The Semantics of Future and Its Use in Program Optimization

Cormac Flanagan* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, Texas

Abstract

The **future** annotations of MultiLisp provide a simple method for taming the implicit parallelism of functional programs. Past research concerning **future**s has focused on implementation issues. In this paper, we present a series of operational semantics for an idealized functional language with **future**s with varying degrees of intensionality. We develop a set-based analysis algorithm from the most intensional semantics, and use that algorithm to perform *touch* optimization on programs. Experiments with the Gambit compiler indicates that this optimization substantially reduces program execution times.

1 Implicit Parallelism via Annotations

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 sub-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 [1, 12] and its Scheme successors 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 with 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. When a program operation requires specific knowledge about the value of some sub-computation but finds a placeholder in its place, the run-time system performs a *touch* operation, which synchronizes the appropriate parallel threads.

Past research on **future**s has almost exclusively concentrated on the efficient implementation of the underlying task creation mechanism [6, 17, 23, 25, 26] and on the extension of the concept to first-class continuations [19, 27]. In contrast, the driving force behind our effort is the desire to develop a semantic framework and semantically well-founded optimizations for languages with **future**. The specific example we choose to consider is the development of an algorithm for removing provably-redundant *touch* operations from programs. Our primary results are a series of semantics for a functional language with **future**s and a program analysis. The first semantics defines **future** to be a semantically-transparent annotation. The second one validates that a **future** expression interpreted as process creation is correct. The last one is a low-level refinement, which explicates just enough information to permit the derivation of a set-based program analysis [14]. The secondary result is a *touch* optimization algorithm (based on the analysis) with its correctness proof. The algorithm was added to the Gambit Scheme compiler [6] and produced significant speedups on a standard set of benchmarks.

The presentation of our results proceeds as follows. The second section introduces an idealized functional language with **future**s, together with its definitional, sequential semantics that interprets **future**s as no-ops. The third section presents an equivalent parallel semantics for **future**s and the fourth section contains the low-level refinement of that semantics. The fifth 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 sixth section. The seventh section presents experimental results demonstrating the effectiveness of this optimization. Section eight discusses related work. For more details, we refer the interested reader to two technical reports on this work [9, 10].

*Supported in part by NSF grant CCR 91-22518, Texas ATP grant 91-003604014 and a sabbatical at Carnegie Mellon University.

To appear in the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 23-25, 1995.

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

Figure 1: The A-normalized Language Λ_a

2 A Functional Language with Futures

2.1 Syntax Given the goal of developing a semantics that is useful for proving the soundness of optimizations, we develop the definitional semantics for **futures** for an intermediate representation of an idealized functional language. Specifically, we use the subset of A-normal forms [11] of an extended λ-calculus-like language that includes conditionals and a **future** construct: see Figure 1. The language also includes primitives for list manipulation, which serve to illustrate the treatment of primitive operations, and an unspecified set of basic constants (numbers, booleans).

The key property of terms in A-normal form is that each intermediate value is explicitly named and that the order of execution follows the lexical nesting of **let**-expressions. The use of A-normal forms facilitates the compile-time analysis of programs [29], and it simplifies the definition of abstract machines [11].

We work with the usual conventions and terminology of the lambda calculus when discussing syntactic issues. In particular, the substitution operation $M[x \leftarrow V]$ replaces all free occurrences of x within M by V, X^0 denotes the set of closed terms of type X (terms, values), and $M \in P$ denotes that the term M occurs in the program P. Also, 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; and $f : A \longrightarrow_p B$ denotes that f is a partial function from A to B.

2.2 Definitional Semantics The semantics of Λ_a is a function from closed programs to results. A result is either an answer, which is a closed value with all λ-expressions replaced by `procedure`, or `error`, indicating that some program operation was misapplied, or $\perp$, if the program does not terminate. We specify the definitional semantics of the language using a sequential abstract machine called the C-machine (see Figure 2), whose states are either closed terms over the run-time language Λ_c or else the special state `error`, and whose deterministic transition rules are the typical leftmost-outermost reductions of the λ-calculus [8]. Each transition rule also specifies the error semantics of a particular class of expressions. For example, the transition rule for **car** defines that if the argument to **car** is a pair, then the transition rule extracts the first element of the pair. If the argument is not a pair, then the transition rule produces the state `error`.

The rule for **future** pretends that **future** is the identity operation. It demands that the body of a **future** expression is first reduced to a value, and then replaces the name for the **future** expression with this value.

The definition of the transition function relies on the notion of *evaluation contexts*. An evaluation context $\mathcal{E}$ is a term with a hole [ ] in place of the next sub-term to be evaluated; *e.g.*, in the term (**let** $(x\ M_1)\ M_2$), the next sub-term to be evaluated is M_1, and thus the definition of evaluation contexts includes (**let** $(x\ \mathcal{E})\ M$).

A machine state is a *final state* if it is a value or the state `error`. No transitions are possible from a final state, and for any state that is not a final state, there is a unique transition step from that state to its successor state. This implies that the relation $eval_c$ is a total function: Either the transition sequence for a program P terminates in a final state, in which case $eval_c(P)$ is an answer or `error`, or else the transition sequence is infinite, in which case $eval_c(P) = \perp$. Since the evaluator $eval_c$ obviously agrees with the sequential semantics of the underlying functional language, **future** is clearly nothing but an annotation.

3 A Parallel Operational Semantics

The sequential C-machine defines **future** as an annotation, and ignores the *intension* of **future** as an advisory instruction concerning parallel evaluation. To understand this intensional aspect of **future**, we need a semantics of **future** that models the concurrent evaluation of **future** expressions.

3.1 The $P(C)$-machine The state space of the $P(C)$-machine is defined in the first part of Figure 3. The set of $P(C)$ values includes the values of the sequential C-machine (constants, variables, closures and pairs), which we refer to as *proper* values. To model futures, the $P(C)$-machine also includes a new class of values called *placeholder variables*. A placeholder variable p represents the result of a computation that is in progress. Once the computation terminates, all occurrences of the placeholder are replaced by the value returned by the computation.

Each C-machine state represents a single thread of control. To model parallel threads, the $P(C)$-machine includes additional states of the form (**f-let** $(p\ S_1)\ S_2$). The *primary* sub-state S_1 is initially the body of a **future** expression, and the *secondary* sub-state S_2 is initially the evaluation context surrounding the **future** expression. The placeholder p represents the result of S_1 in S_2. The usual conventions for binding constructs like λ and **let** apply to **f-let**. The function FP returns the set of free placeholders in a state. The evaluation of S_1 is *mandatory*, since it is guaranteed to contribute to the completion of the program. The evaluation of S_2 is *speculative*, since such work may not be required for the termination of the program. In particular, if S_1 raises an error signal, then the evaluator discards the state

Evaluator:

$$
\begin{array}{rcl}
\mathit{eval}_c : \Lambda_a^0 & \longrightarrow & \mathit{Answers} \cup \{\texttt{error}, \bot\} \\
\mathit{eval}_c(P) & = & \begin{cases} \mathit{unload}_c[V] & \text{if } P \longmapsto_c^* V \\ \texttt{error} & \text{if } P \longmapsto_c^* \texttt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists M_i \in \mathit{State}_c \text{ such that } P = M_0 \text{ and } M_i \longmapsto_c M_{i+1} \end{cases}
\end{array}
$$

Data Specifications:

$$
\begin{array}{rclll}
S & \in & \mathit{State}_c & ::= & M \mid \texttt{error} \quad \text{(States)} \\
M & \in & \Lambda_c & ::= & V \quad \text{(Run-time Language)} \\
& & & & \mid (\textbf{let}\ (x\ V)\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{future}\ M))\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{car}\ V))\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{cdr}\ V))\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{if}\ V\ M\ M))\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{apply}\ V\ V))\ M) \\
& & & & \mid (\textbf{let}\ (x\ M)\ M) \\
V & \in & \mathit{Value}_c & ::= & c \mid x \mid (\lambda x.M) \mid (\textbf{cons}\ V\ V) \quad \text{(Run-time Values)} \\
A & \in & \mathit{Answers} & ::= & c \mid \texttt{procedure} \mid (\textbf{cons}\ A\ A) \quad \text{(Answers)} \\
\mathcal{E} & \in & \mathit{EvalCtxt} & ::= & [\,] \quad \text{(Evaluation Contexts)} \\
& & & & \mid (\textbf{let}\ (x\ \mathcal{E})\ M) \\
& & & & \mid (\textbf{let}\ (x\ (\textbf{future}\ \mathcal{E}))\ M)
\end{array}
$$

Unload Function:

$$
\begin{array}{rcl}
\mathit{unload}_c : \mathit{Value}_c^0 & \longrightarrow & \mathit{Answers} \\
\mathit{unload}_c[c] & = & c \\
\mathit{unload}_c[(\lambda x.M)] & = & \texttt{procedure} \\
\mathit{unload}_c[(\textbf{cons}\ V_1\ V_2)] & = & (\textbf{cons}\ A_1\ A_2) \\
& & A_i = \mathit{unload}_c[V_i]
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
\mathcal{E}[\ (\textbf{let}\ (x\ V)\ M)\ ] & \longmapsto_c & \mathcal{E}[\ M[x \leftarrow V]\ ] & (\mathit{bind}) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{future}\ V))\ M)\ ] & \longmapsto_c & \mathcal{E}[\ M[x \leftarrow V]\ ] & (\mathit{future\text{-}id}) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{car}\ V))\ M)\ ] & \longmapsto_c & \begin{cases} \mathcal{E}[\ M[x \leftarrow V_1]\ ] & \text{if } V = (\textbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } V \neq (\textbf{cons}\ V_1\ V_2) \end{cases} & (\mathit{car}) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{cdr}\ V))\ M)\ ] & \longmapsto_c & \text{analogous to } (\mathit{car}) & (\mathit{cdr}) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{if}\ V\ M_1\ M_2))\ M)\ ] & \longmapsto_c & \begin{cases} \mathcal{E}[\ (\textbf{let}\ (x\ M_1)\ M)\ ] & \text{if } V \neq \texttt{false} \\ \mathcal{E}[\ (\textbf{let}\ (x\ M_2)\ M)\ ] & \text{if } V = \texttt{false} \end{cases} & (\mathit{if}) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{apply}\ V_1\ V_2))\ M)\ ] & \longmapsto_c & \begin{cases} \mathcal{E}[\ (\textbf{let}\ (x\ N[y \leftarrow V_2])\ M)\ ] & \text{if } V_1 = (\lambda y.N) \\ \texttt{error} & \text{if } V_1 \neq (\lambda y.N) \end{cases} & (\mathit{apply})
\end{array}
$$

Figure 2: The sequential C-machine

S_2, and any effort invested in the evaluation of S_2 is wasted. The distinction between mandatory and speculative steps is crucial for ensuring a sound definition of an evaluator and is incorporated into the definition of the transition relation.

Transition Rules We specify the transition relation of the $P(C)$-machine as a quadruple. If $S \longmapsto_{pc}^{n,m} S'$ holds, then the index n is the number of steps involved in the transition from S to S', and the index $m \leq n$ is the number of these steps that are *mandatory*.

The transition rules (*bind*), (*future-id*), (*car*), (*cdr*), (*if*) and (*apply*) are simply the rules of the C-machine, appropriately modified to allow for placeholder variables. An application of one of these rules counts as a mandatory step.

The transition rule (*fork*) initiates parallel evaluation. This rule may be applied whenever the current term includes a **future** expression within an evaluation context, *i.e.*:

$$\mathcal{E}(\textbf{let}\ (x\ (\textbf{future}\ N))\ M)$$

The **future** annotation allows the expression N to be evaluated in parallel with the enclosing context. The machine creates a new placeholder p to represent the result of N, and initiates parallel evaluation of N and $\mathcal{E}[\ (\textbf{let}\ (x\ p)\ M)\ ]$.

The transition rule (*parallel*) permits concurrent evaluation of both sub-states of a parallel state.

The transition rules (*join*) and (*join-error*) merge distinct threads of evaluation. When the primary sub-state S_1 of a parallel state ($\textbf{f-let}\ (p\ S_1)\ S_2$) returns a value, then the rule (*join*) replaces all occurrences of the placeholder p within S_2 by that value. If the primary sub-state S_1 evaluates to `error`, then the rule (*join-error*) discards the secondary sub-state S_2 and returns `error` as the result of the parallel state.

The transition rule (*lift*) restructures nested parallel states, and thus exposes additional parallelism in certain cases. Consider ($\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ S_1)\ V))\ S_3$). The rule (*lift*) allows the value V to be returned to the sub-state S_3 (via a subsequent (*join*) transition), without having to wait on the termination of S_1.[1]

The rules (*reflexive*) and (*transitive*) close the relation under reflexivity and transitivity. We write $S \longmapsto_{pc}^{*} S'$ if $S \longmapsto_{pc}^{n,m} S'$ for some $n, m \in \mathbf{N}$.

Indeterminism Unlike the C-machine, in which each state has a unique successor state, the transition rules of the $P(C)$-machine denote a true relation. In particular, the definition does not specify when the transition rule (*fork*) applies. For example, given the state $\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{future}\ N))\ M)\ ]$, the machine may proceed either by evaluating N sequentially, or by creating a new task via a (*fork*) transition. An implementation may

[1] A second reason for the inclusion of this rule is that it is necessary for an elegant proof of the consistency of the machine using a modified form of the diamond lemma of the lambda calculus.

Evaluator:

$$
\begin{array}{rcl}
eval_{pc} : \Lambda_a^0 & \longrightarrow & Answers \cup \{\mathtt{error}, \bot\} \\
eval_{pc}(P) & = & \begin{cases} unload_c[V] & \text{if } P \longmapsto^*_{pc} V \\ \mathtt{error} & \text{if } P \longmapsto^*_{pc} \mathtt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists S_i \in State_{pc}, n_i, m_i \in \mathbf{N} \text{ with } m_i > 0, P = S_0 \text{ and } S_i \longmapsto^{n_i,m_i}_{pc} S_{i+1} \end{cases}
\end{array}
$$

Data Specifications:

$$
\begin{array}{rclcll}
S & \in & State_{pc} & ::= & M \mid \mathtt{error} \mid (\textbf{f-let}\ (p\ S)\ S) & \text{(States)} \\
M & \in & \Lambda_{pc} & ::= & V \mid (\textbf{let}\ (x\ V)\ M) \mid \ldots & \text{(As for } \Lambda_c\text{)} \\
V & \in & Value_{pc} & ::= & c \mid x \mid (\lambda x. M) \mid (\textbf{cons}\ V\ V) \mid p & \text{(Run-time Values)} \\
p & \in & Ph\text{-}Vars & ::= & \{p_1, p_2, p_3, \ldots\} & \text{(Placeholders Variables)}
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
\mathcal{E}[\ (\textbf{let}\ (x\ V)\ M)\ ] & \longmapsto^{1,1}_{pc} & \mathcal{E}[\ M[x \leftarrow V]\ ] & (bind) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{future}\ V))\ M)\ ] & \longmapsto^{1,1}_{pc} & \mathcal{E}[\ M[x \leftarrow V]\ ] & (future\text{-}id) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{car}\ V))\ M)\ ] & \longmapsto^{1,1}_{pc} & \begin{cases} \mathcal{E}[\ M[x \leftarrow V_1]\ ] & \text{if } V = (\textbf{cons}\ V_1\ V_2) \\ \mathtt{error} & \text{if } V \neq (\textbf{cons}\ V_1\ V_2), V \neq p \end{cases} & (car) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{cdr}\ V))\ M)\ ] & \longmapsto^{1,1}_{pc} & \text{analogous to } (car) & (cdr) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{if}\ V\ M_1\ M_2))\ M)\ ] & \longmapsto^{1,1}_{pc} & \begin{cases} \mathcal{E}[\ (\textbf{let}\ (x\ M_1)\ M)\ ] & \text{if } V \neq \mathtt{false}, V \neq p \\ \mathcal{E}[\ (\textbf{let}\ (x\ M_2)\ M)\ ] & \text{if } V = \mathtt{false} \end{cases} & (if) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{apply}\ V_1\ V_2))\ M)\ ] & \longmapsto^{1,1}_{pc} & \begin{cases} \mathcal{E}[\ (\textbf{let}\ (x\ N[y \leftarrow V_2])\ M)\ ] & \text{if } V_1 = (\lambda y. N) \\ \mathtt{error} & \text{if } V_1 \neq (\lambda y. N), V_1 \neq p \end{cases} & (apply) \\
\mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{future}\ N))\ M)\ ] & \longmapsto^{1,0}_{pc} & (\textbf{f-let}\ (p\ N)\ \mathcal{E}[\ M[x \leftarrow p]\ ]) \quad p \notin FP(\mathcal{E}) \cup FP(M) & (fork) \\
(\textbf{f-let}\ (p\ V)\ S) & \longmapsto^{1,1}_{pc} & S[p \leftarrow V] & (join) \\
(\textbf{f-let}\ (p\ \mathtt{error})\ S) & \longmapsto^{1,1}_{pc} & \mathtt{error} & (join\text{-}error) \\
(\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ S_1)\ S_2))\ S_2) & \longmapsto^{1,1}_{pc} & (\textbf{f-let}\ (p_1\ S_1)\ (\textbf{f-let}\ (p_2\ S_2)\ S_3)) \quad p_1 \notin FP(S_3) & (lift) \\
(\textbf{f-let}\ (p\ S_1)\ S_2) & \longmapsto^{n,b}_{pc} & (\textbf{f-let}\ (p\ S_1')\ S_2') \quad \text{if } S_1 \longmapsto^{a,b}_{pc} S_1', S_2 \longmapsto^{c,d}_{pc} S_2', n = a + c & (parallel) \\
S & \longmapsto^{0,0}_{pc} & S & (reflexive) \\
S & \longmapsto^{n,m}_{pc} & S'' \quad \text{if } S \longmapsto^{a,b}_{pc} S', S' \longmapsto^{c,d}_{pc} S'', n = a + c, m = b + d, n > 0 & (transitive)
\end{array}
$$

Figure 3: The parallel $P(C)$-machine

consequently choose to ignore **future** expressions (along the lines of the C-machine), which yields a sequential execution, to execute *fork* as early as possible, which yields an *eager* task creation mechanism [23, 31], or to choose some strategy in between the extremes, which yields *lazy task creation* [6, 26]

A second source of indeterminism is the transition rule (*parallel*). This rule does not specify the number of steps that parallel sub-states must perform before they synchronize. An implementation of the machine can use almost any scheduling strategy for allocating processors to tasks, as long as it regularly schedules the mandatory thread.

Evaluation In general, the evaluation of a program can proceed in many different directions. Some of these transition sequences may be infinite, even if the program terminates according to the sequential semantics. Consider:

$$P \ = \ (\textbf{let}\ (x\ (\textbf{future}\ \mathtt{error}))\ \Omega)$$

where Ω is some diverging sequential term, *i.e.*, $\Omega \longmapsto^{1,1}_{pc} \Omega$. The sequential evaluator never executes Ω because P's result is $\mathtt{error}$. In contrast, P admits the following infinite parallel transition sequence:

$$
\begin{array}{rlr}
P & \longmapsto^{1,0}_{pc} \ (\textbf{f-let}\ (p\ \mathtt{error})\ \Omega) & \text{via } (fork) \\
 & \longmapsto^{1,0}_{pc} \ (\textbf{f-let}\ (p\ \mathtt{error})\ \Omega) & \text{since } \Omega \longmapsto^{1,1}_{pc} \Omega \\
 & \longmapsto^{1,0}_{pc} \ \ldots &
\end{array}
$$

This "evaluation" diverges because it exclusively consists of speculative transition steps and does not include any mandatory transition steps that contribute to the sequential evaluation of the program. The evaluator for the $P(C)$-machine excludes these *excessively speculative* transition sequences, and only considers transition sequences that regularly includes mandatory steps. For a terminating transition sequence, the number of speculative steps performed is implicitly bounded. For non-terminating sequences, the definition of the evaluator explicitly requires the performance of mandatory transition steps on a regular basis. This constraint implies that an implementation of the machine must keep track of the mandatory thread and must ensure that this mandatory thread is regularly executed.

3.2 Correctness

The *observable* behavior of the $P(C)$-machine on a given program is deterministic, despite its indeterminate *internal* behavior.

Theorem 3.1 *$eval_{pc}$ is a function.*

We prove this consistency using a modified form of the traditional diamond lemma. The modified diamond lemma states that if we reduce an initial state S_1 by two alternative transitions, producing respectively states S_2 and S_3, then there is some state S_4 that is reachable from both S_2 and S_3. Furthermore, the number of *mandatory* steps on the transition from S_1 to S_4 via S_2 is bounded by the *total* number of steps on the transition from S_1 to S_4 via S_3, and vice-versa. This bound is necessary to prove that all transition sequences for a given program exhibit the same termination behavior.

Since each sequential transition rule of the $P(C)$-machine subsumes the corresponding transition rule of the C-machine, every transition of the C-machine is also a transition of the $P(C)$-machine, which implies that the evaluators are equivalent.

Theorem 3.2 $eval_{pc} = eval_c$

Put differently, the $P(C)$-machine is a correct implementation of the C-machine in that both define the same semantics for the source language. Hence, the interpretation of **future** as a task creation construct, with implicit task coordination, is entirely consistent with the definitional semantics of **future** as an annotation.

4 A Low-Level Operational Semantics

Since optimizations heavily rely on static information about the values that variables can assume, the $P(C)$-machine is ill-suited for correctness proofs of appropriate analysis algorithms.[2] On one hand, the states of the $P(C)$-machine contain no binding information relating program variables and values. Instead, the machine relies on substitution for making progress. On the other hand, the representation of run-time values and other objects in the $P(C)$-machine is too coarse. For example, it does not permit a detailed view of the synchronization operations that are required for coordinating **futures**. To address these problems, we refine the $P(C)$-machine to the $P(CEK)$-machine (see Figure 4) using standard techniques [8, 11].

4.1 The $P(CEK)$-machine An evaluation context, which represents the control stack, is now represented as a sequence of activation records (which are similar to closures). A tagged activation record ($\langle \mathbf{ar}\dagger\ x, M, E\rangle$) represents a point where the continuation can be split into separate tasks (cmp. *fork*). The substitution operation is replaced by an *environment* in the usual manner. An environment E is a mapping from variables to run-time values. The empty environment is denoted by $\emptyset$, and the operation $E[x \leftarrow V]$ extends the environment E to map the variable x to the value V.

During the course of the refinement, we also replace each placeholder p with an explicit *undetermined placeholder object* $\langle \mathbf{ph}\ p\ \circ\rangle$. The symbol $\circ$ indicates that the result of the associated computation is unknown. When the associated computation terminates, producing a value V, then the undetermined placeholder object is replaced by the *determined placeholder object* $\langle \mathbf{ph}\ p\ V\rangle$. This change of representation explicates *touch* operations in the form of side-conditions on the appropriate transition rules. The conditions state that an undetermined placeholder object ($\langle \mathbf{ph}\ p\ \circ\rangle$) must have been replaced by a determined placeholder object ($\langle \mathbf{ph}\ p\ V\rangle$) before the program operation can take place. The conditions precisely identify the positions of **car**, **cdr**, **if** and **apply** that demand proper values and also show that operations like **cons** or the second position of **apply** do not need to know anything about the values they process.

[2] The machine is also far too abstract for the derivation of an implementation. This problem is also addressed by the following development.

The transition relation $\longmapsto_{pcek}$ reformulation of the relation $\longmapsto_{pc}$ that takes into account the change of state representation. We write $S \longmapsto^*_{pcek} S'$ if $S \longmapsto^{n,m}_{pcek} S'$ for some $n, m \in \mathbf{N}$

4.2 Correctness The correctness proof for the new machine involves two steps. The first step constructs an intermediate semantics by introducing placeholder objects into the $P(C)$-machine. The second step proves the correctness of the $P(CEK)$-machine with respect to the intermediate semantics using standard proof techniques [8], appropriately modified to account for parallel evaluation.

Theorem 4.1 $eval_{pcek} = eval_{pc}$

5 Touch Optimization

The $P(CEK)$-machine performs *touch* operations on arguments in placeholder-strict positions of all program operations. These implicit *touch* operations guarantee the transparency of placeholders, which makes **future**-based parallelism so convenient to use. Unfortunately, these compiler-inserted *touch* operations impose a significant overhead on the execution of annotated programs. For example, an annotated doubly-recursive version of *fib* performs 1.2 million *touch* operations during the computation of (*fib* 25).

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,[3] the overhead of performing implicit *touch* operations appears to be acceptable at first glance.

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

Evaluator:

$$
\begin{array}{lcl}
eval_{pcek} : \Lambda_a^0 & \longrightarrow & Answers \cup \{\mathtt{error}, \bot\} \\
eval_{pcek}(P) & = & \begin{cases} unload_{pcek}[E(x)] & \text{if } \langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} \langle x, E, \epsilon\rangle \\ \mathtt{error} & \text{if } \langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} \mathtt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists S_i \in State_{pcek}, n_i, m_i \in N \text{ with } m_i > 0,\ S_0 = \langle P, \emptyset, \epsilon\rangle \text{ and } S_i \longmapsto^{n_i, m_i}_{pcek} S_{i+1} \end{cases}
\end{array}
$$

Data Specifications:

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

Auxiliary Functions:

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

Transition Rules:

$$
\begin{array}{rcll}
\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 & (\mathit{bind\text{-}cons}) \\
\multicolumn{4}{c}{\text{transition rules for } (\textbf{let}\ (x\ c)\ M),\ (\textbf{let}\ (x\ y)\ M) \text{ and } (\textbf{let}\ (x\ (\lambda y.N))\ M) \text{ are similar to } (\mathit{bind\text{-}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 & (\mathit{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 } touch_{pcek}[E(y)] = (\textbf{cons}\ V_1\ V_2) \\ \mathtt{error} & \text{if } touch_{pcek}[E(y)] \notin Pair_{pcek} \cup \{\circ\} \end{cases} & (\mathit{car}) \\
\langle(\textbf{let}\ (x\ (\textbf{cdr}\ y))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \text{analogous to } (\mathit{car}) & (\mathit{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 } touch_{pcek}[E(y)] \notin \{\mathtt{false}, \circ\} \\ \langle M_2, E, \langle \textbf{ar}\ x, M, E\rangle.K\rangle & \text{if } touch_{pcek}[E(y)] = \mathtt{false} \end{cases} & (\mathit{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 } touch_{pcek}[E(y)] = \langle(\lambda x'.N), E'\rangle \\ \mathtt{error} & \text{if } touch_{pcek}[E(y)] \notin Cl_{pcek} \cup \{\circ\} \end{cases} & (\mathit{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 & (\mathit{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 & (\mathit{future\text{-}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) & (\mathit{fork}) \\
(\textbf{f-let}\ (p\ \langle x, E, \epsilon\rangle)\ S) & \longmapsto^{1,1}_{pcek} & S[p := E(x)] & (\mathit{join}) \\
\multicolumn{4}{c}{\text{transition rules } (\mathit{join\text{-}error}),\ (\mathit{lift}),\ (\mathit{parallel}),\ (\mathit{reflexive}) \text{ and } (\mathit{transitive}) \text{ are as for the } P(C)\text{-machine}}
\end{array}
$$

Figure 4: The $P(CEK)$-machine

Unfortunately, a standard technique for increasing execution speed in Scheme systems is to disable type-checking typically based on informal correctness arguments or based on type verifiers for the underlying sequential language [33]. 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 on every invocation:

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

Performing these *placeholder?* checks can add a significant overhead to the execution time. Kranz [22] 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, 20, 31]. 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

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

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

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

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

Figure 5: Non-*touch*ing transition rules

traditional solution.

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

5.1 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 $\underline{\mathbf{car}}$, $\underline{\mathbf{cdr}}$, $\underline{\mathbf{if}}$ and $\underline{\mathbf{apply}}$, respectively:

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

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 undefined. The extended language is called $\underline{\Lambda_a}$.

We define the semantics of the extended language $\underline{\Lambda_a}$ by extending the $P(CEK)$-machine with the additional transition rules described in Figure 5. The evaluator for the extended language, $\underline{eval_{pcek}}$, is defined in the usual way (cmp. Figure 4). Unlike $eval_{pcek}$, the evaluator $\underline{eval_{pcek}}$ is no longer a function. There are programs in $\underline{\Lambda_a}$ for which the evaluator $\underline{eval_{pcek}}$ can either return a value or can be unspecified because of the application of a non-*touch*ing operation to a placeholder. Still, the two evaluators clearly agree on programs in Λ_a.

Lemma 5.1 *For* $P \in \Lambda_a$, $\underline{eval_{pcek}}(P) = eval_{pcek}(P)$.

5.2 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 $(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M)$ and we can prove that y is never bound to a placeholder. Then we can replace the expression by the form $(\mathbf{let}\ (x\ (\underline{\mathbf{car}}\ y))\ M)$, which the machine can execute more efficiently without performing a test for placeholdership on y.

This optimization technique relies on a detailed data-flow analysis of the program that determines a conservative approximation to the set of run-time values for each variable. More specifically, we assume that the analysis returns a *valid set environment*, which is a table mapping program variables to a set of run-time values[4] that subsumes the set of values associated with that variable during an execution.

Definition 5.2. (*Set environments, validity*) Let P be a program and let $Vars_P$ be the set of variables occurring in P.

- A mapping $\mathcal{E} : Vars_P \rightarrow \mathcal{P}(Value_{pcek})$ is a *set environment*.
- A set environment $\mathcal{E}$ is *valid* for P if $S \models \mathcal{E}$ holds for every S such that $\langle P, \emptyset, \epsilon\rangle \longmapsto^{*}_{pcek} S$.
- The relation $S \models \mathcal{E}$ holds if every environment in S maps every variable x to a value in $\mathcal{E}(x)$.

∎

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 $\mathcal{T}$ is defined in Figure 6.

The function *sba*, described in the next section, always returns a valid set environment for a program. Assuming the correctness of of set-based analysis, the touch optimization algorithm preserves the meaning of programs, since each transition step of a source program P corresponds to a transition step of the optimized program.

Theorem 5.3 (Correctness of *touch*-optimization) *For* $P \in \Lambda_a^0$, $eval_{pcek}(P) = \underline{eval_{pcek}}(\mathcal{T}_{sba(P)}[P])$.

Any implementation that realizes $\underline{eval_{pcek}}$ correctly can therefore make use of our optimization technique.

6 Set-Based Analysis for Futures

We develop the analysis that produces valid set environments in two steps. 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 program may assume. Any set environment satisfying these constraints

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

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

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

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

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

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

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

is a valid set environment. Second, we develop an algorithm for finding the minimal set environment satisfying these constraints. The constraints we produce are similar to those in Heintze's work on set based analysis for ML [14], though our derivation of these constraints differs substantially.

6.1 Deriving Set Constraints We derive constraints on valid set environments by analyzing the transition rules of the machine. Each constraint we produce is of the form:

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

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

Let P be the program of interest, and suppose that the evaluation of P involves the transition $S \longmapsto^{n,m}_{pcek} S'$, where $S \models \mathcal{E}$. We derive constraints on $\mathcal{E}$ sufficient to ensure that $S' \models \mathcal{E}$ by case analysis on the last transition rule used for $S \longmapsto^{n,m}_{pcek} S'$.

We present three representative cases:

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

$$
\begin{array}{rcl}
S & = & \langle (\textbf{let}\ (x\ c)\ M), E, K \rangle \\
\longmapsto^{1,1}_{pcek}\ S' & = & \langle M, E[x \leftarrow c], K \rangle
\end{array}
$$

  This rule binds x to the constant c, where the term $(\textbf{let}\ (x\ c)\ M)$ occurs in P. To ensure that the set environment $\mathcal{E}$ includes c as one of the possible values of x, we demand that $\mathcal{E}$ satisfy the constraint:

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

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

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

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

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

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

$$
\begin{array}{rcl}
S & = & \langle x, E', \langle \textbf{ar}\ y, M, E \rangle . K \rangle \\
\longmapsto^{1,1}_{pcek}\ S' & = & \langle M, E[y \leftarrow E'(x)], K \rangle
\end{array}
$$

  We need to ensure that $\mathcal{E}$ includes $E'(x)$ as a possible value of y. However, when analyzing a "return instruction" x, we have no information regarding the possible activation records that may receive the value of x during an execution. In contrast, when analyzing an application expression $(\textbf{let}\ (y\ (\textbf{apply}\ f\ z))\ M)$, we know both the calling context and, from $\mathcal{E}(f)$, the set of closures that can possibly be invoked. Furthermore, if $\langle (\lambda x'. N), E \rangle$ is the closure being invoked, then the result of the application will be the current binding of the variable $FinalVar[N]$, where $FinalVar$ is the following function from expressions to variables:

$$
\begin{array}{rcl}
FinalVar : \Lambda_a & \longrightarrow & Vars \\
FinalVar[x] & = & x \\
FinalVar[(\textbf{let}\ (x\ V)\ M)] & = & FinalVar[M] \\
FinalVar[(\textbf{let}\ (x\ (\textbf{future}\ N))\ M)] & = & FinalVar[M] \\
\ldots & & \ldots
\end{array}
$$

  To ensure that $\mathcal{E}$ accounts for the binding of x to the value of $FinalVar[N]$ for all closures that may be invoked, we demand that $\mathcal{E}$ satisfy the constraint

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

Examining each of the transition rules of the machine in a similar manner results in eleven program-based *set constraints* $C_1^P, \ldots, C_{11}^P$ (see Figure 7) sufficient to ensure that a set environment is valid. Put differently, if a set environment $\mathcal{E}$ satisfies the set constraints for a program P, it is easy to prove, using induction on the length of the transition sequence, that $\mathcal{E}$ is valid for P.

Theorem 6.1 (Soundness of Constraints) *If $\mathcal{E}$ satisfies $C_1^P, \ldots, C_{11}^P$, then $\mathcal{E}$ is valid for P.*

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

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

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

$$\frac{(\textbf{let}\ (x\ (\textbf{cons}\ y_1\ y_2))\ M) \in P \qquad V_i \in \mathcal{E}(y_i), i = 1,2}{(\textbf{cons}\ V_1\ V_2) \in \mathcal{E}(x)} \qquad (C_4^P)$$

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

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

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

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

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

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

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

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

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

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

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

$$\frac{(\textbf{let}\ (x\ (\textbf{cons}\ y_1\ y_2))\ M) \in P \qquad \overline{\mathcal{E}}(y_i) \neq \emptyset, i = 1,2}{(\textbf{cons}\ y_1\ y_2)_P \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_4^P})$$

$$\frac{\begin{array}{c}(\textbf{let}\ (x\ (\textbf{car}\ y))\ M) \in P \qquad \overline{V} \in \overline{\mathcal{E}}(y) \\ (\textbf{cons}\ z_1\ z_2)_P \in \overline{touch}\left[\overline{\mathcal{E}}, \overline{V}\right] \qquad \overline{V}_1 \in \overline{\mathcal{E}}(z_1)\end{array}}{\overline{V}_1 \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_5^P})$$

$$\frac{\begin{array}{c}(\textbf{let}\ (x\ (\textbf{cdr}\ y))\ M) \in P \qquad \overline{V} \in \overline{\mathcal{E}}(y) \\ (\textbf{cons}\ z_1\ z_2)_P \in \overline{touch}\left[\overline{\mathcal{E}}, \overline{V}\right] \qquad \overline{V}_2 \in \overline{\mathcal{E}}(z_2)\end{array}}{\overline{V}_2 \in \overline{\mathcal{E}}(x)} \qquad (\overline{C_6^P})$$

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

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

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

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

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

Auxiliary Function $\overline{touch}$:

$$\begin{array}{rcl}
\multicolumn{3}{l}{\overline{touch} : AbsEnv_P \times AbsValue_P \longrightarrow \mathcal{P}(AbsValue_P)} \\
\overline{touch}\left[\overline{\mathcal{E}}, c_P\right] & = & \{c_P\} \\
\overline{touch}\left[\overline{\mathcal{E}}, (\lambda x.M)_P\right] & = & \{(\lambda x.M)_P\} \\
\overline{touch}\left[\overline{\mathcal{E}}, (\textbf{cons}\ x\ y)_P\right] & = & \{(\textbf{cons}\ x\ y)_P\} \\
\overline{touch}\left[\overline{\mathcal{E}}, \langle \textbf{ph}\ x_P\rangle\right] & = & \\
\multicolumn{3}{r}{\{\overline{W} \mid \overline{U} \in \overline{\mathcal{E}}(y) \text{ and } \overline{W} \in \overline{touch}\left[\overline{\mathcal{E}}, \overline{U}\right]\}}
\end{array}$$

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

6.2 Solving Set Constraints

The class of set environments for a given program P, denoted $SetEnv_P$, forms a complete lattice under the natural pointwise partial ordering. Smaller set environments correspond to more accurate approximations, because they include include fewer extraneous bindings. We define set-based analysis as the function that returns the least set environment satisfying the set constraints.

Definition 6.2. (*sba*)

$$sba(P) = \sqcap\{\mathcal{E} \mid \mathcal{E} \text{ satisfies } C_1^P, \ldots, C_{11}^P\}$$

∎

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

$$\begin{array}{l}\overline{V} \in AbsValue_P ::= \\ \qquad c_P \mid (\lambda x.\,M)_P \mid (\textbf{cons}\ x\ y)_P \mid \langle \textbf{ph}\ x_P\rangle \mid \langle \textbf{ph}\ \circ\rangle\end{array}$$

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 this set is $O(|P|)$, where $|P|$ is the length of P.

Abstract values provide finite representations for the infinite set environments encountered during set-based analysis. Specifically, an *abstract set environment* $\overline{\mathcal{E}}$ is a mapping from variables in P to sets of abstract values. The class of all abstract set environments for a program P is denoted $AbsEnv_P$. Each abstract set environment represents a particular set environment according to the following function:

$$\mathcal{F} : AbsEnv_P \longrightarrow SetEnv_P$$
$$\mathcal{F}\left(\overline{\mathcal{E}}\right)(x) = \{V \mid V \varepsilon \overline{\mathcal{E}}(x)\}$$

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

Reformulating the set constraints from Figure 7 for abstract set environments produces the *abstract constraints* $\overline{C_1^P}, \ldots, \overline{C_{11}^P}$ in Figure 8. We define $\overline{sba}(P)$ be the least abstract set environment satisfying the abstract constraints with respect to P.

Definition 6.3. ($\overline{sba}$)

$$\overline{sba}(P) = \sqcap\{\overline{\mathcal{E}} \mid \overline{\mathcal{E}} \text{ satisfies } \overline{C_1^P}, \ldots, \overline{C_{11}^P}\}$$

∎

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

Theorem 6.4 $sba(P) = \mathcal{F}\left(\overline{sba}(P)\right)$

Since $AbsEnv_P$ is finite, of size $O(|P|^2)$, we can calculate $\overline{sba}(P)$ in an iterative manner, starting with the empty abstract set environment $\overline{\mathcal{E}}(x) = \emptyset$. Since we can extend $\overline{\mathcal{E}}$ at most $O(|P|^2)$ times, this algorithm terminates. Furthermore, each time we extend $\overline{\mathcal{E}}$ with a new binding, calculating the additional bindings implied by that new binding takes at most $O(|P|)$ time. Hence, the algorithm runs in $O(|P|^3)$ time.

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

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

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

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

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

7 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 as described in the previous sections extended to a sufficiently large subset of functional Scheme.[5] 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 shared-memory multiprocessor [2]. Figure 9 describes these benchmarks.

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`	Computes the shortest path between all pairs in a 117 node graph using Floyd's algorithm.
`abisort`	Sorts 16384 integers using adaptive bitonic sort.
`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 at a given value of x.

Figure 9: Description of the benchmark programs

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`. How-

[5] 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.

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

Figure 10: Benchmark Results

ever, 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.

8 Related Work

The literature on programming languages contains a number of semantics for parallel, Scheme-like languages. The only one that directly deals with parallelism based on transparent annotations is Moreau's Ph.D. thesis [27]. Moreau studies the functional core of Scheme extended with **pcall** (for evaluating function and argument expressions of an application in parallel) and first-class continuations. His primary goal is to design a semantics for the language that treats **pcall** as a pure annotation. His correctness proofs is far more complicated than our techniques, due to the inclusion of continuations.

Independently, Reppy [28] and Leroy [24] define a formal operational semantics for an ML-like language with first-class synchronization operations. Reppy's language, Concurrent ML, can provide the **future** mechanism as an abstraction over the given primitives. The semantics is a two-level rewriting system. Reppy uses his semantics to prove a type soundness theorem. Leroy formulates a semantics for a subset of CML in the traditional "natural" semantics framework. He also uses his semantics to prove the type soundness of the complete language. Neither Reppy nor Leroy used their semantics for developing analyses or optimizations.

Jaganathan and Weeks [18] define an operational semantics for a simple function language extended with the **spawn** construct by extending Deutsch's transition semantics [5]. They described an analysis for their language that they intend to use in a forthcoming compiler, but they do not have an implementation of their analysis for a full language like functional Scheme, and they do not have optimization algorithms that exploit the results of their analysis.

Wand [32] recently extended his work on correctness proofs for sequential compilers to parallel languages. In his prior work on the correctness of sequential compilers, he derived compilers from the semantic mappings that translate syntax into λ-calculus expressions. The extension of this work to parallel compilers starts from a semantic mapping that translates a Scheme-like language with process creation and communication constructs into a higher-order calculus of communication and computation. After separating the compiler from the "machine", the correctness proof is a combination of the sequential correctness proof and a correctness proof for the parallel portion of the language. The proof techniques are related to the ones we used to prove the equivalence of the P(CEK)-machine and the P(C)-machine.

Kranz et al. [23] 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 [21] 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.

Our analysis methods most closely follows Heintze's work on set-based analysis for the *sequential* language ML [13, 14], but the extension of his technique to parallel languages requires a substantial reformulation of the derivation and correctness proof. Specifically, Heintze uses the "natural" semantics framework to define a set-based "natural" semantics, from which he reads off safeness conditions on set environments. He then presents set constraints whose solution is the minimal safe set environment. We start from a parallel abstract machine and avoid these intermediate steps by deriving our set constraints and proving their correctness directly from the abstract machine semantics.

Other techniques for static analysis of sequential programs include abstract interpretation [3, 4] and Shivers' 0CFA [30]. The relationship between abstract interpretation and set-based analysis was covered by Heintze [13].

Sequential optimization techniques such as tagging optimization [15] and soft-typing [33] are similar in character to *touch* optimization. Both techniques remove the type-dispatches required for dynamic type-checking wherever possible, without changing the behavior of programs, in the same fashion as we remove *touch* operations. However, the analyses relies on conventional type inference techniques.

9 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.

References

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

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

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

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

[5] Deutsch, A. *Modèles Opérationnels de Language de Programmation et Représentations de Relations sue des Languages Rationnels avec Application a la Détermination Statique de Propriétes de Partages Dynamiques de Données.* PhD thesis, Universite Paris VI, 1992.

[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.

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

[8] Felleisen, M., and Friedman, D. P. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts* (Aug. 1986), 193–219.

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

[10] Flanagan, C., and Felleisen, M. Well-founded touch optimization of Parallel Scheme. Rice University Comp. Sci. TR94-239.

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

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

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

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

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

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

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

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

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

[20] Kessler, R.R., and R. Swanson. Concurrent scheme. [16:200–234].

[21] Knopp, J. Improving the performance of parallel lisp by compile time analysis. [16:271–277].

[22] Kranz, D., Halstead, R., and Mohr, E. Mul-T: A high-performance parallel lisp. [16:306–321].

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

[24] Leroy, X. *Typage polymorphe d'un langage algorithmique.* PhD thesis, Université Paris 7, 1992.

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

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

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

[28] Reppy, J.H. *Higher-Order Concurrency.* PhD thesis, Cornell University, Jan. 1992.

[29] Sabry, A., and Felleisen, M. Is continuation-passing useful for data flow analysis. In *PLDI* (1994), 1–12.

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

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

[32] Wand, M. Compiler correctness for parallel languages. Unpublished manuscript, 1995.

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