{cons}\ V_1\ V_2)\\ \mathtt{error} & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \mathit{Pair}_{pcek} \cup \{\circ\} \end{cases} \quad (\mathit{car})
$$

$$
\langle(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \text{analogous to } \mathit{car} \quad (\mathit{cdr})
$$

$$
\langle(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M), E, K\rangle \longmapsto^{1,1}_{pcek}
\begin{cases} \langle M_1, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \{\mathtt{false}, \circ\}\\ \langle M_2, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = \mathtt{false} \end{cases} \quad (\mathit{if})
$$

$$
\langle(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M), E, K\rangle \longmapsto^{1,1}_{pcek}
\begin{cases} \langle N, E'[x' \leftarrow E(z)], \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = \langle(\lambda x'.\, N), E'\rangle\\ \mathtt{error} & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \mathit{Cl}_{pcek} \cup \{\circ\} \end{cases} \quad (\mathit{apply})
$$

$$
\begin{array}{rcll}
\langle(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle N, E, \langle \mathbf{ar}\dagger\ x, M, E\rangle.K\rangle & (\mathit{future})\\
\langle x, E, \langle \mathbf{ar}\dagger\ y, M, E'\rangle.K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E'[y \leftarrow E(x)], K\rangle & (\mathit{future\text{-}id})
\end{array}
$$

$$
\begin{array}{l}
\langle M, E, K_1.\langle \mathbf{ar}\dagger\ x, N, E'\rangle.K_2\rangle \longmapsto^{1,0}_{pcek}\\
\quad (\mathbf{f\text{-}let}\ (p\ \langle M, E, K_1\rangle)\ \langle N, E'[x \leftarrow \langle \mathbf{ph}\ p\ \circ\rangle], K_2\rangle) \quad p \notin FP(E') \cup FP(K_2) \qquad (\mathit{fork})
\end{array}
$$

$$
\begin{array}{rcll}
(\mathbf{f\text{-}let}\ (p\ \langle x, E, \epsilon\rangle)\ S) & \longmapsto^{1,1}_{pcek} & S[p := E(x)] & (\mathit{join})\\
(\mathbf{f\text{-}let}\ (p\ \mathtt{error})\ S) & \longmapsto^{1,1}_{pcek} & \mathtt{error} & (\mathit{join\text{-}error})\\
(\mathbf{f\text{-}let}\ (p_2\ (\mathbf{f\text{-}let}\ (p_1\ S_1)\ S_2))\ S_2) & \longmapsto^{1,1}_{pcek} & (\mathbf{f\text{-}let}\ (p_1\ S_1)\ (\mathbf{f\text{-}let}\ (p_2\ S_2)\ S_3)) & (\mathit{lift})\\
 & & p_1 \notin FP(S_3) & \\
(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2) & \longmapsto^{a+c,b}_{pcek} & (\mathbf{f\text{-}let}\ (p\ S_1')\ S_2') & (\mathit{parallel})\\
 & & \text{if } S_1 \longmapsto^{a,b}_{pcek} S_1' \longmapsto^{c,d}_{pcek} S_2' & \\
S & \longmapsto^{0,0}_{pcek} & S & (\mathit{reflexive})\\
S & \longmapsto^{a+c,b+d}_{pcek} & S'' & (\mathit{transitive})\\
 & & \text{if } S \longmapsto^{a,b}_{pcek} S',\ S' \longmapsto^{c,d}_{pcek} S'',\ a, c > 0 &
\end{array}
$$

FIGURE **3** The $P(CEK)$-machine: Transition Rules majority of Scheme operations already perform a type-dispatch on their arguments,[1] the overhead of performing implicit *touch* operations appears to be acceptable at first glance.

Unfortunately, a standard technique for increasing execution speed in Scheme systems is to disable type-checking typically based on informal correctness arguments or based on type verifiers for the underlying sequential language [26]. When type-checking is disabled, most program operations do *not* perform a type-dispatch on their arguments. Under these circumstances, the source code (**car** x) translates to the pseudo-code:

$$(\textbf{unchecked-car}\ x)$$

Extending the semantics of **car** to perform a *touch* operation on placeholders is now quite expensive, since it then performs an additional check on every invocation:

$$(\textbf{if}\ (\textit{placeholder?}\ x)\ (\textbf{unchecked-car}\ (\textit{touch}\ x))\ (\textbf{unchecked-car}\ x))$$

Performing these *placeholder?* checks can add a significant overhead to the execution time. Kranz [19] and Feeley [5] estimated this cost at nearly 100% of the (sequential) execution time, and our experiments confirm these results (see below).

The classical solution for avoiding this overhead is to provide a compiler switch that disables the automatic insertion of *touch*es, and a *touch* primitive so that programmers can insert *touch* operations *explicitly* where needed [5, 17, 25]. We believe that this solution is flawed for several reasons. First, it clearly destroys the transparent character of **future** annotations. Instead of an annotation that only affects executions on some machines, **future** is now a task creation construct and *touch* is a synchronization tool. Second, to use this solution safely, the programmer must know where placeholders can appear instead of regular values and must add *touch* operations at these places in the program. In contrast to the addition of **future** annotations, the placement of *touch* operations is far more difficult: while the former requires a prediction concerning computational intensity, the latter demands a full understanding of the data flow properties of the program. Since we believe that an accurate prediction of data flow by the programmer is only possible for small programs, we reject this traditional solution.

A better approach than explicit *touch*es is for the compiler to use information provided by a data-flow analysis of the program to remove unnecessary *touch*es wherever possible. This approach substantially reduces the overhead of *touch* operations without sacrificing the simplicity or transparency of **future** annotations.

3.1 Non-touching Primitives

The current language does not provide primitives that do not *touch* arguments in placeholder-strict positions. To express and verify an algorithm that replaces *touch*ing primitives by non-*touch*ing primitives, we extend the language Λ_a with non-touching forms of the placeholder-strict primitive operations, denoted **<u>car</u>**, **<u>cdr</u>**, **<u>if</u>** and **<u>apply</u>**, respectively:

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

[1] Two notable exceptions are **if**, which does not perform a type-dispatch on the value of the test expression, and the equality predicate *eq?*, which is typically implemented as a pointer comparison.

As their name indicates, a non-touching operation behaves in the same manner as the original version as long as its argument in the placeholder-strict position is not a placeholder. If the argument is a place-holder, the behavior of the non-*touch*ing variant is undefined. The extended language is called $\underline{\Lambda_a}$.

We define the semantics of the extended language $\underline{\Lambda_a}$ by extending the $P(CEK)$-machine with the additional transition rules described in Figure 4. The evaluator for the extended language, $\underline{eval_{pcek}}$, is defined in the usual way (cmp. Figures 2 and 3). Unlike $eval_{pcek}$, the evaluator $\underline{eval_{pcek}}$ is no longer a function. There are programs in $\underline{\Lambda_a}$ for which the evaluator $\underline{eval_{pcek}}$ can either return a value or can be unspecified because of the application of a non-*touch*ing operation to a placeholder. Still, the two evaluators clearly agree on programs in Λ_a.

Lemma 3.1 *For* $P \in \Lambda_a$, $\underline{eval_{pcek}}(P) = eval_{pcek}(P)$.

3.2 The Touch Optimization Algorithm

The goal of *touch* optimization is to replace the touching operations **car**, **cdr**, **if** and **apply** by the corresponding non-touching operation whenever possible, *without* changing the semantics of programs. For example, suppose that a program contains (**let** $(x$ (**car** $y))$ M) and we can prove that y is never bound to a placeholder. Then we can replace the expression by the form (**let** $(x$ ($\underline{\textbf{car}}$ $y))$ M), which the machine can execute more efficiently without performing a test for placeholdership on y.

This optimization technique relies on a detailed data-flow analysis of the program that determines a conservative approximation to the set of run-time values for each variable. More specifically, we assume that the analysis returns a *valid set environment*, which is a table mapping program variables to a set of run-time values[2] that subsumes the set of values associated with that variable during an execution.

Definition 3.2. (*Set environments,* $E \models \mathcal{E}$, $S \models \mathcal{E}$, $P \models \mathcal{E}$) Let P be a program and let $Vars_P$ be the set of variables occurring in P.

- A mapping $\mathcal{E} : Vars_P \to \mathcal{P}(Value_{pcek})$ is a *set environment* for P.
- $E \models \mathcal{E}$ (read *E validates $\mathcal{E}$*) if for all $x \in dom(E)$, $E(x) \in \mathcal{E}(x)$.
- The relation $S \models \mathcal{E}$ (read *S validates $\mathcal{E}$*) is defined inductively:
  - $\langle M, E, K\rangle \models \mathcal{E}$ if $E \models \mathcal{E}$.
  - (**f-let** $(p\ S_1)\ S_2) \models \mathcal{E}$ if $S_1 \models \mathcal{E}$ and $S_2 \models \mathcal{E}$.
- $P \models \mathcal{E}$ (read *P validates $\mathcal{E}$*) if for every state S such that $\langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} S$, $S \models \mathcal{E}$.

∎

The basic idea behind *touch* optimization is now easy to explain. If a valid set environment shows that the argument of a *touch*ing version of **car**, **cdr**, **if** or **apply** can never be a placeholder, the optimization algorithm replaces the operation with its non-*touch*ing version. The optimization algorithm $\mathcal{T}$, parameterized over a valid set environment $\mathcal{E}$, is defined in Figure 5.

The function *sba*, described in the next section, always returns a valid set environment for a program. Assuming the correctness of of set-based analysis, the touch optimization algorithm preserves the meaning of programs. For every transition step of a source program P there exists a corresponding transition step for the optimized program $\mathcal{T}(P)$.

[2] Or a least a representation of this set that provides the appropriate information.

$$\langle(\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ y))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \begin{cases} \langle M, E[x \leftarrow V_1], K\rangle & \text{if } E(y) = (\mathbf{cons}\ V_1\ V_2) \\ \text{unspecified} & \text{if } E(y) \in \mathit{Ph\text{-}Obj}_{pcek} \\ \texttt{error} & \text{otherwise} \end{cases} \quad (\underline{car})$$

$$\langle(\mathbf{let}\ (x\ (\underline{\mathbf{cdr}}\ y))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \text{analogous to } \underline{\mathbf{car}} \quad (\underline{cdr})$$

$$\langle(\mathbf{let}\ (x\ (\underline{\mathbf{if}}\ y\ M_1\ M_2))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \begin{cases} \langle M_2, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } E(y) = \texttt{false} \\ \text{unspecified} & \text{if } E(y) \in \mathit{Ph\text{-}Obj}_{pcek} \\ \langle M_1, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{otherwise} \end{cases} \quad (\underline{if})$$

$$\langle(\mathbf{let}\ (x\ (\underline{\mathbf{apply}}\ y\ z))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \begin{cases} \langle N, E'[x' \leftarrow E(z)], \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } E(y) = \langle(\lambda x'.\, N), E'\rangle \\ \text{unspecified} & \text{if } E(y) \in \mathit{Ph\text{-}Obj}_{pcek} \\ \texttt{error} & \text{otherwise} \end{cases} \quad (\underline{apply})$$

FIGURE 4 Non-*touch*ing transition rules

$$\begin{aligned}
\mathcal{T}_{\mathcal{E}} &: \Lambda_a \longrightarrow \underline{\Lambda_a} \\
\mathcal{T}_{\mathcal{E}}[x] &= x \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ c)\ M)] &= (\mathbf{let}\ (x\ c)\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ y)\ M)] &= (\mathbf{let}\ (x\ y)\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\lambda y.\, N))\ M)] &= (\mathbf{let}\ (x\ (\lambda y.\, \mathcal{T}_{\mathcal{E}}(N)))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M)] &= (\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)] &= (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{T}_{\mathcal{E}}[N]))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M)] &= \begin{cases} (\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq \mathit{PValue}_{pcek} \\ (\mathbf{let}\ (x\ (\mathbf{car}\ y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq \mathit{PValue}_{pcek} \end{cases} \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M)] &= \text{analogous to } \mathbf{car} \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M)] &= \begin{cases} (\mathbf{let}\ (x\ (\underline{\mathbf{if}}\ y\ M_1\ M_2))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq \mathit{PValue}_{pcek} \\ (\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq \mathit{PValue}_{pcek} \end{cases} \\
\mathcal{T}_{\mathcal{E}}[(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M)] &= \begin{cases} (\mathbf{let}\ (x\ (\underline{\mathbf{apply}}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq \mathit{PValue}_{pcek} \\ (\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq \mathit{PValue}_{pcek} \end{cases}
\end{aligned}$$

FIGURE 5 The touch optimization algorithm $\mathcal{T}$

Lemma 3.3 (Step Correspondence) *Let P be a program with $\mathcal{E} = sba(P)$, and let S be a state for which $\mathcal{E}$ is valid.* could use any valid $\mathcal{E}$

1. *Suppose $S \longmapsto^{n,m}_{pcek} S'$. Then $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{n,m}_{pcek} \mathcal{T}_{\mathcal{E}}(S')$.*

2. *Suppose $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{n,m}_{pcek} S''$. Then $S \longmapsto^{n,m}_{pcek} S'$ where $\mathcal{T}_{\mathcal{E}}(S') = S''$.*

Proof: We prove the first part by induction on n and by case analysis of $S \longmapsto^{n,m}_{pcek} S'$.

- We first consider the case where $S \longmapsto^{1,1}_{pcek} S'$ via the transition rule *car*. Then $S = \langle (\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M), E, K\rangle$. There are four sub-cases to consider, depending on whether or not $\mathcal{T}_{\mathcal{E}}(y)$ contains placeholders and whether or not $E(y)$ is a pair.
  - Suppose if $\mathcal{E}(y) \not\subseteq PValue_{pcek}$ and $touch_{pcek}[E(y)] = (\mathbf{cons}\ V_1\ V_2)$. Then:

$$\begin{aligned} S' &= \langle M, E[x \leftarrow V_1], K\rangle \\ \mathcal{T}_{\mathcal{E}}(S) &= \langle (\mathbf{let}\ (x\ (\mathbf{car}\ y))\ \mathcal{T}_{\mathcal{E}}(M)), \mathcal{T}_{\mathcal{E}}(E), \mathcal{T}_{\mathcal{E}}(K)\rangle \\ touch(\mathcal{T}_{\mathcal{E}}(E)(y)) &= (\mathbf{cons}\ \mathcal{T}_{\mathcal{E}}(V_1)\ \mathcal{T}_{\mathcal{E}}(V_2)) \\ \mathcal{T}_{\mathcal{E}}(S') &= \langle \mathcal{T}_{\mathcal{E}}(M), \mathcal{T}_{\mathcal{E}}(E[x' \leftarrow V_1]), \mathcal{T}_{\mathcal{E}}(K)\rangle \end{aligned}$$

    Hence $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{1,1}_{pcek} \mathcal{T}_{\mathcal{E}}(S')$ via the rule *car*.
  - Suppose if $\mathcal{E}(y) \not\subseteq PValue_{pcek}$ and $touch_{pcek}[E(y)] \notin Pair_{pcek} \cup \{\circ\}$. Then:

$$\begin{aligned} S' &= \langle \mathtt{error}, \emptyset, \epsilon\rangle \\ \mathcal{T}_{\mathcal{E}}(S) &= \langle (\mathbf{let}\ (x\ (\mathbf{car}\ y))\ \mathcal{T}_{\mathcal{E}}(M)), \mathcal{T}_{\mathcal{E}}(E), \mathcal{T}_{\mathcal{E}}(K)\rangle \\ touch_{pcek}[\mathcal{T}_{\mathcal{E}}(E)(y)] &\notin Pair_{pcek} \cup \{\circ\} \\ \mathcal{T}_{\mathcal{E}}(S') &= \langle \mathtt{error}, \emptyset, \epsilon\rangle \end{aligned}$$

    Hence $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{1,1}_{pcek} \mathcal{T}_{\mathcal{E}}(S')$ via the rule *car*.
  - Suppose if $\mathcal{E}(y) \subseteq PValue_{pcek}$ and $touch_{pcek}[E(y)] = (\mathbf{cons}\ V_1\ V_2)$. Then

$$\begin{aligned} S' &= \langle M, E[x \leftarrow V_1], K\rangle \\ \mathcal{T}_{\mathcal{E}}(S) &= \langle (\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}(M)), \mathcal{T}_{\mathcal{E}}(E), \mathcal{T}_{\mathcal{E}}(K)\rangle \\ E(y) &= (\mathbf{cons}\ V_1\ V_2) \\ \mathcal{T}_{\mathcal{E}}(E)(y) &= (\mathbf{cons}\ \mathcal{T}_{\mathcal{E}}(V_1)\ \mathcal{T}_{\mathcal{E}}(V_2)) \\ \mathcal{T}_{\mathcal{E}}(S') &= \langle \mathcal{T}_{\mathcal{E}}(M), \mathcal{T}_{\mathcal{E}}(E[x' \leftarrow V_1]), \mathcal{T}_{\mathcal{E}}(K)\rangle \end{aligned}$$

    Hence if $\mathcal{E}(y) \subseteq PValue_{pcek}$ via the rule $\underline{car}$.
  - Finally, suppose $\mathcal{E}(y) \subseteq PValue_{pcek}$ and $touch_{pcek}[E(y)] \notin Pair_{pcek} \cup \{\circ\}$. Then

$$\begin{aligned} E(y)) &\notin Pair_{pcek} \cup \{\circ\} \\ S' &= \langle \mathtt{error}, \emptyset, \epsilon\rangle \\ \mathcal{T}_{\mathcal{E}}(S) &= \langle (\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}(M)), \mathcal{T}_{\mathcal{E}}(E), \mathcal{T}_{\mathcal{E}}(K)\rangle \\ \mathcal{T}_{\mathcal{E}}(E)(y) &\notin Pair_{pcek} \cup \{\circ\} \\ \mathcal{T}_{\mathcal{E}}(S') &= \langle \mathtt{error}, \emptyset, \epsilon\rangle \end{aligned}$$

    Hence $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{1,1}_{pcek} \mathcal{T}_{\mathcal{E}}(S')$ via the rule $\underline{car}$.

  Hence if $S \longmapsto^{1,1}_{pcek} S'$ via the rule *car*, then $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{1,1}_{pcek} \mathcal{T}_{\mathcal{E}}(S')$ via one of the rules *car* or $\underline{car}$.
- The other cases are similar.

The second part of the proof proceeds in a similar manner by induction on n and by case analysis of $\mathcal{T}_{\mathcal{E}}(S) \longmapsto^{n,m}_{pcek} S''$.

- Consider the case where $\mathcal{T}_{\mathcal{E}}(S) \longmapsto_{pcek} S''$ via the transition rule $\underline{car}$.

$$\begin{aligned} \mathcal{T}_{\mathcal{E}}(S) &= \langle (\textbf{let}\ (x\ (\underline{\textbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}(M)), \mathcal{T}_{\mathcal{E}}(E), \mathcal{T}_{\mathcal{E}}(K) \rangle \\ S &= \langle (\textbf{let}\ (x\ (\textbf{car}\ y))\ M), E, K \rangle \quad \text{by the definition of } \mathcal{T}_{\mathcal{E}} \\ \mathcal{E}(y) &\subseteq PValue_{pcek} \end{aligned}$$

  - Suppose that $\mathcal{T}_{\mathcal{E}}(E)(y) = (\textbf{cons}\ \mathcal{T}_{\mathcal{E}}(V_1)\ \mathcal{T}_{\mathcal{E}}(V_2))$. Then

$$\begin{aligned} S'' &= \langle \mathcal{T}_{\mathcal{E}}(M), \mathcal{T}_{\mathcal{E}}(E)[x \leftarrow \mathcal{T}_{\mathcal{E}}(V_1)], \mathcal{T}_{\mathcal{E}}(K) \rangle \\ E(y) &= (\textbf{cons}\ V_1\ V_2) \\ S' &= \langle M, E[x \leftarrow V_1], K \rangle \end{aligned}$$

    Hence $S \longmapsto^{1,1}_{pcek} S'$ via the rule *car*, and $\mathcal{T}_{\mathcal{E}}(S') = S''$.
  - Alternatively suppose that $\mathcal{T}_{\mathcal{E}}(E)(y) \notin Pair_{pcek}$. Then

$$\begin{aligned} touch_{pcek}[E(y)] &\notin Pair_{pcek} \cup \{\circ\} \\ S'' &= \langle \texttt{error}, \emptyset, \epsilon \rangle \\ S' &= \langle \texttt{error}, \emptyset, \epsilon \rangle \end{aligned}$$

    Hence $S \longmapsto^{1,1}_{pcek} S'$ via the rule *car*, and $\mathcal{T}_{\mathcal{E}}(S') = S''$.
- The other cases are similar.

∎

The Step Correspondence Lemma implies that a *touch* optimized program exhibits the same behavior as the corresponding unoptimized program.

Theorem 3.4 *For* $P \in \Lambda^0_a$, $\underline{eval_{pcek}}(\mathcal{T}_{sba(P)}(P)) = eval_{pcek}(P)$.

Proof: Both the fact that $\underline{eval_{pcek}}$ is well-defined on $\mathcal{T}_{sba(P)}(P)$ and that the equality holds follow from Lemma 3.3. ∎

In summary, the *touch* optimization algorithm we present removes redundant *touch* operations from programs based on the information provided by set-based analysis. This optimization algorithm is provably-correct with respect to the semantics of **future** as specified by the extended evaluator $\underline{eval_{pcek}}$. Any implementation that realizes $\underline{eval_{pcek}}$ correctly can therefore make use of our optimization technique.

4 Set-Based Analysis for Futures

The development of a sound program analysis consists of two parts. First, we use the transition rules of the $P(CEK)$-machine to derive constraints on the sets of run-time values that a variable in a given program may assume. Any set environment satisfying these constraints is a valid set environment. Second, we develop an algorithm for finding the minimal (*i.e.*, most accurate) set environment satisfying these constraints. Our set constraints are similar to the constraints in Heintze's work on set-based analysis for SML [11], but our derivation differs from his.

4.1 Deriving Set Constraints for Program Variables

A set constraint is of the form:

$$\frac{A}{B}$$

where A and B are statements concerning $\mathcal{E}$, and A also depends on the program being analyzed. A set environment $\mathcal{E}$ *satisfies* this constraint if whenever A holds for $\mathcal{E}$, then B also holds for $\mathcal{E}$.

Suppose P is the program of interest, and suppose that the evaluation of P involves the transition $S \longmapsto^{n,m}_{pcek} S'$. We derive constraints on $\mathcal{E}$ sufficient to ensure that $S' \models \mathcal{E}$ if $S \models \mathcal{E}$. We proceed by case analysis on the last transition rule used for $S \longmapsto^{n,m}_{pcek} S'$, and we present four representative cases:

- Suppose $S \longmapsto^{1,1}_{pcek} S'$ via the transition rule (*bind-const*):

  $$\langle (\mathbf{let}\ (x\ c)\ M), E, K\rangle \ \longmapsto^{1,1}_{pcek}\ \langle M, E[x \leftarrow c], K\rangle$$

  The transition pairs x with the constant c in the extended environment. To ensure that the set $\mathcal{E}(x)$ includes c, we introduce the constraint:

  $$\frac{(\mathbf{let}\ (x\ c)\ M) \in P}{c \in \mathcal{E}(x)} \qquad (C_1^P)$$

  This constraint requires that for each term of the form $(\mathbf{let}\ (x\ c)\ M)$ occurring in P, the constant c must be recorded in $\mathcal{E}$ as one of the possible values of the variable x.

- Suppose $S \longmapsto^{1,1}_{pcek} S'$ via the transition rule (*apply*): In the interesting case, y is bound, either directly or via a placeholder object, to a closure $\langle (\lambda x'.\, N), E'\rangle$, which implies that:

  $$\langle (\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle N, E'[x' \leftarrow E(z)], \langle \mathbf{ar}\ x, M, E\rangle.K\rangle$$

  Then this rule binds x' to the value $E(z)$. To ensure that $\mathcal{E}$ accounts for the binding of x' to $E(z)$, we demand that $\mathcal{E}$ satisfy the constraint:

  $$\frac{\begin{array}{cc}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P & V_y \in \mathcal{E}(y) \\ \mathit{touch}_{pcek}[V_y] = \langle (\lambda x'.\, N), E\rangle & V \in \mathcal{E}(z)\end{array}}{V \in \mathcal{E}(x')} \qquad (C_7^P)$$

- Suppose $S \longmapsto^{1,1}_{pcek} S'$ via the transition rule (*return*):

  $$\langle x, E', \langle \mathbf{ar}\ y, M, E\rangle.K\rangle \ \longmapsto^{1,1}_{pcek}\ \langle M, E[y \leftarrow E'(x)], K\rangle$$

  To acount for transitions according to (*return*), our constraint system must ensure that $\mathcal{E}(y)$ includes the value $E'(x)$. However, a syntax-directed program analysis cannot extract the possible activation records that may receive the value of the final variable in a procedure from the variable or its immediate context. However, it will determine all potential call sites of the procedure and can enforce a relationship between the "return" variable of the procedure and the variable that receives the result of the function call. Thus, if *FinalVarN* is a function that determines the innermost ("result") variable of N, then the crucial constraint on this variable is as follows:

  $$\frac{\begin{array}{cc}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P & V_y \in \mathcal{E}(y) \\ \mathit{touch}_{pcek}[V_y] = \langle (\lambda x'.\, N), E\rangle & V \in \mathcal{E}(\mathit{FinalVar}[N])\end{array}}{V \in \mathcal{E}(x)} \qquad (C_8^P)$$

  The definition of *FinalVar* is straightforward.

Definition 4.1. (*FinalVar*)

$$\begin{array}{rcl} \mathit{FinalVar} : \Lambda_a & \longrightarrow & \mathit{Vars} \\ \mathit{FinalVar}[x] & = & x \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ V)\ M)] & = & \mathit{FinalVar}[M] \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)] & = & \mathit{FinalVar}[M] \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M)] & = & \mathit{FinalVar}[M] \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M)] & = & \mathit{FinalVar}[M] \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M)] & = & \mathit{FinalVar}[M] \\ \mathit{FinalVar}[(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M)] & = & \mathit{FinalVar}[M] \end{array}$$

∎

- The analysis for the transition rules (*future-id*), (*fork*) and (*join*) is analogous to the above case. The constraints C^P_{10} and C^P_{11} ensure that, for each **future** expression in the program, the set environment $\mathcal{E}$ accounts for the bindings created by any (*future-id*), (*fork*) or (*join*) transitions that correspond to that **future** expression.

Examining each of the transition rules of the machine in a similar manner results in eleven program-based *set constraints* $C^P_1, \ldots, C^P_{11}$: see Figure 6.

4.2 Soundness of the Set Constraints

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

Assume that $\mathcal{E}$ satisfies $C^P_1, \ldots, C^P_{11}$. To prove that $P \models \mathcal{E}$, we need to show that $\langle P, \emptyset, \epsilon\rangle \longmapsto^{n,m}_{pcek} S$ implies $S \models \mathcal{E}$. The natural approach is to proceed by induction on n. As part of the proof, we will need to consider intermediate transitions starting from states other than $\langle P, \emptyset, \epsilon\rangle$. Therefore, we need to strengthen the induction hypothesis to:

$$S \models_P \mathcal{E} \text{ and } S \longmapsto^{n,m}_{pcek} S' \text{ implies } S' \models_P \mathcal{E}$$

where the relation $S \models_P \mathcal{E}$ is an appropriately chosen relation. This relation needs to assert a number of properties about the state S in order to support the proof of the induction hypothesis:

- First, the relation needs to assert that the terms contained in S must occur in P, *i.e.*, the evaluation of programs does not involve the creation of new terms.

- Next, the relation needs to assert that S can only contain values and environments that are *compatible with* P and $\mathcal{E}$. A value V is *compatible with* P and $\mathcal{E}$, written $V \odot_P \mathcal{E}$, if each term in V occurs in P and each environment in V is compatible with P and $\mathcal{E}$. Similarly, an environment E is *compatible with* P and $\mathcal{E}$, written $E \odot_P \mathcal{E}$, if each binding in E occurs in $\mathcal{E}$ and each value in E is compatible with P and $\mathcal{E}$.

Definition 4.2. ($V \odot_P \mathcal{E}, E \odot_P \mathcal{E}$)

– The relation $V \odot_P \mathcal{E}$ is the smallest relation satisfying the following clauses:

$$\begin{array}{rll@{\qquad}rll} c \odot_P \mathcal{E} & \text{if} & c \text{ occurs in } P & & & \\ x \odot_P \mathcal{E} & \text{if} & x \text{ occurs in } P & \langle \mathbf{ph}\ p\ \circ\rangle \odot_P \mathcal{E} & & \\ \langle(\lambda x.\,M), E\rangle \odot_P \mathcal{E} & \text{if} & (\lambda x.\,M) \in P \text{ and } E \odot_P \mathcal{E} & \langle \mathbf{ph}\ p\ V\rangle \odot_P \mathcal{E} & \text{if} & V \odot_P \mathcal{E} \\ (\mathbf{cons}\ V_1\ V_2) \odot_P \mathcal{E} & \text{if} & V_1 \odot_P \mathcal{E} \text{ and } V_2 \odot_P \mathcal{E} & & & \end{array}$$

$$\frac{(\mathbf{let}\ (x\ c)\ M) \in P}{c \in \mathcal{E}(x)} \qquad (C_1^P)$$

$$\frac{(\mathbf{let}\ (x\ y)\ M) \in P \qquad V \in \mathcal{E}(y)}{V \in \mathcal{E}(x)} \qquad (C_2^P)$$

$$\frac{(\mathbf{let}\ (x\ (\lambda y.\,N))\ M) \in P \qquad \forall x \in dom(E).\ E(x) \in \mathcal{E}(x)}{\langle(\lambda y.\,N), E\rangle \in \mathcal{E}(x)} \qquad (C_3^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{cons}\ y_1\ y_2))\ M) \in P \qquad V_1 \in \mathcal{E}(y_1) \qquad V_2 \in \mathcal{E}(y_2)}{(\mathbf{cons}\ V_1\ V_2) \in \mathcal{E}(x)} \qquad (C_4^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M) \in P \qquad V_y \in \mathcal{E}(y) \qquad touch_{pcek}[V_y] = (\mathbf{cons}\ V_1\ V_2)}{V_1 \in \mathcal{E}(x)} \qquad (C_5^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M) \in P \qquad V_y \in \mathcal{E}(y) \qquad touch_{pcek}[V_y] = (\mathbf{cons}\ V_1\ V_2)}{V_2 \in \mathcal{E}(x)} \qquad (C_6^P)$$

$$\frac{\begin{array}{c}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P \qquad V_y \in \mathcal{E}(y)\\ touch_{pcek}[V_y] = \langle(\lambda x'.\,N), E\rangle \qquad V \in \mathcal{E}(z)\end{array}}{V \in \mathcal{E}(x')} \qquad (C_7^P)$$

$$\frac{\begin{array}{c}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P \qquad V_y \in \mathcal{E}(y)\\ touch_{pcek}[V_y] = \langle(\lambda x'.\,N), E\rangle \qquad V \in \mathcal{E}(FinalVar[N])\end{array}}{V \in \mathcal{E}(x)} \qquad (C_8^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M) \in P \qquad V \in \mathcal{E}(FinalVar[M_1]) \cup \mathcal{E}(FinalVar[M_2])}{V \in \mathcal{E}(x)} \qquad (C_9^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M) \in P \qquad V_N \in \mathcal{E}(FinalVar[N])}{V_N \in \mathcal{E}(x) \qquad \langle\mathbf{ph}\ p\ V_N\rangle \in \mathcal{E}(x)} \qquad (C_{10}^P)$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M) \in P}{\langle\mathbf{ph}\ p\ \circ\rangle \in \mathcal{E}(x)} \qquad (C_{11}^P)$$

FIGURE **6** Set Constraints on $\mathcal{E}$ with respect to P.

– $E \odot_P \mathcal{E}$ holds if for all $x \in dom(E)$, $E(x) \in \mathcal{E}(x)$ and $E(x) \odot_P \mathcal{E}$.

∎

- The relation must also assert that $\mathcal{E}$ already contains the bindings that could be created during a subsequent (*return*) transition to an existing activation record. This requirement is necessary to support the induction hypothesis in the case of a (*return*) transition. It is enforced by C_8^P.

- Similarly, the relation must assert that $\mathcal{E}$ already contains the bindings that could be created during a subsequent (*future-id*) or (*join*) transition. The formalization of this assertion refers to the auxiliary function *ResultVar*. The result of any state S will be the potential values of the variable *ResultVar*$[S]$, where *ResultVar* is the following function from states to variables:

Definition 4.3. (*ResultVar*)

$$\begin{array}{rcl} \mathit{ResultVar} : \mathit{State}_{pcek} & \longrightarrow & \mathit{Vars} \\ \mathit{ResultVar}[\langle M, E, \epsilon\rangle] & = & \mathit{FinalVar}[M] \\ \mathit{ResultVar}[\langle M, E, K.\langle \mathbf{ar?}\, x, N, E'\rangle\rangle] & = & \mathit{FinalVar}[N] \\ \mathit{ResultVar}[(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2)] & = & \mathit{ResultVar}[S_2] \end{array}$$

∎

- Finally, to allow for error transitions, the relation must hold for the error state `error`.

The complete relation relation, which asserts all of the above properties, is defined by induction on the structure of states.

Definition 4.4. ($S \models_P \mathcal{E}$)

$$\begin{array}{rcl} (\mathbf{f\text{-}let}\ (p\ S_1)\ S_2) \models_P \mathcal{E} & \Leftrightarrow & \left\{ \begin{array}{l} S_1 \models_P \mathcal{E}, S_2 \models_P \mathcal{E} \\ \text{and } \forall V \in \mathcal{E}(\mathit{ResultVar}[S_1]),\ S_2[p := V] \models_P \mathcal{E} \end{array} \right. \\ \langle M, E, K\rangle \models_P \mathcal{E} & \Leftrightarrow & \left\{ \begin{array}{l} M \in P, E \odot_P \mathcal{E} \\ \text{and } K = \langle \mathbf{ar?}\, x_1, M_1, E_1\rangle.\cdots.\langle \mathbf{ar?}\, x_n, M_n, E_n\rangle \\ \text{and } x_i \in P, M_i \in P, E_i \odot_P \mathcal{E} \text{ for } 1 \leq i \leq n \\ \text{and } \mathcal{E}(\mathit{final\text{-}var}(M_i)) \subseteq \mathcal{E}(x_{i+1}) \text{ for } 1 \leq i < n \\ \text{and } \mathcal{E}(\mathit{final\text{-}var}(M)) \subseteq \mathcal{E}(x_1) \text{ if } n > 0 \\ \text{and if } K = K_1.\langle \mathbf{ar}\dagger\ x, M, E\rangle.K_2 \text{ then} \\ \qquad \forall p \in \mathit{Ph\text{-}Vars}, \langle \mathbf{ph}\ p\ \circ\rangle \in \mathcal{E}(x) \text{ and} \\ \qquad \forall V \in \mathcal{E}(\mathit{ResultVar}[\langle M, E, K_1\rangle]), \langle \mathbf{ph}\ p\ V\rangle \in \mathcal{E}(x) \end{array} \right. \\ \texttt{error} \models_P \mathcal{E} & & \end{array}$$

∎

The invariant relation is a stronger relation than the validates relation, *i.e.*, if a states satisfies the invariant relation for a set environment with respect to a given program, then that state obviously validates the set environment.

Lemma 4.5 *$S \models_P \mathcal{E}$ implies $S \models \mathcal{E}$*

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 successors of S.

Lemma 4.6 *Suppose $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{11}^P$. Then $S \models_P \mathcal{E}$ and $S \longmapsto_{pcek}^{n,m} S'$ implies $S' \models_P \mathcal{E}$.*

Proof: By induction on n and by case analysis of the last transition rule for $S \longmapsto_{pcek}^{n,m} S'$. We present the cases for the transition rules (*car*), (*apply*) and (*return*) in detail.

- Suppose $S \longmapsto_{pcek}^{1,1} S'$ via the transition rule (*bind-const*). Then:

$$\begin{array}{rcl} S & = & \langle (\mathbf{let}\ (x\ c)\ M), E, K\rangle \\ S' & = & \langle M, E[x \leftarrow c], K\rangle \end{array}$$

  Since $S \models_P \mathcal{E}$, we know that $(\mathbf{let}\ (x\ c)\ M) \in P$, $E \odot_P \mathcal{E}$ and the conditions on K in the definition of the invariant are satisfied. Because $\mathcal{E}$ satisfies C_1^P, we have that $c \in \mathcal{E}(x)$. Also, by definition, $c \odot_P \mathcal{E}$. Therefore $E[x \leftarrow c] \odot_P \mathcal{E}$, which implies that $S' \models_P \mathcal{E}$.

- Suppose $S \longmapsto^{1,1}_{pcek} S'$ via the transition rule (*apply*). Then:

  $$S = \langle(\textbf{let}\ (x\ (\textbf{apply}\ y\ z))\ M), E, K\rangle$$

  Since the analysis of the error case is trivial, we only consider the case where y is bound, either directly or via a placeholder object, to a closure $\langle(\lambda x'.\,N), E'\rangle$. In this case:

  $$\begin{aligned} touch_{pcek}[E(y)] &= \langle(\lambda x'.\,N), E'\rangle \\ S' &= \langle N, E'[x' \leftarrow E(z)], \langle \textbf{ar}\ x, M, E\rangle.K\rangle \end{aligned}$$

  It is easy to see that $N \in P$. To show that the new environment $E'[x' \leftarrow E(z)]$ is compatible with P and $\mathcal{E}$, we proceed as follows:
  From the relation $S \models_P \mathcal{E}$ we know that $E \odot_P \mathcal{E}$. But $E \odot_P \mathcal{E}$ implies that $E(y) \odot_P \mathcal{E}$, which in turn implies $E' \odot_P \mathcal{E}$. Also, from $E \odot_P \mathcal{E}$ we have that $E(z) \in \mathcal{E}(z)$, which, since $\mathcal{E}$ satisfies C^P_7, implies that $E(z) \in \mathcal{E}(x')$. Finally, $E \odot_P \mathcal{E}$ implies that $E(z) \odot_P \mathcal{E}$. Combining the facts $E' \odot_P \mathcal{E}$, $E(z) \in \mathcal{E}(x')$ and $E(z) \odot_P \mathcal{E}$, we have that $E'[x' \leftarrow E(z)] \odot_P \mathcal{E}$, *i.e.*, the new environment is compatible with P and $\mathcal{E}$.
  Finally, we must show that the appropriate conditions on the new continuation $\langle \textbf{ar}\ x, M, E\rangle.K$ hold. That $\mathcal{E}(\mathit{FinalVar}[N]) \subseteq \mathcal{E}(x)$ follows from C^P_8, and the remaining conditions on the continuation are implied by the relation $S \models_P \mathcal{E}$. Therefore, the invariant holds for S'.

- The final case we consider is for the transition rule (*return*). For this case:

  $$\begin{aligned} S &= \langle x, E', \langle \textbf{ar}\ y, M, E\rangle.K\rangle \\ S' &= \langle M, E[y \leftarrow E'(x)], K\rangle \end{aligned}$$

  That $M \in P$ follows from the relation $S \models_P \mathcal{E}$. To show that the new environment $E[y \leftarrow E'(x)]$ is compatible with P and $\mathcal{E}$ we proceed as follows:
  From the relation $S \models_P \mathcal{E}$ we know that $E' \odot_P \mathcal{E}$, and hence $E'(x) \odot_P \mathcal{E}$. The invariant also implies that $\mathcal{E}(x) \subseteq \mathcal{E}(y)$, which in turn implies that $E'(x) \in \mathcal{E}(y)$. Finally, the invariant implies that $E \odot_P \mathcal{E}$. Putting together the facts $E'(x) \odot_P \mathcal{E}$, $E'(x) \in \mathcal{E}(y)$ and $E \odot_P \mathcal{E}$, we have that $E[y \leftarrow E'(x)] \odot_P \mathcal{E}$, *i.e.*, the new environment is compatible with P and $\mathcal{E}$.
  The required conditions on K follow from the relation $S \models_P \mathcal{E}$. Therefore, the invariant holds for S'.

The remaining cases have similar proofs. ∎

Since the invariant trivially holds for the initial state $\langle P, \emptyset, \epsilon\rangle$, 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 4.7 (Soundness of Constraints) *If $\mathcal{E}$ satisfies $C^P_1, \ldots, C^P_{11}$, then $P \models \mathcal{E}$.*

Proof: Suppose $\mathcal{E}$ satisfies $C^P_1, \ldots, C^P_{11}$. Let S be a state such that $\langle P, \emptyset, \epsilon\rangle \longmapsto^{n,m}_{pcek} S$. By the definition of the invariant relation, $\langle P, \emptyset, \epsilon\rangle \models_P \mathcal{E}$. Therefore Lemma 4.6 implies that $S \models_P \mathcal{E}$, and hence by Lemma 4.5, $S \models \mathcal{E}$. The latter is true for any state derivable from the initial state $\langle P, \emptyset, \epsilon\rangle$, 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.

4.3 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:

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

Smaller set environments correspond to more accurate approximations, since they include fewer extraneous potential values per variable. Therefore, we define set-based analysis as a function that returns the least set environment satisfying the set constraints.

Definition 4.8. (sba)

$$\begin{array}{rcl} sba : \Lambda_a & \longrightarrow & SetEnv_P \\ sba(P) & = & \sqcap\{\mathcal{E} \mid \mathcal{E} \text{ satisfies } C_1^P, \ldots, C_{11}^P\} \end{array}$$

∎

The function sba is well-defined. Since the set constraints are monotonic, it follows that $sba(P)$ is a valid set environment for P. Furthermore, a value V is in $sba(P)(x)$ if and only if from the assumption that a set environment $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{11}^P$ we can prove that $V \in \mathcal{E}(x)$. All that remains is to produce an algorithm for calculating $sba(P)$.

4.4 Solving the Set Constraints

Since $sba(P)$ typically maps variables to infinite sets of possible values, we need a finite representation for these infinite sets. A systematic inspection of the set constraints suggests that the set of closures for a λ-expression can be represented by the λ-expression itself, that the set of closed pairs for a **cons**-expression can be represented by the **cons**-expression, *etc.* The actual sets of run-time values can easily be reconstructed from the representative terms and the set environment. In short, we can take the set of *abstract values* for a program P to be:

$$\overline{V} \in AbsValue_P ::= c_P \mid (\lambda x.\, M)_P \mid (\mathbf{cons}\ x\ y)_P \mid \langle \mathbf{ph}\ x_P \rangle \mid \langle \mathbf{ph}\ \circ \rangle$$

where the constant c_P, the λ-expression $(\lambda x.\, M)_P$, the pair $(\mathbf{cons}\ x\ y)_P$ and the variable x_P are all the respective subterms of P. The size of $AbsValue_P$ is $O(|P|)$, where $|P|$ is the length of P.

Abstract values provide a finite representation for the infinite set environments encountered in set-based analysis. Specifically, an *abstract set environment* $\overline{\mathcal{E}}$ is a mapping from variables in P to finite sets of abstract values. Each abstract value $\overline{V}$ in $\overline{\mathcal{E}}(x)$ represents a set of run-time values V (depending on $\overline{\mathcal{E}}$) according to the relation $V\ in_{\overline{\mathcal{E}}} \overline{V}$, and in a similar manner each set $\overline{\mathcal{E}}(x)$ of abstract values represents a set of machine values according to the relation $V\ in_{\overline{\mathcal{E}}}\ x$:

$$\begin{array}{rcl} c\ in_{\overline{\mathcal{E}}}\ c_p & & \\ \langle(\lambda x.\, M), E\rangle\ in_{\overline{\mathcal{E}}}\ (\lambda x.\, M)_P & \Leftrightarrow & \forall x \in dom(E).\ E(x)\ in_{\overline{\mathcal{E}}}\ x \\ (\mathbf{cons}\ V_1\ V_2)\ in_{\overline{\mathcal{E}}}\ (\mathbf{cons}\ y_1\ y_2) & \Leftrightarrow & V_i\ in_{\overline{\mathcal{E}}}\ y_i \\ \langle \mathbf{ph}\ p\ V\rangle\ in_{\overline{\mathcal{E}}}\ \langle \mathbf{ph}\ y_P\rangle & \Leftrightarrow & V\ in_{\overline{\mathcal{E}}}\ y \\ \langle \mathbf{ph}\ p\ \circ\rangle\ in_{\overline{\mathcal{E}}}\ \langle \mathbf{ph}\ \circ\rangle & & \\ & & \\ V\ in_{\overline{\mathcal{E}}}\ x & \Leftrightarrow & \exists \overline{V} \in \overline{\mathcal{E}}(x) \text{ with } V\ in_{\overline{\mathcal{E}}} \overline{V} \end{array}$$

$$\frac{(\mathbf{let}\ (x\ c)\ M) \in P}{c_P \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_1^P})$$

$$\frac{(\mathbf{let}\ (x\ y)\ M) \in P \qquad \overline{V} \in \overline{\mathcal{E}}(y)}{\overline{V} \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_2^P})$$

$$\frac{(\mathbf{let}\ (x\ (\lambda y.\, N))\ M) \in P}{(\lambda y.\, N)_P \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_3^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{cons}\ y_1\ y_2))\ M) \in P \qquad \overline{\mathcal{E}}(y_1) \neq \emptyset \qquad \overline{\mathcal{E}}(y_2) \neq \emptyset}{(\mathbf{cons}\ y_1\ y_2)_P \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_4^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M) \in P \qquad \overline{V}_y \in \overline{\mathcal{E}}(y) \qquad (\mathbf{cons}\ z_1\ z_2)_P \in \overline{touch}[\overline{\mathcal{E}}, \overline{V}_y\,] \qquad \overline{V} \in \overline{\mathcal{E}}(z_1)}{\overline{V} \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_5^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M) \in P \qquad \overline{V}_y \in \overline{\mathcal{E}}(y) \qquad (\mathbf{cons}\ z_1\ z_2)_P \in \overline{touch}[\overline{\mathcal{E}}, \overline{V}_y\,] \qquad \overline{V} \in \overline{\mathcal{E}}(z_2)}{\overline{V} \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_6^P})$$

$$\frac{\begin{array}{c}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P \qquad \overline{V}_y \in \overline{\mathcal{E}}(y) \\ (\lambda x'.\, N)_P \in \overline{touch}[\overline{\mathcal{E}}, \overline{V}_y\,] \qquad \overline{V} \in \overline{\mathcal{E}}(z)\end{array}}{\overline{V} \in \overline{\mathcal{E}}(x')} \qquad (\overline{C_7^P})$$

$$\frac{\begin{array}{c}(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M) \in P \qquad \overline{V}_y \in \overline{\mathcal{E}}(y) \\ (\lambda x'.\, N)_P \in \overline{touch}[\overline{\mathcal{E}}, \overline{V}_y\,] \qquad \overline{V} \in \overline{\mathcal{E}}(\mathit{FinalVar}[N])\end{array}}{\overline{V} \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_8^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M) \in P \qquad \overline{V} \in \overline{\mathcal{E}}(\mathit{FinalVar}[M_1]) \cup \overline{\mathcal{E}}(\mathit{FinalVar}[M_2])}{\overline{V} \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_9^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M) \in P \qquad \overline{V}_V \in \overline{\mathcal{E}}(\mathit{FinalVar}[N])}{\overline{V}_V \in \overline{\mathcal{E}}(x) \qquad \langle \mathbf{ph}\ \mathit{FinalVar}[N]_P \rangle \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_{10}^P})$$

$$\frac{(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M) \in P}{\langle \mathbf{ph}\ \circ \rangle \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_{11}^P})$$

Auxiliary Function:

$$\overline{touch} : \mathit{AbsEnv}_P \times \mathit{AbsValue}_P \longrightarrow \mathcal{P}(\mathit{AbsValue}_P)$$

$$\overline{touch}[\overline{\mathcal{E}}, \overline{V}\,] = \begin{cases} \{\overline{V}\} & \text{if } \overline{V} = c_P \text{ or } \overline{V} = (\lambda x.\, M)_P \text{ or } \overline{V} = (\mathbf{cons}\ x\ y)_P \\ \{\overline{W} \mid \overline{U} \in \overline{\mathcal{E}}(y) \text{ and } \overline{W} \in \overline{touch}[\overline{\mathcal{E}}, \overline{U}\,]\} & \text{if } \overline{V} = \langle \mathbf{ph}\ y_P \rangle \end{cases}$$

FIGURE **7** Abstract Constraints on $\overline{\mathcal{E}}$ with respect to P.

The class of all abstract set environments for P given program is denoted AbsEnv_P. Each abstract set environment $\overline{\mathcal{E}}$ is a finite representation of a potentially infinite set environment, according to the following function:

$$\begin{array}{rcl} \mathcal{F} : \mathit{AbsEnv}_P & \longrightarrow & \mathit{SetEnv}_P \\ \mathcal{F}\left(\overline{\mathcal{E}}\right)(x) & = & \{V \mid V\ \mathit{in}_{\overline{\mathcal{E}}}\ x\} \end{array}$$

Reformulating the set constraints from Figure 6 for abstract set environments produces the *abstract constraints* $\overline{C_1^P}, \ldots, \overline{C_{11}^P}$ on $\overline{\mathcal{E}}$ with respect to P: see Figure 7. We define $\overline{sba}(P)$ to be the least abstract set environment satisfying the abstract constraints with respect to P.

Definition 4.9. ($\overline{sba}$)

$$\begin{array}{rcl} \overline{sba} : \Lambda_a & \longrightarrow & AbsEnv_P \\ \overline{sba}(P) & = & \sqcap\{\overline{\mathcal{E}} \mid \overline{\mathcal{E}} \text{ satisfies } \overline{C_1^P}, \ldots, \overline{C_{11}^P}\} \end{array}$$

∎

Since each of the abstract constraints in monotonic, $\overline{sba}(P)$ satisfies $\overline{C_1^P}, \ldots, \overline{C_{11}^P}$. Furthermore $\overline{V} \in \overline{sba}(P)(x)$ if and only if we can prove, based on the assumption that an abstract set environment $\overline{\mathcal{E}}$ satisfies $\overline{C_1^P}, \ldots, \overline{C_{11}^P}$, that $\overline{V} \in \overline{\mathcal{E}}(x)$.

The correspondence between set constraints and abstract constraints implies that $\overline{sba}(P)$ is a finite representation for $sba(P)$.

Theorem 4.10 (Correctness of Abstraction) $sba(P) = \mathcal{F}(\overline{sba}(P))$

Proof: See Appendix A. ∎

The class of abstract set environments forms a complete lattice of size $O(|P|^2)$ under the natural pointwise partial order. Therefore we can calculate $\overline{sba}(P)$ in an iterative manner, starting from the empty abstract set environment $\overline{\mathcal{E}}(x) = \emptyset$, and repeatedly extending $\overline{\mathcal{E}}$ with additional bindings as required by the set constraints, until $\overline{\mathcal{E}}$ contains all the required bindings. Since we can extend $\overline{\mathcal{E}}$ at most $O(|P|^2)$ times, this algorithm terminates. Furthermore, each time we extend $\overline{\mathcal{E}}$ with a new binding, calculating the additional bindings implied by that new binding takes at most $O(|P|)$ time. Hence, the entire algorithm runs in $O(|P|^3)$ time. We include an implementation of this algorithm in Appendix B.

Optimization algorithms can interpret the abstract set environment $\overline{sba}(P)$ in a straightforward manner. For example, the query on $sba(P)$ from the *touch* optimization algorithm:

$$sba(P)(y) \subseteq PValue_{pcek}$$

is equivalent to the following query on $\overline{sba}(P)$:

$$\overline{sba}(P)(y) \subseteq \{c_P, (\lambda x.\, M)_P, (\mathbf{cons}\ x\ y)_P\}$$

In a similar manner other queries on $sba(P)$ can easily be reformulated in terms of $\overline{sba}(P)$.

5 Experimental Results

We extended the Gambit compiler [5, 6], which makes no attempt to remove *touch* operations from programs, with a preprocessor that implements the set-based analysis algorithm and the *touch* optimization algorithm. The analysis and the optimization algorithm are as described in the previous sections extended to a sufficiently large subset of functional Scheme.[3] We used the extended Gambit compiler to test the effectiveness of *touch* optimization on the suite of benchmarks contained in Feeley's Ph.D. thesis [5] on a GP1000 shared-memory multiprocessor [2]. Figure 8 describes these benchmarks.

Each benchmark was tested on the original compiler (*standard*) and on the modified compiler (*touch optimized*). The results of the test runs are documented in Figure 9. The first two columns

[3] Five of the benchmarks include a small number (one or two per benchmark) of explicit *touch* operations for coordinating side-effects. They do not affect the validity of the analysis and *touch* optimization algorithms.

Program	Description
`fib`	Computes the 25^{th} fibonacci number using a doubly-recursive algorithm.
`queens`	Computes the number of solutions to the n-queens problem, for $n = 10$.
`rantree`	Traverses a binary tree with 32768 nodes.
`mm`	Multiplies two 50 by 50 matrices of integers.
`scan`	Computes the parallel prefix sum of a vector of 32768 integers.
`sum`	Uses a divide-and-conquer algorithm to sum a vector of 32768 integers.
`tridiag`	Solves a tridiagonal system of 32767 equations.
`allpairs`	Parallel Floyd's algorithm, computes shortest path between all pairs in a 117 node graph.
`abisort`	Sorts 16384 integers using the adaptive bitonic sort algorithm.
`mst`	Computes the minimum spanning tree of a 1000 node graph.
`qsort`	Uses a parallel Quicksort algorithm to sort 1000 integers.
`poly`	Computes the square of a 200 term polynomial, and evaluates the resulting polynomial.

FIGURE **8** Description of the Benchmark Programs

Benchmark	*standard*		*touch optimized*				
	touches ($n = 1$)		*touches* ($n = 1$)		speedup over *standard* (%)		
	count(K)	overhead(%)	count(K)	overhead(%)	$n = 1$	$n = 4$	$n = 16$
`fib`	1214	85.0	122	10.2	40.5	39.9	36.7
`queens`	2116	41.2	35	1.5	28.1	30.4	28.1
`rantree`	327	67.5	14	2.6	38.7	37.2	26.8
`mm`	1828	121.0	3	<1	54.7	44.1	23.6
`scan`	1278	126.8	66	4.1	54.1	43.4	19.0
`sum`	525	107.3	33	6.1	48.8	37.9	20.0
`tridiag`	811	110.8	7	<1	52.1	29.4	5.8
`allpairs`	32360	150.4	14	<1	60.0	39.6	<1
`abisort`	5751	106.5	9	<1	51.3	31.1	24.4
`mst`	20422	91.4	750	5.3	45.0	17.2	<1
`qsort`	253	43.3	78	19.9	16.4	<1	<1
`poly`	526	65.3	121	16.2	29.7	12.5	<1

FIGURE **9** Benchmark Results

present the number of *touch* operations performed during the execution of a benchmark using the *standard* compiler (column 1), and the sequential execution overhead of these *touch* operations (column 2). To determine the absolute overhead of *touch*, we also ran the programs on a single processor after removing all *touch* operations. The next two columns contain the corresponding measurements for the *touch optimizing* compiler. The *touch* optimization algorithm reduces the number of *touch* operations to a small fraction of the original number (column 3), thus reducing the average overhead of *touch* operations from approximately 90% to less than 10% (column 4).

The last three columns show the relative speedup of each benchmark for one, four, and 16 processor configurations, respectively. The number compares the running time of the benchmarks using the standard compiler with the optimizing compiler. As expected, the relative speedup *decreases* as the number of processors *increases*, because the execution time is then dominated by other factors, such as memory contention and communication costs. For most benchmarks, the benefit of our *touch* optimization is still substantial, producing an average speedup over the *standard* compiler of 37% on four processors, and of 20% on 16 processors. The exceptions are the last three benchmarks, `mst`, `qsort`, and `poly`. However, even Feeley [5] described these as "poorly parallel" programs, in which the effects of memory contention and communication costs are especially visible. It is therefore not surprising that our optimizing compiler does not improve the running time in these cases.

6 Related Work

Kranz et al. [13, 20] briefly describe a simplistic algorithm for *touch* optimization based on a first-order type analysis. The algorithm lowers the *touch* overhead to 65% from 100% in standard benchmarks, that is, it is significantly less effective than our *touch* optimization. The paper does not address the semantics of **future** or the well-foundedness of the optimizations. Knopp [18] reports the existence of a *touch* optimization algorithm based on abstract interpretation. His paper presents neither a semantics nor the abstract interpretation. He only reports the reduction of *static* counts of *touch* operations for an implementation of Common Lisp with **future**. Neither paper gives an indication concerning the expense of the analysis algorithms.[4]

At *LFP*'94, Jaganathan and Weeks [15] described an analysis for explicitly parallel symbolic programs, which they intend to use in a forthcoming compiler. They remark that the analysis could be used for *touch* optimizations. Their semantics and their derivation of the analysis significantly differ from ours so that we have not been able to compare the two analyses in detail. They do not have an implementation of their algorithm for a full language like functional Scheme, and they do not have optimization algorithms that exploit the results of their analysis.

Much work has been done on the static analysis of *sequential* programs, including abstract interpretation [3] and 0CFA [24]. Our analysis is most closely follows Heintze's work on set-based analysis for the sequential language ML [10], but the extension of this technique to parallel languages requires a substantial reformulation of the derivation and correctness proof. 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 presents set constraints whose solution is the minimal safe set environment. We start from an parallel abstract machine and avoid these intermediate steps by deriving our set constraints and proving their correctness directly from the abstract machine semantics.

Other techniques for static analysis of sequential programs include abstract interpretation [3, 4] and Shivers' 0CFA [24]. The relationship between abstract interpretation and set-based analysis was covered by Heintze [10].

Sequential optimization techniques such as tagging optimization [12] and soft-typing [26] are similar in character to *touch* optimization. Both techniques remove the type-dispatches required for dynamic type-checking wherever possible, without changing the behavior of programs, in the same fashion as we remove *touch* operations. However, the analyses relies on conventional type inference techniques.

---

[4] Ito's group [Ito: personal communication, April 22, 1994] reports an attempt at *touch* optimization based on abstract interpretation. His group abandoned the effort due to the exponential cost of the abstract interpretation algorithm.

7 Conclusion

The development of a semantics for **future**s directly leads to the derivation of a powerful program analysis. The analysis is computationally inexpensive but yields enough information to eliminate numerous implicit *touch* operations. We believe that the construction of this simple *touch* optimization algorithm clearly illustrates how semantics can contribute to the development of advanced compilers. We intend to use our semantic characterization for the derivation of other program optimizations in Gambit and for the design of truly transparent **future** annotations for languages with imperative constructs.

Acknowledgments We thank Marc Feeley for discussions concerning *touch* optimizations and for his assistance in testing the effectiveness of our algorithm, and Nevin Heintze for discussions on set-based analysis and for access to his implementation of set based analysis for ML.

A Correctness Proof for Abstract Representation

We use in_P as shorthand for $in_{\overline{sba}(P)}$.

Theorem 4.10 (Correctness of Abstraction) $\mathcal{F}\left(\overline{sba}(P)\right) = sba(P)$

Proof: We show the equivalence of $sba(P)$ and $\mathcal{F}\left(\overline{sba}(P)\right)$ by proving the set inclusion in both directions. Let x be a variable in $Vars_P$. Then:

$$\begin{array}{rll} & V \in \mathcal{F}\left(\overline{sba}(P)\right)(x) & \\ \Rightarrow & V \; in_P \; x & \text{by definition of } \mathcal{F} \\ \Rightarrow & V \in sba(P)(x) & \text{by Lemma A.1} \end{array}$$

Also:

$$\begin{array}{rll} & V \in sba(P)(x) & \\ \Rightarrow & \exists \overline{V} \in \overline{sba}(P)(x) \text{ with } V \; in_P \; \overline{V} & \text{by Lemma A.3} \\ \Rightarrow & V \; in_P \; x & \\ \Rightarrow & V \in \mathcal{F}\left(\overline{sba}(P)\right)(x) & \text{by definition of } \mathcal{F} \end{array}$$

Hence $\mathcal{F}\left(\overline{sba}(P)\right) = sba(P)$. ∎

Lemma A.1 ($\overline{sba}$ includes sba) *$V \; in_P \; x$ implies $V \in sba(P)(x)$.*

Proof: Assume that $V \; in_P \; x$. Then there exists $\overline{V} \in \overline{sba}(P)(x)$ with $V \; in_P \; \overline{V}$. Hence there must be a proof, based on the assumption that $\overline{sba}(P)$ satisfies the abstract constraints, that proves $\overline{V} \in \overline{sba}(P)(x)$. We proceed by lexicographic induction on the size of V and on the length of this proof, and by case analysis on the last abstract constraint used in the proof.

$\overline{C_1^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_1^P}$. Then $\overline{V} = c_P$, $V = c$ and (**let** $(x\ c)\ M$) occurs in P, and hence by C_1^P we have that $V \in sba(P)(x)$.

$\overline{C_2^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_2^P}$. Then (**let** $(x\ y)\ M$) occurs in P. Also, $\overline{V} \in \overline{sba}(P)(y)$ via a shorter proof, so by induction we have $V \in sba(P)(y)$. Using constraint C_2^P gives us $V \in sba(P)(x)$.

$\overline{C_3^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_3^P}$. Then (**let** $(x\ (\lambda y.\, N))\ M$) occurs in P, $\overline{V} = (\lambda y.\, M)_P$ and $V = \langle(\lambda y.\, N), E\rangle$ where for each $z \in dom(E)$, $E(z)\ in_P\ z$. Each $E(z)$ is smaller than V, therefore by induction we have that for each $z \in dom(E)$, $E(z) \in sba(P)(z)$. Using constraint C_3^P then shows that $V \in sba(P)(x)$.

$\overline{C_4^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_4^P}$. Then (**let** $(x$ (**cons** $y_1\ y_2))\ M$) occurs in P, $\overline{V} =$ (**cons** $y_1\ y_2)_P$ and $V =$ (**cons** $V_1\ V_2$) where $V_i\ in_P\ y_i$. Each V_i is smaller than V, therefore by induction we have that $V_i \in sba(P)(y_i)$. Using constraint C_4^P then shows that $V \in sba(P)(x)$.

$\overline{C_5^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_5^P}$. Then (**let** $(x$ (**car** $y))\ M$) occurs in P. We first with the case where $\overline{V}_y$ is not an abstract placeholder. In this case $\overline{V}_y =$ (**cons** $z_1\ z_2)_P$ and $\overline{V}_y \in \overline{sba}(P)(y)$ via a shorter proof. By Lemma A.2, there exists $V'\ in_P$ (**cons** $z_1\ z_2)_P$, therefore there exists $V_2\ in_P\ z_2$. Since $V\ in_P\ \overline{V}$ and $\overline{V} \in \overline{sba}(z_1)$, (**cons** $V\ V_2)\ in_P$ (**cons** $z_1\ z_2$). By induction we have that (**cons** $V\ V_2) \in sba(P)(y)$, and finally C_5^P implies that $V \in sba(P)(x)$.

The case where $\overline{V}_y$ is an abstract placeholder follows by a similar argument.

$\overline{C_6^P}$ The analysis for the constraint $\overline{C_6^P}$ is similar to the previous case.

$\overline{C_7^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x')$ via constraint $\overline{C_7^P}$. Again, we first with the case where $\overline{V}_y$ is not an abstract placeholder. In this case $(\lambda x'.\, N)_P \in \overline{sba}(P)(y)$ and $\overline{V} \in \overline{sba}(P)(z)$ via shorter proofs. Since $\langle(\lambda x'.\, N), \emptyset\rangle\ in_P\ (\lambda x'.\, N)_P$, by induction we have that $\langle(\lambda x'.\, N), \emptyset\rangle \in sba(P)(y)$, and that $V \in sba(P)(z)$. An application of C_7^P proves that $V \in sba(P)(x')$.

The case where $\overline{V}_y$ is an abstract placeholder follows by a similar argument.

$\overline{C_8^P}$ The analysis for the constraint $\overline{C_8^P}$ is similar to the previous case.

$\overline{C_9^P}, \overline{C_{10}^P}$ The analysis for the constraint $\overline{C_9^P}$ and the left implication of $\overline{C_{10}^P}$ is similar to the analysis for $\overline{C_2^P}$.

$\overline{C_{10}^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via the right implication of constraint $\overline{C_{10}^P}$, where $\overline{V} = \langle$**ph** $FinalVar[N]\rangle$ and $V\ in_P\ \langle$**ph** $FinalVar[N]\rangle$. Then $V = \langle$**ph** $p\ V_N\rangle$ where $V_N\ in_P\ FinalVar[N]$. By induction, $V_N \in sba(P)(FinalVar[N])$. Hence the constraint C_{10}^P implies that $V \in sba(P)(x)$.

$\overline{C_{11}^P}$ The analysis for the constraint $\overline{C_{11}^P}$ is trivial.

∎

Lemma A.2 (Non-emptiness of abstract values) *$\overline{V} \in \overline{sba}(P)(x)$ implies there exists $V\ in_P\ \overline{V}$.*

Proof: The proof is by induction on the length of the proof that $\overline{V} \in \overline{sba}(P)(x)$. We proceed by case analysis on the last constraint used in the proof.

$\overline{C_1^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_1^P}$. Then $\overline{V} = c_P$, and $c\ in_P\ \overline{V}$.

$\overline{C_2^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_2^P}$. Then $\overline{V} \in \overline{sba}(P)(y)$ via a shorter proof, and the lemma holds by induction.

$\overline{C_3^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_3^P}$. Then $\overline{V} = (\lambda y.\, M)_P$ and hence $\langle(\lambda y.\, M), \emptyset\rangle\ in_P\ \overline{V}$.

$\overline{C_4^P}$ Suppose $\overline{V} \in \overline{sba}(P)(x)$ via constraint $\overline{C_4^P}$. Then $\overline{V} = (\mathbf{cons}\ y_1\ y_2)_P$, and $\overline{V}_i \in \overline{sba}(P)(y_i)$, for some V_i, $i = 1, 2$, via shorter proofs. Therefore by induction, there exists $V_i\ in_P\ \overline{V}_i$, and hence $(\mathbf{cons}\ V_1\ V_2)\ in_P\ \overline{V}$.

- The analysis of the constraints $\overline{C_5^P}$, $\overline{C_6^P}$, $\overline{C_7^P}$, $\overline{C_8^P}$, $\overline{C_9^P}$ and the left implication of $\overline{C_{10}^P}$ is similar to the analysis for $\overline{C_2^P}$.

$\overline{C_{10}^P}$ Suppose $\langle\mathbf{ph}\ \mathit{FinalVar}[N]\rangle \in \overline{sba}(P)(x)$ via constraint $\overline{C_{10}^P}$. Then we have $\overline{V}_N \in \overline{sba}(P)(\mathit{FinalVar}[N])$ via a shorter proof, so by induction there exists $V\ in_P\ \overline{V}_N$. Hence we have that $\langle\mathbf{ph}\ p\ V\rangle\ in_{\overline{\mathcal{E}}}\ \langle\mathbf{ph}\ \mathit{FinalVar}[N]\rangle$.

∎

Lemma A.3 (*sba* **includes** $\overline{sba}$) *$V \in sba(P)(x)$ implies there exists $\overline{V} \in \overline{sba}(P)(x)$ with $V\ in_P\ \overline{V}$.*

Proof: Assume that $V \in sba(P)(x)$. Then there exists a proof, based on the assumption that $sba(P)$ satisfies the set constraints, that proves that $V \in sba(P)(x)$. We proceed by induction on the length of this proof, and by case analysis on the last step in the proof.

C_1^P We take $\overline{V} = c_P$, and the lemma holds.

C_2^P For this case $V \in sba(P)(y)$ via a shorter proof, so by induction there exists $\overline{V} \in \overline{sba}(P)(y)$ with $V\ in_P\ \overline{V}$. Then $\overline{C_2^P}$ implies that $\overline{V} \in \overline{sba}(P)(x)$.

C_3^P In this case, $V = \langle(\lambda y.\, N), E\rangle$, and for each $z_i \in dom(E)$. $E(z_i) \in sba(P)(z_i)$ via shorter proofs. Therefore, there exist $\overline{V}_i$ such that $\overline{V}_i \in \overline{sba}(P)(z_i)$ and $E(z_i)\ in_P\ \overline{V}_i$. Now take $\overline{V} = (\lambda y.\, N)_P$, then by $\overline{C_3^P}$ we have $\overline{V} \in \overline{sba}(P)(x)$, and $\langle(\lambda y.\, N), E\rangle\ in_P\ \overline{V}$.

C_4^P In this case, $V = (\mathbf{cons}\ V_1\ V_2)$, and $V_i \in sba(P)(y_i)$ via shorter proofs. Therefore, there exist $\overline{V}_i \in \overline{sba}(P)(y_i)$ with $V_i\ in_P\ \overline{V}_i$. Now take $\overline{V} = (\mathbf{cons}\ y_1\ y_2)$, then by $\overline{C_4^P}$ we have that $\overline{V} \in \overline{sba}(P)(x)$.

C_5^P Suppose $V_1 \in sba(P)(x)$ via constraint C_5^P. For simplicity, we consider only the case where V_y is not a placeholder, since the case involving placeholder is similar but more complicated.

Thus $V_y = (\mathbf{cons}\ V_1\ V_2) \in sba(P)(y)$ via a shorter proof. By induction there exists $\overline{V}_y$ such that $\overline{V}_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ \overline{V}_y$. Hence $\overline{V}_y = (\mathbf{cons}\ z_1\ z_2)_P$, $\overline{V}_1 \in \overline{sba}(P)(z_1)$ and $V_1\ in_P\ \overline{V}_1$. From $\overline{C_5^P}$ we can prove $\overline{V}_1 \in \overline{sba}(P)(x)$, and the lemma holds for this case.

C_6^P This case is analogous to the previous case.

C_7^P Suppose $V \in sba(P)(x')$ via constraint C_7^P. For simplicity, we consider only the case where V_y is not a placeholder, since the case involving placeholder is similar but more complicated.

Thus $V_y = \langle(\lambda x'.\, N), E\rangle \in sba(P)(y)$ via a shorter proof. By induction there exists $\overline{V}_y$ such that $\overline{V}_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ \overline{V}_y$. Hence $\overline{V} = (\lambda x'.\, N)_P$. Also, $V \in sba(P)(z)$ via a shorter proof, so by induction there exists $\overline{V}$ such that $\overline{V} \in \overline{sba}(P)(y)$ and $V\ in_P\ \overline{V}$. Hence, by $\overline{C_7^P}$ we have $\overline{V} \in \overline{sba}(P)(x')$.

C_8^P Again, we consider only the case where V is not a placeholder.
Thus $V_y = \langle(\lambda x'.\, N), E\rangle$, and $V_y \in sba(P)(y)$ via a shorter proof. By induction there exists $\overline{V}_y$ such that $\overline{V}_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ \overline{V}_y$. Hence $\overline{V}_y = (\lambda x'.\, N)_P$. Also, $V \in sba(P)(FinalVar[N])$ via a shorter proof, so by induction there exists $\overline{V}$ such that $\overline{V} \in \overline{sba}(P)(FinalVar[N])$ and $V\ in_P\ \overline{V}$. Hence, by $\overline{C_8^P}$ we have $\overline{V} \in \overline{sba}(P)(x)$.

C_9^P, C_{10}^P The analysis for the constraint C_9^P and the left implication of C_{10}^P is similar to the analysis for C_2^P.

C_{10}^P Suppose $\langle \mathbf{ph}\ p\ V_N\rangle \in sba(P)(x)$ via the right implication of C_{10}^P. Then we have $V_N \in sba(P)(FinalVar[N])$ via a shorter proof, so by induction there exists $\overline{V}_N \in \overline{sba}(P)(FinalVar[N])$ with $V_N\ in_P\ \overline{V}_N$. By $\overline{C_{10}^P}$ we have that $\langle \mathbf{ph}\ FinalVar[N]\rangle \in \overline{sba}(P)(x)$, and $\langle \mathbf{ph}\ p\ V_N\rangle\ in_P\ \langle \mathbf{ph}\ FinalVar[N]\rangle$.

C_{11}^P This case is trivial.

∎

B Set Based Analysis Algorithm

A complete $O(|P|^3)$ set-based analysis algorithm for the intermediate language Λ_a is included in Figure 10[5]. The algorithm is written in Scheme extended with a special form **match** for pattern matching.

The algorithm relies on a group of auxiliary functions that maintain, for each program variable, an associated set of abstract values and an associated set of constraints, both of which are initially empty. The function *add-absvalue!* extends the abstract value set of a variable; the predicate *in-absvalue-set?* tests for membership in that set, and the iterator function (*foreach-absvalue var fn*) calls *fn* on each abstract value associated with the variable *var*. The functions *add-aconstraint!*, *in-constraint-set?* and *foreach-constraint* behave in a similar manner on constraint sets. We assume that the functions *add-absvalue!*, *add-aconstraint!*, *in-absvalue-set?* and *in-constraint-set?* operate in constant time, and that the functions *foreach-absvalue* and *foreach-constraint* operate in time linear in the number of elements in the appropriate set.

The function *SBA* traverses an expression to ensure that the expression satisfies the abstract constraints, and returns the final variable of that expression. Certain constraints cannot be satisfied immediately. For example, the constraint $\overline{C_2^P}$ requires that all abstract values in $\overline{\mathcal{E}}(y)$ must be in $\overline{\mathcal{E}}(x)$, but during the analysis of a program we may not yet know all the abstract values that may be added to $\overline{\mathcal{E}}(y)$. Therefore, we associate a constraint '(`propogate-to` ,x) with the variable y. This constraint ensures that whenever an abstract value is added to $\overline{\mathcal{E}}(y)$, then that abstract value is also propogated to $\overline{\mathcal{E}}(x)$. The function *interpret-constraint* is called for constraint and abstract value associated with a given variable, and ensures that the constraint is satisfied, by creating new abstract values or new constraints as necessary.

The $O(|P|^3)$ time bound can be verified as follows: Each call of the functions *new-absvalue!* 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 abstract values, and the number of constraints are all $O(|P|)$. Therefore the test conditions of the two **unless** expressions succeed at most $O(|P|^2)$

[5] For simplicity, the algorithm implements simpler versions of the abstract constraints $\overline{C_4^P}$ and $\overline{C_{10}^P}$ that do not include that non-emptyness conditions $\overline{\mathcal{E}}(y_i) \neq \emptyset$ and $\overline{\mathcal{E}}(FinalVarN) \neq \emptyset$. We believe the extra degree of approximation introduced by this simplification is negligible in practice.

times each. Hence the body of each **unless** expression is executed 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.

References

[1] Baker, H., and Hewitt, C. The incremental garbage collection of processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages* (1977), vol. 12(8), pp. 55–59.

[2] BBN Advanced Computers, Inc., Cambridge, MA. *Inside the GP1000.* 1989.

[3] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analyses of programs by consruction or approximation of fixpoints. In *POPL* (1977), pp. 238–252.

[4] Cousot, P., and Cousot, R. Higer order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages. *ICCL* (1994), 95–112.

[5] Feeley, M. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, Brandeis University, 1993.

[6] Feeley, M., and Miller, J. S. A parallel virtual machine for efficient scheme compilation. In *LFP* (1990).

[7] Flanagan, C., and Felleisen, M. The semantics of Future. Rice University Comp. Sci. TR94-238.

[8] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *PLDI* (1993), pp. 237–247.

[9] Halstead, R. Multilisp: A language for concurrent symbolic computataion. *ACM Transactions on Programming Languages and Systems 7*, 4 (1985), 501–538.

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

[11] Heintze, N. Set-based analysis of ML programs. In *LFP* (1994), pp. 306–317.

[12] Henglein, F. Global tagging optimization by type inference. In *LFP* (1992), pp. 205–215.

[13] Ito, T., and Halstead, R., Eds. *Parallel Lisp: Languages and Systems.* Springer-Verlag *Lecture Notes in Computer Science* **441**, 1989.

[14] Ito, T., and Matsui, M. A parallel lisp language: Pailisp and its kernel specification. [15:58–100].

[15] Jagannathan, S., and Weeks, S. Analyzing stores and references in a parallel symbolic language. In *LFP* (1994), pp. 294–305.

[16] Katz, M., and Weise, D. Continuing into the future: on the interaction of futures and first-class continuations. In *LFP* (1990).

```
(define SBA
  (lambda (expression)
    (match expression
      [(? variable? x) x]
      [`(let (,x ,exp) ,body)
       (match exp
         [(? constant? c) (new-absvalue! x c)]
         [(? variable? y) (new-constraint! y `(propogate-to ,x))]
         [`(cons ,y1 ,y2) (new-absvalue! x `(cons ,y1 ,y2))]
         [`(lambda ,y ,N)
          (let ([finalvar-N (SBA N)])
            (new-absvalue! x `(lambda ,y ,N ,finalvar-N)))]
         [`(future ,N)
          (let ([finalvar-N (SBA N)])
            (new-constraint! finalvar-N `(propogate-to ,x))
            (new-absvalue! x `(ph ,finalvar-N))
            (new-absvalue! x '(ph-circ)))]
         [`(car ,y) (new-constraint! y `(propogate-car-to ,x))]
         [`(cdr ,y) (new-constraint! y `(propogate-cdr-to ,x))]
         [`(if ,y ,M1 ,M2)
          (let ([finalvar-M1 (SBA M1)][finalvar-M2 (SBA M2)])
            (new-constraint! finalvar-M1 `(propogate-to ,x))
            (new-constraint! finalvar-M2 `(propogate-to ,x)))]
         [`(apply ,y ,z) (new-constraint! y `(application ,x ,z))])
       (SBA body)])))

(define new-absvalue!
  (lambda (var absvalue)
    (unless (in-absvalue-set? var absvalue)
      (add-absvalue! var absvalue)
      (foreach-constraint var (lambda (constraint) (interpret-constraint constraint absvalue))))))

(define new-constraint!
  (lambda (var constraint)
    (unless (in-constraint-set? var constraint)
      (add-constraint! var constraint)
      (foreach-absvalue var (lambda (absvalue) (interpret-constraint constraint absvalue))))))

(define interpret-constraint
  (lambda (constraint absvalue)
    (match (cons constraint absvalue)
      [`((propogate-to ,x) . ,v) (new-absvalue! x v)]
      [`((propogate-car-to ,x) . (cons ,y1 ,y2)) (new-constraint! y1 `(propogate-to ,x))]
      [`((propogate-cdr-to ,x) . (cons ,y1 ,y2)) (new-constraint! y2 `(propogate-to ,x))]
      [`((application ,result ,arg) . (lambda ,para ,_ ,finalvar))
       (new-constraint! finalvar `(propogate-to ,result))
       (new-constraint! arg `(propogate-to ,para))]
      [_ (void)])))
```

FIGURE **10** The Set-Based Analysis Algorithm.

[17] Kessler, R.R., and R. Swanson. Concurrent scheme. [15:200–234].

[18] Knopp, J. Improving the performance of parallel lisp by compile time analysis. [15:271–277].

[19] Kranz, D., Halstead, R., and Mohr, E. Mul-T: A high-performance parallel lisp. [15:306–321].

[20] Kranz, D., Halstead, R., and Mohr, E. Mul-T: A high-performance parallel lisp. In *PLDI* (1989), pp. 81–90.

[21] Miller, J. *MultiScheme: A Parallel Processing System.* PhD thesis, MIT, 1987.

[22] Mohr, E., Kranz, R., and Halstead, R. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP* (1990).

[23] Moreau, L. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* PhD thesis, Universite de Liege, 1994.

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

[25] Swanson, M., Kessler, R., and Lindstrom, G. An implementation of portable standard lisp on the BBN butterfly. In *LFP* (1988), pp. 132–142.

[26] Wright, A. and R. Cartwright. A practical soft type system for scheme. In *LFP* (1994), pp. 250–262.