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

Copyright ©1994 by

Cormac Flanagan and Matthias Felleisen

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 at all placeholder-strict positions of program operations to ensure proper synchronization between threads. These *touch* operations contribute substantially to a program's execution time. 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 significantly reduces the overhead imposed by *touch* operations.

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

Contents

1 Futures, Touches and Transparency **1**

2 Review: The Language and Its Parallel Semantics **2**

3 Touch Optimization **4**
- 3.1 Non-touching Primitives 6
- 3.2 The Touch Optimization Algorithm 7

4 Set-Based Analysis for Futures **11**
- 4.1 Deriving Set Constraints for Program Variables 11
- 4.2 Soundness of the Set Constraints 15
- 4.3 From Set Constraints to Set-Based Analysis 19
- 4.4 Performing Set-Based Analysis 20

5 Experimental Results **23**

6 Related Work **25**

7 Conclusion **26**

A Correctness Proof for Representation **26**

B Set Based Analysis Algorithm **31**

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 [2, 11] 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 mechanism [6, 16, 22, 23, 24] and on the extension of the language with higher-order control constructs [18, 25]. 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 [8], we developed a series of semantics for an idealized functional language with **future**. The last semantics is particularly suited to the development of analysis and optimization algorithms, because it exposes appropriate details regarding program executions. Here we 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 [6] 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 [8]. 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. This technical report is an expansion of a preliminary conference report [9].

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 [10] of an extended λ-calculus-like language that includes conditionals and a **future** construct: see Figure 1.

M	$\in$	Λ_a	$::=$	x	(Terms)
				$\mid$ (**let** (x V) M)	
				$\mid$ (**let** (x (**future** M)) M)	
				$\mid$ (**let** (x (**car** y)) M)	
				$\mid$ (**let** (x (**cdr** y)) M)	
				$\mid$ (**let** (x (**if** y M M)) M)	
				$\mid$ (**let** (x (**apply** y z)) M)	
V	$\in$	*Value*	$::=$	$c \mid x \mid (\lambda x.\, M) \mid$ (**cons** x y)	(Values)
x	$\in$	*Vars*	$=$	$\{x, y, z, \ldots\}$	(Variables)
c	$\in$	*Const*	$=$	$\{\texttt{nil}, 0, 1, \ldots\}$	(Constants)

FIGURE 1: The A-normalized Language Λ_a

In a previous paper [8], 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 well-suited for the development of program analyses, since it contains explicit binding information relating program variables to their values. The machine is also will-suited for 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; $f : A \longrightarrow_f B$ denotes that f is a finite map from A to B; $M \in P$ denotes that the term M occurs in the

Evaluator:

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

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

Data Specifications:

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

Auxiliary Functions:

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

$$\begin{array}{rcl}
touch_{pcek} : Value_{pcek} & \longrightarrow & PValue_{pcek} \cup \{\circ\} \\
touch_{pcek}[\langle\textbf{ph}\ p\ \circ\rangle] & = & \circ \\
touch_{pcek}[\langle\textbf{ph}\ p\ V\rangle] & = & touch_{pcek}[V] \\
touch_{pcek}[V] & = & V \text{ if } V \in PValue_{pcek}
\end{array}$$

Placeholder Substitution $S[p := V]$, $E[p := V]$, $K[p := V]$:

$$\begin{array}{rcl}
(\textbf{f-let}\ (p'\ S_1)\ S_2)[p := V] & = & \begin{cases} (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2) & \text{if } p = p' \\ (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2[p := V]) & \text{if } p \neq p' \end{cases} \\
\langle M, E, K\rangle[p := V] & = & \langle M, E[p := V], K[p := V]\rangle \\
\texttt{error}[p := V] & = & \texttt{error} \\
E[p := V] & = & E' \text{ such that } E'(x) = \begin{cases} \langle\textbf{ph}\ p\ V\rangle & \text{if } E(x) = \langle\textbf{ph}\ p\ \circ\rangle \\ E(x) & \text{otherwise} \end{cases} \\
\epsilon[p := V] & = & \epsilon \\
(\langle\textbf{ar}\ x, M, E\rangle.K)[p := V] & = & \langle\textbf{ar}\ x, M, E[p := V]\rangle.K[p := V] \\
(\langle\textbf{ar}\dagger\ x, M, E\rangle.K)[p := V] & = & \langle\textbf{ar}\dagger\ x, M, E[p := V]\rangle.K[p := V]
\end{array}$$

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

program P; and $\langle\textbf{ar}?\ x, M, E\rangle$ refers to either a normal activation record $\langle\textbf{ar}\ x, M, E\rangle$ or a tagged activation record $\langle\textbf{ar}\dagger\ x, M, E\rangle$.

Transition Rules:

$$\langle(\textbf{let}\ (x\ c)\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle M, E[x \leftarrow c], K\rangle \qquad (\textit{bind-const})$$

$$\langle(\textbf{let}\ (x\ y)\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle M, E[x \leftarrow E(y)], K\rangle \qquad (\textit{bind-var})$$

$$\langle(\textbf{let}\ (x\ (\lambda y.N))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle M, E[x \leftarrow \langle(\lambda y.N), E'\rangle], K\rangle \quad \text{where } E' = E|_{FV[(\lambda y.\,N)]} \qquad (\textit{bind-lam})$$

$$\langle(\textbf{let}\ (x\ (\textbf{cons}\ y\ z))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle M, E[x \leftarrow (\textbf{cons}\ E(y)\ E(z))], K\rangle \qquad (\textit{bind-cons})$$

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

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

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

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

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

$$\langle(\textbf{let}\ (x\ (\textbf{future}\ N))\ M), E, K\rangle \longmapsto^{1,1}_{pcek} \langle N, E, \langle \textbf{ar}\dagger\ x, M, E\rangle.K\rangle \qquad (\textit{future})$$

$$\langle x, E, \langle \textbf{ar}\dagger\ y, M, E'\rangle.K\rangle \longmapsto^{1,1}_{pcek} \langle M, E'[y \leftarrow E(x)], K\rangle \qquad (\textit{future-id})$$

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

$$(\textbf{f-let}\ (p\ \langle x, E, \epsilon\rangle)\ S) \longmapsto^{1,1}_{pcek} S[p := E(x)] \qquad (\textit{join})$$

$$(\textbf{f-let}\ (p\ \texttt{error})\ S) \longmapsto^{1,1}_{pcek} \texttt{error} \qquad (\textit{join-error})$$

$$(\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ S_1)\ S_2))\ S_3) \longmapsto^{1,1}_{pcek} (\textbf{f-let}\ (p_1\ S_1)\ (\textbf{f-let}\ (p_2\ S_2)\ S_3)) \quad p_1 \notin FP(S_3) \qquad (\textit{lift})$$

$$(\textbf{f-let}\ (p\ S_1)\ S_2) \longmapsto^{a+c,b}_{pcek} (\textbf{f-let}\ (p\ S_1')\ S_2') \quad \text{if } S_1 \longmapsto^{a,b}_{pcek} S_1',\ S_2 \longmapsto^{c,d}_{pcek} S_2' \qquad (\textit{parallel})$$

$$S \longmapsto^{0,0}_{pcek} S \qquad (\textit{reflexive})$$

$$S \longmapsto^{a+c,b+d}_{pcek} S'' \quad \text{if } S \longmapsto^{a,b}_{pcek} S',\ S' \longmapsto^{c,d}_{pcek} S'',\ a, c > 0 \qquad (\textit{transitive})$$

FIGURE 3: The $P(CEK)$-machine: Transition Rules

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* (see Figure 4) performs 1.2 million *touch* operations during the computation of (*fib* 25).

```
(define fib
  (lambda (n)
    (if (< n 2)
        1
        (let ([r1 (future (fib (- n 1)))])
          (let ([r2 (fib (- n 2))])
            (+ r1 r2))))))
```

FIGURE 4: The definition of *fib*.

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 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 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.

However, 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 [29, 1]. 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:

```
(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 for every execution:

```
(if (placeholder? x) (unchecked-car (touch x)) (unchecked-car x))
```

[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 implemented as a pointer comparison.

Performing these *placeholder?* checks can add a significant overhead to the execution time. Kranz [21] and Feeley [6] 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 [6, 19, 27]. 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 provably possible[2]. 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 **car**, **cdr**, **if** and **apply**, respectively:

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

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 unspecified, and any arbitrary state may be produced. The extended language is called $\underline{\Lambda_a}$.

We define the semantics of the extended language $\underline{\Lambda_a}$ by extending the $P(\overline{CEK})$-machine with the additional transition rules described in Figure 5. The evaluator for the extended language, $\underline{eval_{pcek}}$, is defined in the usual way (cmp. Figures 2). Unlike $eval_{pcek}$, the evaluator $\underline{eval_{pcek}}$ is no longer a function. There are programs in $\underline{\Lambda_a}$ for which the evaluator

[2] This approach is similar in character to optimization techniques such as soft-typing [29, 1] and tagging optimization [14], which remove the type-dispatches required in dynamically-typed languages wherever possible. However, touch-optimization also applies to a statically-typed language with **future**s, where soft-typing and tagging optimization techniques are not as useful.

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

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

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

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

FIGURE 5: Non-*touch*ing transition rules

$\underline{eval_{pcek}}$ 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** $(x\ (\textbf{car}\ y))\ M$) and we can prove that y is never bound to a placeholder. Then we can replace the expression with (**let** $(x\ (\underline{\textbf{car}}\ y))\ M$), which the machine can execute 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[3] 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 $\mathcal{E} : \mathit{Vars}_P \to \mathcal{P}(\mathit{Value}_{pcek})$ is a *set environment* for P.

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

$$
\begin{aligned}
\mathcal{T}_{\mathcal{E}} : \Lambda_a &\longrightarrow \underline{\Lambda_a} \\
\mathcal{T}_{\mathcal{E}}[x] &= x \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ c)\ M)] &= (\textbf{let } (x\ c)\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ y)\ M)] &= (\textbf{let } (x\ y)\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\lambda y.N))\ M)] &= (\textbf{let } (x\ (\lambda y.\mathcal{T}_{\mathcal{E}}(N)))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{cons } y\ z))\ M)] &= (\textbf{let } (x\ (\textbf{cons } y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) \\
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{future } N))\ M)] &= (\textbf{let } (x\ (\textbf{future } \mathcal{T}_{\mathcal{E}}[N]))\ \mathcal{T}_{\mathcal{E}}[M])
\end{aligned}
$$

$$
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{car } y))\ M)] = \begin{cases} (\textbf{let } (x\ (\underline{\textbf{car}}\ y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq PValue_{pcek} \\ (\textbf{let } (x\ (\textbf{car } y))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq PValue_{pcek} \end{cases}
$$

$$
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{cdr } y))\ M)] = \text{analogous to } \textbf{car}
$$

$$
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{if } y\ M_1\ M_2))\ M)] = \begin{cases} (\textbf{let } (x\ (\underline{\textbf{if}}\ y\ M_1\ M_2))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq PValue_{pcek} \\ (\textbf{let } (x\ (\textbf{if } y\ M_1\ M_2))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq PValue_{pcek} \end{cases}
$$

$$
\mathcal{T}_{\mathcal{E}}[(\textbf{let } (x\ (\textbf{apply } y\ z))\ M)] = \begin{cases} (\textbf{let } (x\ (\underline{\textbf{apply}}\ y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \subseteq PValue_{pcek} \\ (\textbf{let } (x\ (\textbf{apply } y\ z))\ \mathcal{T}_{\mathcal{E}}[M]) & \text{if } \mathcal{E}(y) \not\subseteq PValue_{pcek} \end{cases}
$$

FIGURE 6: The touch optimization algorithm $\mathcal{T}$

- $E \models \mathcal{E}$ (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 $E \models \mathcal{E}$.
  - $(\textbf{f-let } (p\ S_1)\ S_2) \models \mathcal{E}$ 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 set environment $\mathcal{E}$, is defined in Figure 6. If $\mathcal{E}$ is valid for the program being optimized, then the touch optimization 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 optimized program $\mathcal{T}_{\mathcal{E}}(P)$. We extend $\mathcal{T}_{\mathcal{E}}$ in a natural manner to states to aid in the proof of this correspondence.

Lemma 3.3 (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 $S \longmapsto^{n,m}_{pcek} S'$. 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 $S = \langle(\textbf{let}\ (x\ (\textbf{car}\ y))\ M), E, K\rangle$. There are four sub-cases to consider, depending on whether or not $\mathcal{E}(y)$ contains placeholders and whether or not $E(y)$ is a pair.
  - Assume $\mathcal{E}(y) \not\subseteq PValue_{pcek}$ and $touch_{pcek}[E(y)] = (\textbf{cons}\ V_1\ V_2)$. Then:

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

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

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

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

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

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

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

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

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

Main Lemma 3.4 *For $P \in \Lambda^0_a$, if $P \models \mathcal{E}$ then $\underline{eval_{pcek}}(\mathcal{T}_{\mathcal{E}}(P)) = eval_{pcek}(P)$.*

Proof: Both the fact that $\underline{eval_{pcek}}$ is well-defined on $\mathcal{T}_{\mathcal{E}}(P)$ 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 a valid set environment. 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 proceeds in two stages. First, we use the transition rules of the $P(CEK)$-machine to derive constraints on the sets of run-time values that variables 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 [13], but our derivation substantially differs from his.

4.1 Deriving Set Constraints for Program Variables

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 derive constraints on $\mathcal{E}$ sufficient to ensure that each transition during the evaluation of P preserves the validity of $\mathcal{E}$. Suppose the evaluation of P involves the transition $S \longmapsto^{n,m}_{pcek} S'$. Then we require that the constraints on $\mathcal{E}$ imply that if $S \models \mathcal{E}$ then $S' \models \mathcal{E}$. We proceed by case analysis on the last transition rule used for $S \longmapsto^{n,m}_{pcek} S'$ and present six representative cases:

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

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

   The transition pairs x with the constant c in the extended environment. To ensure that the set $\mathcal{E}(x)$ includes c, only need to ensure that for each term of the form $(\textbf{let } (x\ c)\ M)$ occurring in P, the constant c must be recorded in $\mathcal{E}$ as one of the possible values of the variable x. This suggests the constraint:

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

2. 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:

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

   A transition according to this rule binds x' to the value $E(z)$. To ensure that the set $\mathcal{E}(x')$ includes the value $E(z)$, we demand that $\mathcal{E}(x')$ satisfy the constraint:

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

3. 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 account 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. Instead, it can 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 $\mathit{FinalVar}[\,N\,]$ is a function that determines the innermost ("result") variable of N, then the crucial constraint on this variable is as follows:

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

The definition of *FinalVar* is straightforward.

Definition 4.1. (*FinalVar*)

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

∎

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

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

A transition according to this rule returns the current value of x to a tagged activation record. As with the previous case, our analysis cannot extract the possible tagged activation records that may receive the value of x from the variable or its immediate context. Instead, by examining each **future**-expression $(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)$, the analysis determines which variable x may receive the value of $\mathit{FinalVar}[\,N\,]$ via a (*future-id*) transition. The following constraint then ensures that $\mathcal{E}$ accounts for these bindings:

$$\frac{(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M) \in P \qquad V \in \mathcal{E}(\mathit{FinalVar}[\,N\,])}{V \in \mathcal{E}(x)} \qquad (C_{10}^P)$$

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

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

$$\frac{(\textbf{let}\ (x\ (\lambda y.\, N))\ M) \in P \quad dom(E) = FV[(\lambda y.\, N)] \quad \forall x \in dom(E).\ E(x) \in \mathcal{E}(x)}{\langle (\lambda y.\, N), E\rangle \in \mathcal{E}(x)} \qquad (C_3^P)$$

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

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

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

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

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

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

$$\frac{(\textbf{let}\ (x\ (\textbf{future}\, N))\ M) \in P \qquad V \in \mathcal{E}(FinalVar[N])}{V \in \mathcal{E}(x)} \qquad (C_{10}^P)$$

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

$$\frac{(\textbf{let}\ (x\ (\textbf{future}\, N))\ M) \in P \qquad V \in \mathcal{E}(FinalVar[N])}{\langle \textbf{ph}\ p\ V\rangle \in \mathcal{E}(x)} \qquad (C_{12}^P)$$

FIGURE 7: Set Constraints on $\mathcal{E}$ with respect to P.

5. Suppose $S \longmapsto^{1,0}_{pcek} S'$ via the transition rule (*fork*):

$$\langle M, E, K_1.\langle \textbf{ar}\dagger\ x, N, E'\rangle.K_2\rangle \longmapsto^{1,0}_{pcek} (\textbf{f-let}\ (p\ \langle M, E, K_1\rangle)\ \langle N, E'[x \leftarrow \langle \textbf{ph}\ p\ \circ\rangle], K_2\rangle) \quad p \notin FP(E') \cup FP(K_2)$$

Transitions according to this rule return an undetermined placeholder $\langle \textbf{ph}\ p\ \circ\rangle$ to the continuation $\langle \textbf{ar}\dagger\ x, N, E'\rangle.K_2$ of a **future** expression. To ensure that $\overline{\mathcal{E}}$ accounts for bindings created by such transitions, we impose a second constraint on **future**-expressions:

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

6. A transition according to the rule (*join*) replaces each occurrence of an undetermined placeholder with a determined placeholder containing the result of the appropriate **future** expression:

$$(\textbf{f-let}\ (p\ \langle x, E, \epsilon\rangle)\ S)\ \longmapsto^{1,1}_{pcek}\ S[p := E(x)]$$

As with cases 3 and 4, our analysis cannot determine the possible placeholders that may receive the value of a variable x from the variable or its immediate context. Instead, to account for the environment bindings that are modified by (*join*) transitions, we introduce a third constraint on **future** expressions:

$$\frac{(\textbf{let}\ (x\ (\textbf{future}\ N))\ M) \in P \qquad V \in \mathcal{E}(\mathit{FinalVar}[\,N\,])}{\langle \textbf{ph}\ p\ V\rangle \in \mathcal{E}(x)} \qquad (C^P_{12})$$

Examining each of the transition rules of the machine in a similar manner results in twelve program-based *set constraints* $C^P_1, \ldots, C^P_{12}$: see Figure 7.

Set Constraints for a Sample Program To illustrate how the set constraints determine valid set environments, consider the following program *cons-it*. The program defines a function *cons-it* that returns a cons-cell containing two copies of it's argument, and the program then applies this function to the numbers 1 and 2, producing results $pair_1$ and $pair_2$ respectively.

```
(let (cons-it
       (lambda x . (let (r (cons x x)) r)))
  (let (t1 1)
    (let (pair1 (cons-it t1))
      (let (t2 2)
        (let (pair2 (cons-it t2))
          pair2)))))
```

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

From C^P_3: $$\langle(\lambda x.(\textbf{let}\ (r\ (\textbf{cons}\ x\ x))\ r)), \emptyset\rangle \in \mathcal{E}(\mathit{cons\text{-}it}) \qquad (1)$$

From C^P_4: $$\frac{V_1 \in \mathcal{E}(x) \qquad V_2 \in \mathcal{E}(x)}{(\textbf{cons}\ V_1\ V_2) \in \mathcal{E}(r)} \qquad (2)$$

From C^P_1: $$1 \in \mathcal{E}(t_1) \qquad (3)$$

From C^P_7: $$\frac{\langle(\lambda x'.N), E\rangle \in \mathcal{E}(\mathit{cons\text{-}it}) \qquad V \in \mathcal{E}(t_1)}{V \in \mathcal{E}(x')} \qquad (4)$$

From C^P_8: $$\frac{\langle(\lambda x'.N), E\rangle \in \mathcal{E}(\mathit{cons\text{-}it}) \qquad V \in \mathcal{E}(\mathit{FinalVar}[\,N\,])}{V \in \mathcal{E}(pair_1)} \qquad (5)$$

From C^P_1: $$2 \in \mathcal{E}(t_2) \qquad (6)$$

From C^P_7: $$\frac{\langle(\lambda x'.N), E\rangle \in \mathcal{E}(\mathit{cons\text{-}it}) \qquad V \in \mathcal{E}(t_2)}{V \in \mathcal{E}(x')} \qquad (7)$$

From C^P_8: $$\frac{\langle(\lambda x'.N), E\rangle \in \mathcal{E}(\mathit{cons\text{-}it}) \qquad V \in \mathcal{E}(\mathit{FinalVar}[\,N\,])}{V \in \mathcal{E}(pair_2)} \qquad (8)$$

We can solve these constraints by starting with an empty set environment $\mathcal{E}$ and repeatedly extending $\mathcal{E}$ as required by the constraints until all the constraints are satisfied, as follows:

From (1) :	$\langle(\lambda x.(\textbf{let } (r\ (\textbf{cons } x\ x))\ r)), \emptyset\rangle \in \mathcal{E}(\textit{cons-it})$	(9)
From (3) :	$1 \in \mathcal{E}(t_1)$	(10)
From (4),(9),(10) :	$1 \in \mathcal{E}(x)$	(11)
From (2),(11) :	$(\textbf{cons } 1\ 1) \in \mathcal{E}(r)$	(12)
From (5),(9),(12) :	$(\textbf{cons } 1\ 1) \in \mathcal{E}(pair_1)$	(13)
From (6) :	$2 \in \mathcal{E}(t_2)$	(14)
From (9),(7),(14) :	$2 \in \mathcal{E}(x)$	(15)
From (2),(15) :	$(\textbf{cons } 2\ 2) \in \mathcal{E}(r)$	(16)
From (2),(11),(15) :	$(\textbf{cons } 1\ 2) \in \mathcal{E}(r)$	(17)
From (2),(11),(15) :	$(\textbf{cons } 2\ 1) \in \mathcal{E}(r)$	(18)
From (5),(9),(16) :	$(\textbf{cons } 2\ 2) \in \mathcal{E}(pair_1)$	(19)
From (5),(9),(17) :	$(\textbf{cons } 1\ 2) \in \mathcal{E}(pair_1)$	(20)
From (5),(9),(18) :	$(\textbf{cons } 2\ 1) \in \mathcal{E}(pair_1)$	(21)
From (8),(9),(12) :	$(\textbf{cons } 1\ 1) \in \mathcal{E}(pair_2)$	(22)
From (8),(9),(16) :	$(\textbf{cons } 2\ 2) \in \mathcal{E}(pair_2)$	(23)
From (8),(9),(17) :	$(\textbf{cons } 1\ 2) \in \mathcal{E}(pair_2)$	(24)
From (8),(9),(18) :	$(\textbf{cons } 2\ 1) \in \mathcal{E}(pair_2)$	(25)

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

$$\begin{aligned}
\mathcal{E}(\textit{cons-it}) &= \{\langle(\lambda x.(\textbf{let } (r\ (\textbf{cons } x\ x))\ r)), \emptyset\rangle\} \\
\mathcal{E}(x) &= \{1, 2\} \\
\mathcal{E}(t_1) &= \{1\} \\
\mathcal{E}(t_2) &= \{2\} \\
\mathcal{E}(r) = \mathcal{E}(pair_1) = \mathcal{E}(pair_2) &= \{(\textbf{cons } 1\ 1), (\textbf{cons } 2\ 2), (\textbf{cons } 1\ 2), (\textbf{cons } 2\ 1)\}
\end{aligned}$$

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 the set constraints, $\mathcal{E}$ is the smallest set environment satisfying the set constraints. Of course, $\mathcal{E}$ is not the smallest *valid* set environment. For example, $\mathcal{E}(pair_1)$ contains the value (**cons** 2 2), even though that value would never be associated with $pair_1$ during an execution.

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 for a given program P, then $\mathcal{E}$ must be valid for P.

Assume that $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{12}^P$. To prove that $P \models \mathcal{E}$, we need to show that $\langle P, \emptyset, \epsilon\rangle \longmapsto_{pcek}^{n,m} S$ 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:

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

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

- First, the invariant 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 invariant relation needs to assert that S can only contain values and environments that are *compatible with* P and $\mathcal{E}$. A run-time value V is *compatible with* P and $\mathcal{E}$, written $V \models_P \mathcal{E}$, if each syntactic term occurring in the run-time value V also 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 $E \models_P \mathcal{E}$, if each binding in E occurs in $\mathcal{E}$ and each value in E is compatible with P and $\mathcal{E}$.

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

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

$$\begin{array}{rll} c \models_P \mathcal{E} & \text{if} & c \text{ occurs in } P \\ x \models_P \mathcal{E} & \text{if} & x \text{ occurs in } P \\ \langle(\lambda x.\, M), E\rangle \models_P \mathcal{E} & \text{if} & (\lambda x.\, M) \in P, \mathit{dom}(E) = FV[(\lambda x.\, M)] \text{ and } E \models_P \mathcal{E} \\ (\mathbf{cons}\ V_1\ V_2) \models_P \mathcal{E} & \text{if} & V_1 \models_P \mathcal{E} \text{ and } V_2 \models_P \mathcal{E} \\ \langle \mathbf{ph}\ p\ \circ\rangle \models_P \mathcal{E} & & \\ \langle \mathbf{ph}\ p\ V\rangle \models_P \mathcal{E} & \text{if} & V \models_P \mathcal{E} \end{array}$$

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

  ∎

- The invariant relation must also assert that $\mathcal{E}$ already contains the bindings that could be created during a subsequent (*return*) or (*future-id*) transition to an existing activation record. This requirement is necessary to support the induction hypothesis in the case of a (*return*) or (*future-id*) transition, and is specified in Definition 4.4 below by the clause:

$$\begin{array}{l} \text{if } K = \langle \mathbf{ar}?\ x_1, M_1, E_1\rangle.\cdots.\langle \mathbf{ar}?\ x_n, M_n, E_n\rangle \\ \quad \text{then } \ldots \\ \quad \text{and } \mathcal{E}(\mathit{final\text{-}var}(M_i)) \subseteq \mathcal{E}(x_{i+1}) \text{ for } 1 \le i < n \\ \quad \text{and } \mathcal{E}(\mathit{final\text{-}var}(M)) \subseteq \mathcal{E}(x_1) \text{ if } n > 0 \end{array}$$

- Similarly, the invariant relation must assert that $\mathcal{E}$ already contains the bindings that could be created during a subsequent (*fork*) or (*join*) transition. This requirement is specified in Definition 4.4 below by the clause for **f-let** states and by the clause:

$$\begin{array}{l} \text{if } K = K_1.\langle \mathbf{ar}\dagger\ x, M, E\rangle.K_2 \\ \quad \text{then } \forall p \in \mathit{Ph\text{-}Vars}, \langle \mathbf{ph}\ p\ \circ\rangle \in \mathcal{E}(x) \\ \quad \text{and } \forall V \in \mathcal{E}(\mathit{ResultVar}[\,\langle M, E, K_1\rangle\,]), \langle \mathbf{ph}\ p\ V\rangle \in \mathcal{E}(x) \end{array}$$

  The auxiliary function $\mathit{ResultVar}[\,S\,]$ returns determines the variable whose value will be returned as the result of S.

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 invariant relation must hold for the error state `error`.

The complete invariant relation, which asserts all of the above properties about S, P and $\mathcal{E}$, is defined by induction on the size of states.

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

$$(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2) \models_P \mathcal{E} \quad \Leftrightarrow \quad \left\{ \begin{array}{l} S_1 \models_P \mathcal{E}, S_2 \models_P \mathcal{E} \\ \text{and } \forall V \in \mathcal{E}(\mathit{ResultVar}[\,S_1\,]),\ S_2[p := V] \models_P \mathcal{E} \end{array} \right.$$

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

$$\texttt{error} \models_P \mathcal{E}$$

∎

The invariant relation is a stronger relation than the plain "validates" relation, *i.e.*, if a state 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 $S \models \mathcal{E}$*

Proof: By induction on the structure of S. ∎

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 $C_1^P, \ldots, C_{12}^P$. 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 (*bind-const*), (*apply*) and (*return*) in detail.

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

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

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

- 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:

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

  Obviously $N \in P$. To show that the new environment $E'[x' \leftarrow E(z)]$ is compatible with P and $\mathcal{E}$, we proceed as follows:

  From $S \models_P \mathcal{E}$ we know that $E \models_P \mathcal{E}$. But $E \models_P \mathcal{E}$ implies that $E(y) \models_P \mathcal{E}$, which in turn implies $E' \models_P \mathcal{E}$. Also, $E \models_P \mathcal{E}$ shows that $E(z) \in \mathcal{E}(z)$, which, since $\mathcal{E}$ satisfies C_7^P, implies that $E(z) \in \mathcal{E}(x')$. Finally, $E \models_P \mathcal{E}$ implies that $E(z) \models_P \mathcal{E}$. Combining the facts $E' \models_P \mathcal{E}$, $E(z) \in \mathcal{E}(x')$, and $E(z) \models_P \mathcal{E}$ yields $E'[x' \leftarrow E(z)] \models_P \mathcal{E}$, *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_8^P, 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 $S \models_P \mathcal{E}$ we know that $E' \models_P \mathcal{E}$, and hence $E'(x) \models_P \mathcal{E}$. 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 $E \models_P \mathcal{E}$. Combining the facts $E'(x) \models_P \mathcal{E}$, $E'(x) \in \mathcal{E}(y)$, and $E \models_P \mathcal{E}$ shows that $E[y \leftarrow E'(x)] \models_P \mathcal{E}$, *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.

Main Lemma 4.7 (Soundness of Constraints) *If $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{12}^P$, then $P \models \mathcal{E}$.*

> **Proof:** Suppose $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{12}^P$. Let S be a state such that $\langle P, \emptyset, \epsilon \rangle \longmapsto_{pcek}^{n,m} S$. 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:

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

The least set environment $\bot_{\mathcal{E}}$ in the lattice is the trivial mapping such that $\bot_{\mathcal{E}}(x) = \emptyset$. Each approximation constraint C_i^P can be interpreted as a function, also denoted by C_i^P, on this lattice. Since the presence of each binding in $C_i^P(\mathcal{E})$ depends only on the presence of a finite number of bindings in $\mathcal{E}$, it follows that the functions C_i^P are both monotonic and continuous. We define the function $\mathcal{C}_P$ to be the lub of the functions $C_1^P, \ldots, C_{12}^P$:

$$\begin{aligned} \mathcal{C}_P &: SetEnv_P \longrightarrow SetEnv_P \\ \mathcal{C}_P(\mathcal{E}) &= C_1^P(\mathcal{E}) \sqcup \cdots \sqcup C_{12}^P(\mathcal{E}) \end{aligned}$$

Since the combined function $\mathcal{C}_P$ is also monotonic and continuous, the least fixpoint given by $\bigsqcup_{n \in N} {\mathcal{C}_P}^n(\bot_{\mathcal{E}})$ exists. We define set-based analysis as the function that returns this least fixpoint.

Definition 4.8. (*sba*) The function $sba : \Lambda_a \longrightarrow SetEnv_P$ is given by:

$$sba(P) = \bigsqcup_{n \in N} {\mathcal{C}_P}^n(\bot_{\mathcal{E}})$$

∎

Since the fixpoints of $\mathcal{C}_P$ are exactly the solutions to the set constraints $C_1^P, \ldots, C_{12}^P$, 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.

Main Lemma 4.9 (Correctness of *sba*) $P \models sba(P)$

Combining this lemma with the previous Lemma 3.4 proves that the touch optimization algorithm, parameterized over the results of set-based analysis, always preserves the meaning of programs.

Theorem 4.10 (Correctness of Touch Optimization) *For $P \in \Lambda_a$,*

$$\underline{eval_{pcek}}(\mathcal{T}_{sba(P)}(P)) = eval_{pcek}(P).$$

This theorem shows that our touch optimizing transformation is *provably-correct* with respect to the semantics of programs as specified by the evaluators $eval_{pcek}$ and $\underline{eval_{pcek}}$.

4.4 Performing Set-Based Analysis

Since $sba(P)$ typically maps variables to infinite sets of possible values, we need a finite representation for these infinite sets. To derive such a finite representation, we simply remove the redundant information contained in these infinite sets. To illustrate this idea, consider the source program expression:

$$(\textbf{let}\ (x\ (\textbf{cons}\ y_1\ y_2))\ M)$$

According to C_4^P, $sba(P)(x)$ must contain at least all the cons-cells $(\textbf{cons}\ V_1\ V_2)$ such that $V_1 \in sba(P)(y_1)$ and $V_2 \in sba(P)(y_2)$. A more compact representation for the same information would be to use the program term $(\textbf{cons}\ y_1\ y_2)$ to represent the set $\{(\textbf{cons}\ V_1\ V_2) \mid V_1 \in sba(P)(y_1), V_2 \in sba(P)(y_2)\}$. The possible values of x can then be reconstructed from $(\textbf{cons}\ y_1\ y_2)$ and from the possible values for the variables y_1 and y_2.

In a similar manner, we can use the λ-expression $(\lambda x.\, M)$ to represent all closures of the form $\langle(\lambda x.\, M), E\rangle$, since the possible environments in the closure can easily be reconstructed from the set environment. A systematic inspection of the set constraints produces the following set of *representation values* for a program P:

$$W \in RepValue_P ::= c_P \mid (\lambda x.\, M)_P \mid (\textbf{cons}\ x\ y)_P \mid \langle \textbf{ph}\ x_P\rangle \mid \langle \textbf{ph}\ \circ\rangle$$

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

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

$$\begin{array}{rcl}
c\ in_{\overline{\mathcal{E}}}\ c & & \\
\langle(\lambda x.\, M), E\rangle\ in_{\overline{\mathcal{E}}}\ (\lambda x.\, M) & \Leftrightarrow & \left\{\begin{array}{l} dom(E) = FV[(\lambda x.\, M)] \\ \text{and } \forall y \in dom(E).\ E(y)\ in_{\overline{\mathcal{E}}}\ y \end{array}\right. \\
(\textbf{cons}\ V_1\ V_2)\ in_{\overline{\mathcal{E}}}\ (\textbf{cons}\ y_1\ y_2) & \Leftrightarrow & V_1\ in_{\overline{\mathcal{E}}}\ y_1, V_2\ in_{\overline{\mathcal{E}}}\ y_2 \\
\langle \textbf{ph}\ p\ V\rangle\ in_{\overline{\mathcal{E}}}\ \langle \textbf{ph}\ y\rangle & \Leftrightarrow & V\ in_{\overline{\mathcal{E}}}\ y \\
\langle \textbf{ph}\ p\ \circ\rangle\ in_{\overline{\mathcal{E}}}\ \langle \textbf{ph}\ \circ\rangle & & \\
 & & \\
V\ in_{\overline{\mathcal{E}}}\ x & \Leftrightarrow & \exists W \in \overline{\mathcal{E}}(x) \text{ with } V\ in_{\overline{\mathcal{E}}} W
\end{array}$$

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

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

$$\frac{(\textbf{let } (x\ (\lambda y.\, N))\ M) \in P \qquad \forall z \in FV[(\lambda y.\, N)].\ \overline{\mathcal{E}}(z) \neq \emptyset}{(\lambda y.\, N) \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_3^P})$$

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

$$\frac{(\textbf{let } (x\ (\textbf{car } y))\ M) \in P \qquad W_y \in \overline{\mathcal{E}}(y) \qquad (\textbf{cons } z_1\ z_2) \in \overline{touch}[\overline{\mathcal{E}}, W_y\,] \qquad W \in \overline{\mathcal{E}}(z_1)}{W \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_5^P})$$

$$\frac{(\textbf{let } (x\ (\textbf{cdr } y))\ M) \in P \qquad W_y \in \overline{\mathcal{E}}(y) \qquad (\textbf{cons } z_1\ z_2) \in \overline{touch}[\overline{\mathcal{E}}, W_y\,] \qquad W \in \overline{\mathcal{E}}(z_2)}{W \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_6^P})$$

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

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

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

$$\frac{(\textbf{let } (x\ (\textbf{future } N))\ M) \in P \qquad W \in \overline{\mathcal{E}}(FinalVar[\,N\,])}{W \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_{10}^P})$$

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

$$\frac{(\textbf{let } (x\ (\textbf{future } N))\ M) \in P \qquad \overline{\mathcal{E}}(FinalVar[\,N\,]) \neq \emptyset}{\langle \textbf{ph } FinalVar[\,N\,] \rangle \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_{12}^P})$$

Auxiliary Function:

$$\overline{touch} : RepEnv_P \times RepValue_P \longrightarrow \mathcal{P}(RepValue_P)$$

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

FIGURE 8: Representation Constraints on $\overline{\mathcal{E}}$ with respect to P.

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

$$\begin{array}{rcl} \mathcal{F} : RepEnv_P & \longrightarrow & SetEnv_P \\ \mathcal{F}\left(\overline{\mathcal{E}}\right)(x) & = & \{V \mid V\ in_{\overline{\mathcal{E}}}\ x\} \end{array}$$

Reformulating the set constraints from Figure 7 for representation environments pro-

duces the *representation constraints* $\overline{C_1^P}, \ldots, \overline{C_{12}^P}$ on $\overline{\mathcal{E}}$ with respect to P: see Figure 8. As before, we can interpret each constraint as a monotonic and continuous function on the lattice of representation environments. We define $\overline{sba}(P)$ to be the least fixpoint of the lub of these functions.

Definition 4.11. ($\overline{sba}$) The function $\overline{sba} : \Lambda_a \longrightarrow RepEnv_P$ is given by:

$$\overline{sba}(P) = \bigsqcup_{n \in N} \overline{\mathcal{C}_P}^n(\bot_{\mathcal{E}})$$

where the function $\overline{\mathcal{C}_P} : RepEnv_P \longrightarrow RepEnv_P$ denotes the lub of the representation constraints interpreted as functions, *i.e.*,

$$\overline{\mathcal{C}_P}(\mathcal{E}) = \overline{C_1^P}(\mathcal{E}) \sqcup \cdots \sqcup \overline{C_{12}^P}(\mathcal{E}).$$

∎

The correspondence between set constraints and representation constraints implies that $\overline{sba}(P)$ is a finite representation for $sba(P)$. In particular, our change of representation does not introduce any extra notion of approximation.

Theorem 4.12 (Correctness of Representation) $sba(P) = \mathcal{F}(\overline{sba}(P))$

Proof: See Appendix A. ∎

Since the set of representation values is finite, we can calculate $\overline{sba}(P)$ in an iterative manner. Appendix B contains an $O(|P|^3)$ time algorithm that performs this calculation.

Optimization algorithms can interpret the representation 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 $\overline{sba}(P)$:

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

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

Representation Constraints for a Sample Program In Section 4.1, we derived a set environment $\mathcal{E}$ satisfying the set constraints with respect to the program *cons-it*. We now illustrate how to calculate $\overline{sba}(\textit{cons-it})$.

Instantiating the representation constraints with respect to the program produces the following constraints:

$$\text{From } \overline{C_3^P}: \quad (\lambda x.(\textbf{let } (r\ (\textbf{cons } x\ x))\ r)) \in \overline{\mathcal{E}}(\textit{cons-it}) \tag{1}$$

$$\text{From } \overline{C_4^P}: \quad (\textbf{cons } x\ x) \in \overline{\mathcal{E}}(r) \tag{2}$$

$$\text{From } \overline{C_1^P}: \quad 1 \in \overline{\mathcal{E}}(t_1) \tag{3}$$

$$\text{From } \overline{C_7^P}: \quad \frac{(\lambda x'.N) \in \overline{\mathcal{E}}(\textit{cons-it}) \qquad W \in \overline{\mathcal{E}}(t_1)}{W \in \overline{\mathcal{E}}(x')} \tag{4}$$

$$\text{From } \overline{C_8^P}: \quad \frac{(\lambda x'.N) \in \overline{\mathcal{E}}(\textit{cons-it}) \qquad W \in \overline{\mathcal{E}}(\textit{FinalVar}[N])}{W \in \overline{\mathcal{E}}(\textit{pair}_1)} \tag{5}$$

$$\text{From } \overline{C_1^P}: \quad 2 \in \overline{\mathcal{E}}(t_2) \tag{6}$$

$$\text{From } \overline{C_7^P}: \quad \frac{(\lambda x'.N) \in \overline{\mathcal{E}}(\textit{cons-it}) \qquad W \in \overline{\mathcal{E}}(t_2)}{W \in \overline{\mathcal{E}}(x')} \tag{7}$$

$$\text{From } \overline{C_8^P}: \quad \frac{(\lambda x'.N) \in \overline{\mathcal{E}}(\textit{cons-it}) \qquad W \in \overline{\mathcal{E}}(\textit{FinalVar}[N])}{W \in \overline{\mathcal{E}}(\textit{pair}_2)} \tag{8}$$

We solve these constraints by repeatedly extending $\overline{\mathcal{E}}$ until all the constraints are satisfied, as follows:

$$\text{From } (1): \quad (\lambda x.(\textbf{let } (r\ (\textbf{cons } x\ x))\ r)) \in \overline{\mathcal{E}}(\textit{cons-it}) \tag{9}$$

$$\text{From } (2): \quad (\textbf{cons } x\ x) \in \overline{\mathcal{E}}(r) \tag{10}$$

$$\text{From } (3): \quad 1 \in \overline{\mathcal{E}}(t_1) \tag{11}$$

$$\text{From } (4),(9),(11): \quad 1 \in \overline{\mathcal{E}}(x) \tag{11}$$

$$\text{From } (5),(9),(10): \quad (\textbf{cons } x\ x) \in \overline{\mathcal{E}}(\textit{pair}_1) \tag{12}$$

$$\text{From } (6): \quad 2 \in \overline{\mathcal{E}}(t_2) \tag{13}$$

$$\text{From } (7),(9),(13): \quad 2 \in \overline{\mathcal{E}}(x) \tag{15}$$

$$\text{From } (8),(9),(11): \quad (\textbf{cons } x\ x) \in \overline{\mathcal{E}}(\textit{pair}_1) \tag{16}$$

At this stage we have the representation environment given by:

$$\begin{aligned} \overline{\mathcal{E}}(\textit{cons-it}) &= \{(\lambda x.(\textbf{let } (r\ (\textbf{cons } x\ x))\ r))\} \\ \overline{\mathcal{E}}(x) &= \{1, 2\} \\ \overline{\mathcal{E}}(t_1) &= \{1\} \\ \overline{\mathcal{E}}(t_2) &= \{2\} \\ \overline{\mathcal{E}}(r) = \overline{\mathcal{E}}(\textit{pair}_1) = \overline{\mathcal{E}}(\textit{pair}_2) &= \{(\textbf{cons } x\ x)\} \end{aligned}$$

This representation environment satisfies the representation constraints with respect to the program *cons-it*. A quick examination verifies that $\mathcal{F}\left(\overline{\mathcal{E}}\right)$ is the set environment $\mathcal{E}$ we derived previously for the program.

5 Experimental Results

We extended the Gambit compiler [6, 7], 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

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, finds 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 result.

FIGURE 9: Description of the Benchmark Programs

Benchmark Program	*standard* compiler		*touch optimized* compiler				
	touch operations ($n = 1$)		*touch* operations ($n = 1$)		performance increase over *standard* (%)		
	count(K)	overhead(%)	count(K)	overhead(%)	$n = 1$	$n = 4$	$n = 16$
`fib`	1214	85.0	122	10.2	68	66	58
`queens`	2116	41.2	35	1.5	39	44	39
`rantree`	327	67.5	14	2.6	63	59	37
`mm`	1828	121.0	3	<1	121	79	31
`scan`	1278	126.8	66	4.1	118	77	23
`sum`	525	107.3	33	6.1	95	61	25
`tridiag`	811	110.8	7	<1	109	42	6
`allpairs`	32360	150.4	14	<1	150	66	<1
`abisort`	5751	106.5	9	<1	105	45	32
`mst`	20422	91.4	750	5.3	82	21	<1
`qsort`	253	43.3	78	19.9	20	<1	<1
`poly`	526	65.3	121	16.2	42	14	<1

FIGURE 10: Benchmark Results

as described in the previous sections extended to a sufficiently large subset of functional Scheme.[4] 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 [6] on a GP1000

[4]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.

shared-memory multiprocessor [3]. Figure 9 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 10. The first two columns 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 [6] 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. [15, 22] 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 [20] 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.[5]

At *LFP*'94, Jaganathan and Weeks [17] 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.

Much work has been done on the static analysis of *sequential* programs, including abstract interpretation [4] and 0CFA [26]. Our analysis most closely follows Heintze's work on set-based analysis for the sequential language ML [12], but the extension of this technique

[5] 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.

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 Heintze's 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 [4, 5] and Shivers' 0CFA [26]. The relationship between abstract interpretation and set-based analysis was covered by Heintze [12].

Sequential optimization techniques such as tagging optimization [14] and soft-typing [29, 1] 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 relys on conventional type inference techniques, whereas ours exploits the Scheme view of types as sets directly.

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 Representation

We use in_P as shorthand for $in_{\overline{sba}(P)}$. The following theorem shows that $\overline{sba}(P)$ produces a representation of $sba(P)$.

Theorem 4.12 (Correctness of Representation) $\mathcal{F}\left(\overline{sba}(P)\right) = sba(P)$

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*. Then:

	$V \in \mathcal{F}\left(\overline{sba}(P)\right)(x)$	
implies	$V \; in_P \; x$	by definition of $\mathcal{F}$
implies	$V \in sba(P)(x)$	by Lemma A.1

Also:

	$V \in sba(P)(x)$	
implies	$\exists W \in \overline{sba}(P)(x)$ with $V \; in_P \; W$	by Lemma A.5
implies	$V \; in_P \; x$	
implies	$V \in \mathcal{F}\left(\overline{sba}(P)\right)(x)$	by definition of $\mathcal{F}$

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 $V \; in_P \; x$ (by a formal derivation). Then for some representation value W, $V \; in_P \; W$ and $W \in \overline{sba}(P)(x)$. We proceed by induction on the length of the derivation that $V \; in_P \; x$, and by case analysis on the last representation constraint used in the sub-derivation proving $W \in \overline{sba}(P)(x)$:

$\overline{C_1^P}$: Then $W = c$, $V = c$ and (**let** $(x \; c) \; M$) occurs in P. Hence by C_1^P we have $V \in sba(P)(x)$.

$\overline{C_2^P}$: Then (**let** $(x \; y) \; M$) occurs in P. Also, since $W \in \overline{sba}(P)(y)$ via a sub-derivation, by induction we have $V \in sba(P)(y)$. Constraint C_2^P then shows that $V \in sba(P)(x)$.

$\overline{C_3^P}$: Then (**let** $(x \; (\lambda y.\, N)) \; M$) occurs in P, $W = (\lambda y.\, N)$ and $V = \langle (\lambda y.\, N), E \rangle$, with $dom(E) = FV[(\lambda x.\, M)]$. The derivation for $V \; in_P \; W$ includes, for each $z \in dom(E)$, a sub-derivation showing that $E(z) \; in_P \; z$. 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 $V \in sba(P)(x)$.

$\overline{C_4^P}$: Then (**let** $(x \; (\textbf{cons} \; y_1 \; y_2)) \; M$) occurs in P, $W = (\textbf{cons} \; y_1 \; y_2)$ and $V = (\textbf{cons} \; V_1 \; V_2)$. Since the derivation for $V \; in_P \; W$ includes sub-derivations showing that $V_i \; in_P \; y_i$, by induction we have that $V_i \in sba(P)(y_i)$. Using constraint C_4^P then shows that $V \in sba(P)(x)$.

$\overline{C_5^P}$: We first consider the case where W_y is not a representation placeholder, *i.e.*, where $W_y = (\textbf{cons} \; z_1 \; z_2)$.

Then (**let** $(x \; (\textbf{car} \; y)) \; M$) occurs in P, $(\textbf{cons} \; z_1 \; z_2) \in \overline{sba}(P)(y)$, and $W \in \overline{sba}(P)(z_1)$. Since $V \; in_P \; W$, by induction $V \in sba(P)(z_1)$.

By Lemma A.2, there exists V' such that $V' \; in_P \; (\textbf{cons} \; z_1 \; z_2)$ via a derivation of the same or smaller length than the derivation that $(\textbf{cons} \; z_1 \; z_2) \in$

$\overline{sba}(P)(y)$. Because $V'\ in_P\ (\mathbf{cons}\ z_1\ z_2)$, V' must be of the form $(\mathbf{cons}\ V_1\ V_2)$. Hence $V_2\ in_P\ z_2$ via a shorter derivation, and therefore by induction $V_2 \in sba(P)(z_2)$.

Given that $V \in sba(P)(z_1)$ and $V_2 \in sba(P)(z_2)$, Lemma A.3 proves that $(\mathbf{cons}\ V\ V_2) \in sba(P)(y)$. Using constraint C_5^P gives $V \in sba(P)(x)$.

The case where W_y is a representation 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 $W \in \overline{sba}(P)(x')$ via constraint $\overline{C_7^P}$. Again, we first deal with the case where W_y is not a representation placeholder, *i.e.*, where $W_y = (\lambda x'.\, N)$.

Then $(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M)$ occurs in P and $(\lambda x'.\, N) \in \overline{sba}(P)(y)$ via a sub-derivation. By Lemma A.4, there exists an environment E' such that $\langle (\lambda x'.\, N), E' \rangle \in sba(P)(y)$.

Also, $W \in \overline{sba}(P)(z)$ via a sub-derivation, so by induction we have that $V \in sba(P)(z)$. An application of C_7^P proves that $V \in sba(P)(x')$.

The case where W_y is a representation 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 constraints $\overline{C_9^P}$ and $\overline{C_{10}^P}$ is similar to the analysis for $\overline{C_2^P}$.

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

$\overline{C_{12}^P}$: Then $(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)$ occurs in P, $W = \langle \mathbf{ph}\ FinalVar[\,N\,] \rangle$ and $V = \langle \mathbf{ph}\ p\ V_N \rangle$. Since the derivation for $V\ in_P\ W$ includes a sub-derivation showing that $V_N\ in_P\ FinalVar[\,N\,]$, by induction we have that $V_N \in sba(P)(FinalVar[\,N\,])$. Using constraint C_{12}^P then shows that $V \in sba(P)(x)$.

■

Lemma A.2 (Non-emptiness of representation values) *If $W \in \overline{sba}(P)(x)$ via a derivation of length n, then there exists some V such that $V\ in_P\ W$ via a derivation of length $\leq n$.*

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

$\overline{C_1^P}$: Then $W = c$, and $c\ in_P\ W$.

$\overline{C_2^P}$: Then $W \in \overline{sba}(P)(y)$ via a sub-derivation, and the lemma holds by induction.

$\overline{C_3^P}$: Then $W = (\lambda y.\, M)$ and hence $\langle (\lambda y.\, M), \emptyset \rangle\ in_P\ W$.

$\overline{C_4^P}$: Then $W = (\mathbf{cons}\ y_1\ y_2)$, and $W_i \in \overline{sba}(P)(y_i)$, for some V_i, $i = 1,2$, via sub-derivations. Therefore by induction, there exists $V_i\ in_P\ W_i$, and hence $(\mathbf{cons}\ V_1\ V_2)\ in_P\ W$.

- 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 $\overline{C_{10}^P}$ is similar to the analysis for $\overline{C_2^P}$.

$\overline{C_{11}^P}$: The analysis of the constraint $\overline{C_{11}^P}$ is similar to the analysis for $\overline{C_1^P}$.

$\overline{C_{12}^P}$: Then $W = \langle \mathbf{ph}\ FinalVar[\,N\,]\rangle \in \overline{sba}(P)(x)$, $W_N \in \overline{sba}(P)(FinalVar[\,N\,])$ via a sub-derivation, so by induction there exists $V\ in_P\ W_N$. Hence we have that $\langle \mathbf{ph}\ p\ V\rangle\ in_{\overline{\mathcal{E}}} \langle \mathbf{ph}\ FinalVar[\,N\,]\rangle$.

∎

Lemma A.3 *If* $V_1 \in sba(P)(z_1)$, $V_2 \in sba(P)(z_2)$ *and* $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(x)$ *then* $(\mathbf{cons}\ V_1\ V_2) \in sba(P)(x)$.

Proof: The proof is by induction on the length of the derivation that $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(x)$. We proceed by case analysis on the last constraint used in the derivation:

$\overline{C_2^P}$: Suppose $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(x)$ via constraint $\overline{C_2^P}$. Then $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(y)$ via a sub-derivation, hence $(\mathbf{cons}\ V_1\ V_2) \in sba(P)(y)$ by induction, and therefore $(\mathbf{cons}\ V_1\ V_2) \in sba(P)(x)$ via C_2^P.

$\overline{C_4^P}$: Suppose $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(x)$ via constraint $\overline{C_4^P}$. Then $(\mathbf{cons}\ V_1\ V_2) \in sba(P)(x)$ via C_4^P.

- The cases for the constraints $\overline{C_5^P}$, $\overline{C_6^P}$, $\overline{C_7^P}$, $\overline{C_8^P}$, $\overline{C_9^P}$ and $\overline{C_{10}^P}$ follow by similar reasoning to the case for $\overline{C_2^P}$.
- The last constraint in the derivation of $(\mathbf{cons}\ z_1\ z_2) \in \overline{sba}(P)(x)$ cannot be one of $\overline{C_1^P}$, $\overline{C_3^P}$, $\overline{C_{11}^P}$ or $\overline{C_{12}^P}$.

∎

Lemma A.4 *If* $(\lambda x'.\,N) \in \overline{sba}(P)(x)$ *then there exists some environment* E' *such that* $\langle(\lambda x'.\,N), E'\rangle \in sba(P)(x)$.

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

$\overline{C_2^P}$: Suppose $(\lambda x'.\,N) \in \overline{sba}(P)(x)$ via constraint $\overline{C_2^P}$. Then $(\lambda x'.\,N) \in \overline{sba}(P)(y)$ via a sub-derivation, hence $\langle(\lambda x'.\,N), E'\rangle \in sba(P)(y)$ by induction, and therefore $\langle(\lambda x'.\,N), E'\rangle \in sba(P)(x)$ via C_2^P.

$\overline{C_3^P}$: Suppose $(\lambda x'.\,N) \in \overline{sba}(P)(x)$ via constraint $\overline{C_3^P}$. Let the free variables of $(\lambda x'.\,N)$ be called $z_1, \ldots, z_n$. For each z_i there exists $W_i \in \overline{sba}(P)(z_i)$. Hence by Lemma A.2, there exists $V_i \in sba(P)(z_i)$. Let E' be the environment $\langle z_1 \mapsto V_1, \ldots, z_n \mapsto V_n\rangle$, and we have by C_3^P that $\langle(\lambda x'.\,N), E'\rangle \in sba(P)(x)$.

- The cases for the constraints $\overline{C_5^P}$, $\overline{C_6^P}$, $\overline{C_7^P}$, $\overline{C_8^P}$, $\overline{C_9^P}$ and $\overline{C_{10}^P}$ follow by similar reasoning to the case for $\overline{C_2^P}$.
- The last constraint in the derivation of $(\lambda x'.\, N) \in \overline{sba}(P)(x)$ cannot be one of $\overline{C_1^P}$, $\overline{C_4^P}$, $\overline{C_{11}^P}$ or $\overline{C_{12}^P}$.

∎

Lemma A.5 (*sba* includes $\overline{sba}$:) *$V \in sba(P)(x)$ implies there exists $W \in \overline{sba}(P)(x)$ with $V\ in_P\ W$.*

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

C_1^P: We take $W = c$, and the lemma holds.

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

C_3^P: In this case, $V = \langle(\lambda y.\, N), E\rangle$, where $dom(E) \subseteq FV[(\lambda x.\, M)]$, and for each $z_i \in dom(E)$ we have $E(z_i) \in sba(P)(z_i)$ via sub-derivations. Therefore, there exist W_i such that $W_i \in \overline{sba}(P)(z_i)$ and $E(z_i)\ in_P\ W_i$. Now take $W = (\lambda y.\, N)$, then $V\ in_P\ W$ and by $\overline{C_3^P}$, $W \in \overline{sba}(P)(x)$.

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

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 sub-derivation. By induction there exists W_y such that $W_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ W_y$. Hence $W_y = (\mathbf{cons}\ z_1\ z_2)$, $W_1 \in \overline{sba}(P)(z_1)$ and $V_1\ in_P\ W_1$. From $\overline{C_5^P}$ we can prove $W_1 \in \overline{sba}(P)(x)$, 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 sub-derivation. By induction there exists W_y such that $W_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ W_y$. Hence $W = (\lambda x'.\, N)$. Also, $V \in sba(P)(z)$ via a sub-derivation, so by induction there exists W such that $W \in \overline{sba}(P)(y)$ and $V\ in_P\ W$. Hence, by $\overline{C_7^P}$ we have $W \in \overline{sba}(P)(x')$.

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 sub-derivation. By induction there exists W_y such that $W_y \in \overline{sba}(P)(y)$ and $V_y\ in_P\ W_y$. Hence

$W_y = (\lambda x'.\, N)$. Also, $V \in sba(P)(FinalVar[\,N\,])$ via a sub-derivation, so by induction there exists W such that $W \in \overline{sba}(P)(FinalVar[\,N\,])$ and $V\ in_P\ W$. Hence, by $\overline{C_8^P}$ we have $W \in \overline{sba}(P)(x)$.

C_9^P, C_{10}^P: The analysis for the constraints C_9^P and C_{10}^P is similar to the analysis for C_2^P.

C_{12}^P: In this case $V = \langle \mathbf{ph}\ p\ V_N \rangle \in sba(P)(x)$ via C_{12}^P, $V_N \in sba(P)(FinalVar[\,N\,])$ via a sub-derivation, so by induction there exists $W_N \in \overline{sba}(P)(FinalVar[\,N\,])$ with $V_N\ in_P\ W_N$. By $\overline{C_{12}^P}$ 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 $O(|P|^3)$ time set-based analysis algorithm[6] for the intermediate language Λ_a is included in Figures 11 through 13. The algorithm is written in Scheme extended with a special form **match** for pattern matching [28].

The function *SBA* traverses an expression to derive the representation constraints for that expression. Certain representation constraints cannot be satisfied immediately. For example, the constraint $\overline{C_2^P}$ requires that all representation values in $\overline{\mathcal{E}}(y)$ must be in $\overline{\mathcal{E}}(x)$, but the algorithm may not yet know all the representation values that may be added to $\overline{\mathcal{E}}(y)$. Therefore, the function *SBA* associate a constraint '(propagate-to $,x$) with the variable y. This constraint then gets applied (via the function *interpret-constraint*) to each representation value that is added to $\overline{\mathcal{E}}(y)$, and propagates that representation value to $\overline{\mathcal{E}}(x)$.

The functions *new-repvalue!* and *new-constraint!* associate representation values and constraints, respectively, with variables. To ensure that the constraints are satisfied, the function *interpret-constraint* is called for each constraint and representation value associated with a given variable. This function ensures that the constraint is satisfied, by creating new representation values or new constraints as necessary.

The $O(|P|^3)$ time bound can be verified as follows: We assume that the functions *in-*$\overline{\mathcal{E}}$*?*, *add-to-*$\overline{\mathcal{E}}$*!*, *in-*$\mathcal{C}$*?* and *add-to-*$\mathcal{C}$*!* operate in constant time, and that the functions *foreach-in-*$\overline{\mathcal{E}}$ and *foreach-in-*$\mathcal{C}$ operate in time linear in the number of elements in the appropriate set.

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 representation values, and the number of constraints are all $O(|P|)$. Therefore the test conditions of the two **unless** expressions succeed at most $O(|P|^2)$ times

[6] For clarity of presentation, the algorithm implements simpler versions of the representation constraints $\overline{C_3^P}$, $\overline{C_4^P}$ and $\overline{C_{12}^P}$ that do not include that non-emptyness conditions. We believe the extra degree of approximation introduced by this simplification is minor, and extending the algorithm to include these conditions is not difficult.

```
;; inputs:   a Λa term M
;; outputs:  calculates FinalVar[M]
;; effects:  derives the representation 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-repvalue! x c)]
           [(? variable? y) (new-constraint! y `(propagate-to ,x))]
           [`(cons ,y1 ,y2) (new-repvalue! x `(cons ,y1 ,y2))]
           [`(lambda ,y ,N)
            (let ([finalvar-N (SBA N)])
              (new-repvalue! x `(lambda ,y ,N ,finalvar-N)))]
           [`(future ,N)
            (let ([finalvar-N (SBA N)])
              (new-constraint! finalvar-N `(propagate-to ,x))
              (new-repvalue! x `(ph ,finalvar-N))
              (new-repvalue! x '(ph-circ)))]
           [`(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! finalvar-M1 `(propagate-to ,x))
              (new-constraint! finalvar-M2 `(propagate-to ,x)))]
           [`(apply ,y ,z) (new-constraint! y `(application ,x ,z))])
         (SBA body)])))
```

FIGURE 11: The Set-Based Analysis Algorithm

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] AIKEN, A., AND WIMMERS, E. L. Soft typing with conditional types. In *POPL* (1994), pp. 163–173.

[2] 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) of *SIGPLAN Notices*, pp. 55–59.

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

;; 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 (repvalue) (interpret-constraint constraint repvalue))))))

;; inputs:   a constraint and a representation value
;; outputs:  void
;; effects:  ensures that the constraint is satisfied, given that the constraint
;;           and the representation value are associated with the same variable
(define interpret-constraint
  (lambda (constraint repvalue)
    (match (cons constraint repvalue)
        [`((propagate-to ,x) . ,v) (new-repvalue! 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))]
        [_ (void)])))
```

FIGURE 12: The Set-Based Analysis Algorithm: Auxiliary Functions 1

[3] BBN ADVANCED COMPUTERS, INC., CAMBRIDGE, MA. *Inside the GP1000.* 1989.

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

[5] 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.

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

```
;; The following three functions maintain a mapping ℰ̄ from program
;; variables to sets of representation values

;; inputs:   a variable and a representation value
;; outputs:  returns true if ℰ̄(var) contains the representation value
;; effects:  none
(define in-ℰ̄? (lambda (var repvalue) ...))

;; inputs:   a variable and a representation value
;; outputs:  void
;; effects:  adds repvalue to ℰ̄(var)
(define add-to-ℰ̄! (lambda (var repvalue) ...))

;; inputs:   a variable and a function
;; outputs:  none
;; effects:  applies fn to each representation value in ℰ(var)
(define foreach-in-ℰ̄ (lambda (var fn) ...))

;; The following three functions maintain a mapping 𝒞 from program
;; variables to sets of constraints

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

;; inputs:   a variable and a constraint
;; outputs:  void
;; effects:  adds constraint to 𝒞(var)
(define add-to-𝒞! (lambda (var constraint) ...))

;; inputs:   a variable and a function
;; outputs:  none
;; effects:  applies fn to each constraint in ℰ(var)
(define foreach-in-𝒞 (lambda (var fn) ...))
```

FIGURE 13: The Set-Based Analysis Algorithm: Auxiliary Functions 2

[7] FEELEY, M., AND MILLER, J. S. A parallel virtual machine for efficient scheme compilation. In *LFP* (1990).

[8] FLANAGAN, C., AND FELLEISEN, M. The semantics of Future. Rice University Computer Science TR94-238.

[9] FLANAGAN, C., AND FELLEISEN, M. The semantics of future and its use in program optimizations. In *POPL* (Jan. 1995).

[10] FLANAGAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. The essence of compiling with continuations. In *PLDI* (1993), pp. 237–247.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] Wright, A. K., and Duba, B. F. Pattern matching for Scheme. Unpublished manuscript, 1993.

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