The Semantics of Future

Cormac Flanagan
Matthias Felleisen

Rice COMP TR94-238

October 1994

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

The Semantics of Future

Cormac Flanagan* Matthias Felleisen*

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

Abstract

The **future** annotation introduced by MultiLisp provides a simple method for taming the implicit parallelism of functional programs. Prior research on **future**s has concentrated on implementation and design issues, and has largely ignored the development of a semantic characterization of **future**s. This paper presents four operational semantics for an idealized functional language with **future**s with varying degrees of intensionality. The first semantics defines **future** to be a semantically-transparent annotation. The second semantics interprets a **future** expression as a potentially parallel task. The third semantics explicates the coordination of parallel tasks and the need for *touch* operations on placeholder-strict arguments to certain primitive operations by introducing placeholder objects. The fourth and last semantics is a low-level refinement of the third semantics, which explicates just enough information to permit the smooth derivation of program analyses. The paper includes proofs showing the equivalence of these semantics.

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

Contents

1 Futures for Parallel Computation **1**

2 A Functional Language with Futures **2**
- 2.1 Syntax 2
- 2.2 Definitional Semantics 4

3 Parallel Operational Semantics **5**
- 3.1 Specification of the $P(C)$-machine 5
  - State Space 5
  - Transition Rules 7
  - Indeterminism 8
  - Evaluation 8
  - Placeholder Transparency and Synchronization 9
- 3.2 Consistency of the $P(C)$-Machine 11
- 3.3 Correctness of the $P(C)$-Machine 17

4 Placeholder Object Semantics **18**
- 4.1 Specification of the $P(C_{ph})$-machine 19
- 4.2 Correctness of the $P(C_{ph})$-machine 21

5 The Low-Level Operational Semantics **24**
- 5.1 Specification of the $P(CEK)$-machine 24
- 5.2 Correctness of the $P(CEK)$-machine 27

6 Related Work **27**

7 Applications of the Semantics **28**

A Correctness Proof for the $P(CEK)$-Machine **29**

1 Futures for Parallel Computation

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 [1, 11] 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 to the rest of the program.

While past research on **future**s has concentrated on implementation and design issues [4, 13, 20, 16, 18, 15, 19], this technical report focuses on the semantics of **future**s. Specifically, it presents a series of semantics with varying degrees of intensionality. The first semantics equates the value and behavior of (**future** e) as that of e, which is the simplest way to interpret **future** as an annotation. The second semantics interprets an annotated expression as a potentially parallel task. The third semantics explicates the coordination of parallel tasks with the introduction of explicit placeholder objects. Since all of these semantics are formulated as program rewriting systems, the equivalence proofs rely on standard techniques like bisimulation and a diamond lemma. The fourth and last semantics is a low-level refinement of the third semantics, which explicates just enough information to permit the smooth derivation of program analyses.

A companion paper describes the use of our semantics in program optimization [7]. It focuses on the derivation of a provably correct program analysis algorithm from the last semantics. The paper also describes a optimization algorithm for lowering the overhead of task coordination in a language with **future**s, and the effects of implementing this optimization for the Gambit compiler [4].

The presentation of our semantics for **future** proceeds as follows. The second section introduces the language we consider and its definitional semantics, which interprets **future**s as annotations. The third section presents a parallel program rewriting semantics of **future**s, and the fourth section introduces placeholder objects and *touch* operations. The fifth section contains the low-level refinement. The sixth section compares our approach to related work concerning the semantics of parallel functional languages, and the last section discusses some applications for the our semantics. Appendix A contains the correctness proof for the low-level semantics.

Notation We use $f : A \longrightarrow B$ to denote that f is a total function from A to B, and similarly use $f : A \longrightarrow_f B$ to mean that f is a finite map from A to B.

2 A Functional Language with Futures

2.1 Syntax

Our motivation in developing a semantics for **future** is to provide a sound basis for the compilation of programs with **future**s. Compilers typically convert source programs into a simple intermediate representation, and then proceed to compile and optimize the intermediate representation of the program. Therefore, we formulate the semantics of **future**s for an intermediate representation of an idealized functional language. Specifically, we use the subset of A-normal forms [9] of a λ-calculus-like language with a **let** construct, a **future** construct, a conditional and an explicit **apply** primitive: see Figure 1. The language also includes the primitives **cons**, **car**, and **cdr** for list manipulation, which will serve to illustrate the treatment of primitive operations, and an unspecified set of basic primitive constants that includes at least numbers and the empty list `nil`.

The key property of terms in A-normal form is that each intermediate value is explicitly named and where the order of execution follows the lexical nesting of **let**-expressions. The use of A-normal forms facilitates the compile-time analysis of programs since every intermediate value is named [22], and it simplifies the definition of abstract machines [9], a fact that we exploit in the development of abstract machines below.

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

FIGURE 1: The A-normalized Language Λ_a

A variable occurrence is *free* if it is not bound by an enclosing λ-expression or **let**-expression. A term is *closed* if it contains no free variables. We identify terms that differ only by consistent renaming of bound variables. The operation $M[x \leftarrow V]$ denotes the capture-free substitution of V for all free occurrences of x within M. We use X^0 to denote the set of closed terms of type X (terms, values).

Evaluator:

$$eval_c : \Lambda_a^0 \longrightarrow Answers \cup \{\texttt{error}, \perp\}$$

$$eval_c(P) = \begin{cases} unload_c[V] & \text{if } P \longmapsto_c^* V \\ \texttt{error} & \text{if } P \longmapsto_c^* \texttt{error} \\ \perp & \text{if } \forall i \in \mathbf{N}\ \exists M_i \in State_c \text{ such that} \\ & \quad P = M_0 \text{ and } M_i \longmapsto_c M_{i+1} \end{cases}$$

Data Specifications:

$$\begin{array}{lclcll}
S & \in & State_c & ::= & M \mid \texttt{error} & \text{(States)} \\
M & \in & \Lambda_c & ::= & V & \text{(Run-time Language)} \\
 & & & & \mid (\mathbf{let}\ (x\ V)\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{future}\ M))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{car}\ V))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{cdr}\ V))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{if}\ V\ M\ M))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{apply}\ V\ V))\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ M)\ M) & \\
V & \in & Value_c & ::= & c \mid x \mid (\lambda x.\, M) \mid (\mathbf{cons}\ V\ V) & \text{(Run-time Values)} \\
F & \in & FinalState_c & ::= & V \mid \texttt{error} & \text{(Final States)} \\
A & \in & Answers & ::= & c \mid \texttt{procedure} \mid (\mathbf{cons}\ A\ A) & \text{(Answers)} \\
\mathcal{E} & \in & EvalCtxt & ::= & [\,] & \text{(Evaluation Contexts)} \\
 & & & & \mid (\mathbf{let}\ (x\ \mathcal{E})\ M) & \\
 & & & & \mid (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}))\ M) &
\end{array}$$

Unload Function:

$$\begin{array}{rcl}
unload_c : Value_c^0 & \longrightarrow & Answers \\
unload_c[c] & = & c \\
unload_c[(\lambda x.\, M)] & = & \texttt{procedure} \\
unload_c[(\mathbf{cons}\ V_1\ V_2)] & = & (\mathbf{cons}\ unload_c[V_1]\ unload_c[V_2])
\end{array}$$

Transition Function:

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

FIGURE 2: The sequential C-machine

2.2 Definitional Semantics

The semantics of the A-normalized language is a function from Λ_a^0 to results. A result is either an answer, which is a closed value with all λ-expression replaced by the tag `procedure`,[1] or `error`, indicating that some program operation was misapplied, or $\perp$, if the program diverges. We specify the definitional semantics for Λ_a^0 using a sequential abstract machine called the C-machine (see Figure 2), whose states are either closed terms in the run-time language Λ_c or the special state `error`, and whose deterministic transition rules are the typical leftmost-outermost reductions of the lambda calculus [5]. 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 only unusual transition rule is the one for **future** expressions. This rule 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 either a value or the special 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.

Lemma 2.1 (Uniform Evaluation Theorem) *Let $M \in State_c$.*

1. *If $M \notin FinalState_c$, there exists a unique term M' such that $M \longmapsto_c M'$.*
2. *If $M \in FinalState_c$, there is no term M' such that $M \longmapsto_c M'$.*

Proof: We use structural induction on $State_c$ to prove that any term that is not a final state matches the lefthand side of exactly one of the clauses for $\longmapsto_c$. Since each of these clauses defines a function on Λ_c, the relation $\longmapsto_c$ relates each term that is not a final state to a unique successor term. The proof that a final state has no successor is straightforward. ∎

The Unique Evaluation Theorem implies that the relation $eval_c$ is a well-defined 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 $eval_c(P) = \perp$.

Theorem 2.2 *The evaluator $eval_c$ is a total function.*

Since the evaluator $eval_c$ of the C-machine obviously agrees with the sequential semantics of the underlying functional language, **future** is clearly nothing but an annotation.

[1] We remove λ-expressions from answers so that the observable behavior of programs does not depend on the terms themselves, but only on their meaning.

3 Parallel Operational Semantics

The sequential C-machine defines **future** as an annotation, and ignores the *intension* of **future** as an advisory instruction concerning parallel evaluations. To understand this intensional aspect of **future** annotations, we need a semantics of **future** that models the concurrent evaluation of **future** expressions. For this purpose, we reformulate the C machine to account for parallelism. The result is the $P(C)$-machine.

Evaluator:

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

Data Specifications:

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

Unload Function:

$$
\begin{array}{rcl}
unload_{pc} : Value_{pc}^0 & \longrightarrow & Answers \\
unload_{pc}[W] & = & unload_c[W]
\end{array}
$$

Placeholder Substitution $S[p \leftarrow V]$:

$$
\begin{array}{rcl}
M[p \leftarrow V] & = & M \text{ with all free occurrences of } p \text{ replaced by } V \\
\mathtt{error}[p \leftarrow V] & = & \mathtt{error} \\
(\textbf{f-let}\ (p'\ S_1)\ S_2)[p \leftarrow V] & = & \begin{cases} (\textbf{f-let}\ (p'\ S_1[p \leftarrow V])\ S_2) & \text{if } p = p' \\ (\textbf{f-let}\ (p'\ S_1[p \leftarrow V])\ S_2[p \leftarrow V]) & \text{if } p \neq p' \end{cases}
\end{array}
$$

Free Placeholder Function FP:

$$
\begin{array}{rcl}
FP : State_{pc} & \longrightarrow & P(Ph\text{-}Vars) \\
FP(M) & = & \{p \mid p \text{ occurs in } M\} \\
FP(\mathtt{error}) & = & \emptyset \\
FP((\textbf{f-let}\ (p\ S_1)\ S_2)) & = & FP(S_1) \cup (FP(S_2) \setminus \{p\})
\end{array}
$$

FIGURE 3: The parallel $P(C)$-machine: Evaluator and Data Specifications

---

3.1 Specification of the $P(C)$-machine

State Space The state space of the $P(C)$-machine is defined in Figure 3. The set of $P(C)$ values includes the values of the sequential C-machine (constants, variables, closures

Transition Rules:

$$\mathcal{E}[\,(\mathbf{let}\ (x\ V)\ M)\,] \longmapsto_{pc}^{1,1} \mathcal{E}[\,M[x \leftarrow V]\,] \qquad (\mathit{bind})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{future}\ V))\ M)\,] \longmapsto_{pc}^{1,1} \mathcal{E}[\,M[x \leftarrow V]\,] \qquad (\mathit{future\text{-}id})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{car}\ V))\ M)\,] \longmapsto_{pc}^{1,1} \begin{cases} \mathcal{E}[\,M[x \leftarrow V_1]\,] & \text{if } V = (\mathbf{cons}\ V_1\ V_2) \\ \mathtt{error} & \text{if } V \neq (\mathbf{cons}\ V_1\ V_2), V \neq p \end{cases} \qquad (\mathit{car})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{cdr}\ V))\ M)\,] \longmapsto_{pc}^{1,1} \begin{cases} \mathcal{E}[\,M[x \leftarrow V_2]\,] & \text{if } V = (\mathbf{cons}\ V_1\ V_2) \\ \mathtt{error} & \text{if } V \neq (\mathbf{cons}\ V_1\ V_2), V \neq p \end{cases} \qquad (\mathit{cdr})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{if}\ V\ M_1\ M_2))\ M)\,] \longmapsto_{pc}^{1,1} \begin{cases} \mathcal{E}[\,(\mathbf{let}\ (x\ M_1)\ M)\,] & \text{if } V \neq \mathtt{nil}, V \neq p \\ \mathcal{E}[\,(\mathbf{let}\ (x\ M_2)\ M)\,] & \text{if } V = \mathtt{nil} \end{cases} \qquad (\mathit{if})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{apply}\ V_1\ V_2))\ M)\,] \longmapsto_{pc}^{1,1} \begin{cases} \mathcal{E}[\,(\mathbf{let}\ (x\ N[y \leftarrow V_2])\ M)\,] & \text{if } V_1 = (\lambda y.\,N) \\ \mathtt{error} & \text{if } V_1 \neq (\lambda y.\,N), V_1 \neq p \end{cases} \qquad (\mathit{apply})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)\,] \longmapsto_{pc}^{1,0} (\mathbf{f\text{-}let}\ (p\ N)\ \mathcal{E}[\,M[x \leftarrow p]\,]) \quad p \notin FP(\mathcal{E}) \cup FP(M) \qquad (\mathit{fork})$$

$$(\mathbf{f\text{-}let}\ (p\ V)\ S) \longmapsto_{pc}^{1,1} S[p \leftarrow V] \qquad (\mathit{join})$$

$$(\mathbf{f\text{-}let}\ (p\ \mathtt{error})\ S) \longmapsto_{pc}^{1,1} \mathtt{error} \qquad (\mathit{join\text{-}error})$$

$$(\mathbf{f\text{-}let}\ (p_2\ (\mathbf{f\text{-}let}\ (p_1\ S_1)\ S_2))\ S_2) \longmapsto_{pc}^{1,1} (\mathbf{f\text{-}let}\ (p_1\ S_1)\ (\mathbf{f\text{-}let}\ (p_2\ S_2)\ S_3)) \quad p_1 \notin FP(S_3) \qquad (\mathit{lift})$$

$$(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2) \longmapsto_{pc}^{a+c,b} (\mathbf{f\text{-}let}\ (p\ S_1')\ S_2') \quad \text{if } S_1 \longmapsto_{pc}^{a,b} S_1',\ S_2 \longmapsto_{pc}^{c,d} S_2' \qquad (\mathit{parallel})$$

$$S \longmapsto_{pc}^{0,0} S \qquad (\mathit{reflexive})$$

$$S \longmapsto_{pc}^{a+c,b+d} S'' \quad \text{if } S \longmapsto_{pc}^{a,b} S', S' \longmapsto_{pc}^{c,d} S'' \text{ and } a, c > 0 \qquad (\mathit{transitive})$$

FIGURE 4: The parallel $P(C)$-machine: Transition Rules

and pairs), which we refer to as *proper* values. To model the implementation of 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 state of the C-machine represents a single thread of control or *task*. To model the concurrent evaluation of **future** expressions by 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 the **future** expression, and the *secondary* sub-state S_2 is initially the 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**. We use $S[p \leftarrow V]$ and $\mathcal{E}[p \leftarrow W]$ to denote the capture-free substitution of V for all free occurrences of p within a state S and an evaluation context $\mathcal{E}$ respectively. A state is *closed* if it contains no free variables or free placeholders. The evaluation of S_1 is considered *mandatory*, since it is guaranteed to contribute to the completion of the computation. 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 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 The transition relation of the $P(C)$-machine is specified as a quadruple: see Figure 4. 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 relation is formulated as a collection of transition rules. The rules (*bind*), (*future-id*), (*car*), (*cdr*), (*if*) and (*apply*) are simply the transition rules of the C-machine, appropriately modified to allow for undetermined placeholders. 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 $\mathcal{E}[\ (\textbf{let}\ (x\ [\ ])\ M)\ ]$. 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 (**f-let** $(p\ S_1)\ S_2$).

The transition rules (*join*) and (*join-error*) merge distinct threads of evaluation. When the primary sub-state S_1 of a parallel state (**f-let** $(p\ S_1)\ S_2$) returns a value V, 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 the state (**f-let** $(p_2$ (**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.[2]

[2]The addition of this optimization rule is the result of proving the soundness of the semantics via the standard diamond lemma. Without this rule, the machine does not possess the diamond property (but is probably still consistent).

The rules (*reflexive*) and (*transitive*) close the relation under reflexivity and transitivity.

We write $S \longmapsto^{*}_{pc} S'$ if $S \longmapsto^{n,m}_{pc} S'$ for some $n, m \in \mathbf{N}$. A state S is in *normal form* if there is no state S' such that $S \longmapsto^{n,m}_{pc} S'$ with $n > 0$. A state is a *final state* if it is either a value, or the state `error`, and a state is *blocked* if it is in normal form but not a final state.

Indeterminism Unlike the *functional* C-machine, which maps each state to a unique successor state, the transition *relation* of the $P(C)$-machine has an important degree of freedom. The definition of the $P(C)$-machine does not specify when the transition rule (*fork*) applies. For example, consider the state $\mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)\ ]$. Since the context surrounding N is an evaluation context, an implementation of the machine may proceed either by evaluating N, or by creating a new task via a (*fork*) transition. The choice of whether or not to apply the transition rule (*fork*) is entirely up to the implementation of the machine. An implementation may *immediately* apply this rule whenever a **future** expression is encountered, realizing a task creation strategy called *eager task creation* [16, 23, 10]. Alternatively, an implementation may *never* invoke the rule (*fork*), resulting in a purely sequential evaluation. In between these two extremes lie a range of strategies where new tasks are created according to some implementation-dependent and possibly load-dependent algorithm. A particularly efficient strategy is *lazy task creation* [4, 19], where new tasks are created via *fork* transitions only when the additional parallelism can exploit idle computing resources.

A second source of indeterminism in the specification of the $P(C)$-machine 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. The only constraint, as specified in the definition of $eval_{pc}$, is that the implementation must perform mandatory computation steps on a regular basis.

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

$$P \ = \ (\mathbf{let}\ (x\ (\mathbf{future}\ E))\ \Omega)$$

where E is a term that raises a run-time error, and Ω is some diverging sequential term such that $\Omega \longmapsto^{1,1}_{pc} \Omega_1 \longmapsto^{1,1}_{pc} \Omega_2 \longmapsto^{1,1}_{pc} \cdots$. The sequential evaluator never executes Ω because P's result is `error`. In contrast, P admits the following infinite parallel transition sequence:

$$\begin{array}{rll}
P & \longmapsto^{1,0}_{pc} (\mathbf{f\text{-}let}\ (p\ E)\ \Omega) & \text{via } (\textit{fork}) \\
 & \longmapsto^{1,0}_{pc} (\mathbf{f\text{-}let}\ (p\ E)\ \Omega_1) & \text{since } \Omega \longmapsto^{1,1}_{pc} \Omega_1 \\
 & \longmapsto^{1,0}_{pc} (\mathbf{f\text{-}let}\ (p\ E)\ \Omega_2) & \text{since } \Omega_1 \longmapsto^{1,1}_{pc} \Omega_2 \\
 & \longmapsto^{1,0}_{pc} \cdots &
\end{array}$$

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 admits transition sequences that regularly includes mandatory transition steps.[3] 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 that mandatory transition steps are performed 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.

In summary, the $P(C)$-machine arbitrarily chooses any transition sequence that regularly performs mandatory computation, and reports on the behavior of that sequence. If the chosen transition sequence produces either a value V or `error`, then $eval_{pc}$ returns $unload_{pc}[V]$ or `error` respectively. If the chosen transition sequence does not terminate, then $eval_{pc}$ returns $\perp$. As we will prove below, the evaluator relation $eval_{pc}$ is a total function and agrees with the sequential evaluator $eval_c$.

Placeholder Transparency and Synchronization We say that a program operation is *placeholder-strict* in a position if it needs specific information about the value of the corresponding argument. For example, the operations **car** and **cdr** are placeholder-strict, and **if** and **apply** are placeholder-strict in their first position. Whenever an undetermined placeholder appears in a placeholder-strict argument position of one of these operations, then that operation must *block* until the placeholder is determined and specific information about the value of the argument is known. We model this behavior in the $P(C)$-machine via side-conditions associated with the transition rules (*car*), (*cdr*), (*if*) and (*apply*). These side-conditions ensure that if a placeholder-strict argument is an undetermined placeholder, then the transition rule cannot fire.

For a brief illustration of this idea, consider the following transition sequence for the following program P:

$$
\begin{array}{rcll}
P & = & (\mathbf{let}\ (y\ (\mathbf{future}\ (\mathbf{let}\ (x\ 1)\ (\mathbf{cons}\ x\ x)))) & (1)\\
 & & \qquad (\mathbf{let}\ (z\ (\mathbf{car}\ y))\ z)) & \\
 & \longmapsto_{pc}^{1,0} & (\mathbf{f\text{-}let}\ (p\ \boxed{(\mathbf{let}\ (x\ 1)\ (\mathbf{cons}\ x\ x))}) & (2)\\
 & & \qquad (\mathbf{let}\ (z\ (\mathbf{car}\ p))\ z)) & \\
 & \longmapsto_{pc}^{1,1} & (\mathbf{f\text{-}let}\ (p\ \boxed{(\mathbf{cons}\ 1\ 1)}) & (3)\\
 & & \qquad (\mathbf{let}\ (z\ (\mathbf{car}\ p))\ z)) & \\
 & \longmapsto_{pc}^{1,1} & (\mathbf{let}\ (z\ (\mathbf{car}\ (\mathbf{cons}\ 1\ 1)))\ z) & (4)\\
 & \longmapsto_{pc}^{1,1} & 1 & (5)
\end{array}
$$

The first transition in this sequence creates a new task for the evaluation of the **future** expression via a (*fork*) transition. After task creation (line 2), no transition steps are possible from the secondary sub-state (**let** (z (**car** p)) z). The transition rule (*car*) cannot fire since the argument to the operation (**car**) is a placeholder variable. Evaluation of the primary substate (highlighted with a box) proceeds unhindered. Once the primary substate produces a value (line 3), the transition rule (*join*) synchronizes the separate threads of computation by replacing all occurrences of p by that value. After synchronization (line 4),

[3] The concept of a mandatory step is closely related to the notion of legitimacy introduced by Katz and Weise [15].

the operation **car** applies to the new argument (**cons** 1 1), and execution continues with the program returning the answer 1. Since program operations block whenever an argument in a placeholder-strict position is undetermined, the $P(C)$-machine never performs a transition *before* a placeholder is determined that it would perform differently *after* the placeholder is determined. Hence the transition relation of the machine exhibits a *substitutivity* property: the transition relation commutes with substitution of values for placeholders. The proof of this property relies on a Substitution Lemma for placeholders.

Lemma 3.1 (Substitution Lemma for $\cdot[\cdot \leftarrow \cdot]$) *If $M \in \Lambda_{pc}$ and $V, W \in Value_{pc}$, then*

$$(M[p \leftarrow W])[x \leftarrow V[p \leftarrow W]] = (M[x \leftarrow V])[p \leftarrow W].$$

Proof: The proof proceeds by induction on the structure of M. ∎

Lemma 3.2 (Substitutivity; Placeholder Transparency) *If $S_1 \longmapsto_{pc}^{n,m} S_2$, then for any placeholder p and any $W \in Value_{pc}$, $S_1[p \leftarrow W] \longmapsto_{pc}^{n,m} S_2[p \leftarrow W]$.*

Proof: The proof is by lexicographic induction induction on n and on the size of S_1, and proceeds by case analysis on the last step in the transition $S_1 \longmapsto_{pc}^{n,m} S_2$.

- Suppose $S_1 \longmapsto_{pc}^{1,1} S_2$ via the rule (*car*) with $S_1 = \mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{car}\ V))\ M)\ ]$. Let $\mathcal{E}' = \mathcal{E}[p \leftarrow W]$, $V' = V[p \leftarrow W]$ and $M' = M[p \leftarrow W]$. We consider three possibilities for V:
  - Suppose $V = (\textbf{cons}\ V_1\ V_2)$. Then $S_2 = \mathcal{E}[\ M[x \leftarrow V_1]\ ]$. Let $V_1' = V_1[p \leftarrow W]$ and $V_2' = V_2[p \leftarrow W]$.

$$\begin{array}{rcll} S_1[p \leftarrow W] & = & \mathcal{E}'[\ (\textbf{let}\ (x\ (\textbf{car}\ (\textbf{cons}\ V_1'\ V_2')))\ M')\ ] & \\ & \longmapsto_{pc}^{1,1} & \mathcal{E}'[\ M'[x \leftarrow V_1']\ ] & \\ & = & S_2[p \leftarrow W] & \text{by Lemma 3.1} \end{array}$$

  - Suppose V is neither a pair nor a placeholder. Then $S_2 = \texttt{error}$, and since V' is neither a pair nor a placeholder,

$$\begin{array}{rcl} S_1[p \leftarrow W] & = & \mathcal{E}'[\ (\textbf{let}\ (x\ (\textbf{car}\ V'))\ M')\ ] \\ & \longmapsto_{pc}^{1,1} & \texttt{error} \\ & = & S_2[p \leftarrow W] \end{array}$$

  - The case where V is a placeholder is impossible, since no transitions are possible from such a state.
- The reasoning for the transition rules (*bind*), (*future-id*), (*cdr*), (*if*), (*apply*), (*fork*), (*join*), (*join-error*), (*lift*), (*reflexive*) and (*transitive*) is similar.
- We consider the case where $S_1 \longmapsto_{pc}^{n,m} S_2$ via the rule (*parallel*) in order to justify our use of lexicographic induction. For this case:

$$\begin{array}{rclcrcl} S_1 & = & (\textbf{f-let}\ (p'\ S_1')\ S_1'') & \quad & S_1' & \longmapsto_{pc}^{a,m} & S_2' \\ S_2 & = & (\textbf{f-let}\ (p'\ S_2')\ S_2'') & & S_1'' & \longmapsto_{pc}^{c,d} & S_2'' \end{array}$$

and $n = a + c$. Since S_1' and S_1'' are substates of S_1, by the inductive hypothesis we have:

$$\begin{array}{lll} S_1'[p \leftarrow W] & \longmapsto_{pc}^{a,m} & S_2'[p \leftarrow W] \\ S_1''[p \leftarrow W] & \longmapsto_{pc}^{c,d} & S_2''[p \leftarrow W] \end{array}$$

Hence:

$$\begin{array}{rll} S_1[p \leftarrow W] & = & (\textbf{f-let}\ (p'\ S_1'[p \leftarrow W])\ S_1''[p \leftarrow W]) \\ & \longmapsto_{pc}^{n,m} & (\textbf{f-let}\ (p'\ S_2'[p \leftarrow W])\ S_2''[p \leftarrow W]) \quad \text{via } (\textit{parallel}) \\ & = & S_2[p \leftarrow W] \end{array}$$

∎

The Substitutivity Lemma shows that undetermined placeholders represent results of parallel computations in a transparent manner. This property is crucial in proving the correctness of the machine.

3.2 Consistency of the $P(C)$-Machine

The *observable* behavior of the $P(C)$-machine on a given program is deterministic, despite its indeterminate *internal* behavior. We prove this consistency in the traditional manner, using a modified form of the Diamond Lemma. The proof of the Diamond Lemma relies on the following two obvious properties of the relation $\longmapsto_{pc}$.

Lemma 3.3 *(1) If $S \longmapsto_{pc}^{n,m} S'$, then $n \geq m$. (2) If $S \longmapsto_{pc}^{0,0} S'$, then $S = S'$.*

Proof: By induction of n, and case analysis of $S \longmapsto_{pc}^{n,m} S'$. ∎

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 twice the *total* number of steps on the transition from S_1 to S_4 via S_3, and vice-versa. This bound implies that all transition sequences for a given program exhibit the same termination behavior.

Lemma 3.4 (Modified Diamond Lemma) *Let $S_1, S_2, S_3 \in State_{pc}$. If $S_1 \longmapsto_{pc}^{n_1,m_1} S_2$ and $S_1 \longmapsto_{pc}^{n_2,m_2} S_3$, then there exists $S_4 \in State_{pc}$ and $n_3, m_3, n_4, m_4 \in \mathbf{N}$ such that $S_2 \longmapsto_{pc}^{n_3,m_3} S_4$ and $S_3 \longmapsto_{pc}^{n_4,m_4} S_4$. Furthermore $m_1 + m_3 \leq 2(n_2 + n_4)$ and $m_2 + m_4 \leq 2(n_1 + n_3)$.*

Proof: The proof is by lexicographic induction on n_1, n_2 and the size of S_1. We define the transition relation (*seq*) to be the union of the transition relations (*bind*), (*car*), (*cdr*), (*if*) and (*apply*). Then the transition relation of the $P(C)$-machine is the union of the eight relations (*seq*), (*future-id*), (*fork*), (*join*), (*join-error*), (*lift*), (*parallel*), (*reflexive*) and (*transitive*). The proof proceeds by case analysis of the pair of transition rules used for $S_1 \longmapsto_{pc}^{n_1} S_2$ and $S_1 \longmapsto_{pc}^{n_2} S_3$.

The following table enumerates the possible combinations after symmetry considerations, and annotates each case with a reference to the argument used to prove that case.

	(seq	*(f-id*	*(fork*	*(join*	*(join-error*	*(lift*	*(parallel*	*(refl.*	*(trans.*
(seq)	≡	×	3	×	×	×	×	1	2
(future-id)		≡	4	×	×	×	×	1	2
(fork)			5	×	×	×	×	1	2
join				≡	×	×	6	1	2
(join-error)					≡	×	7	1	2
(lift)						≡	9	1	2
(parallel)							8	1	2
(reflexive)								1	1
(transitive)									2

The following arguments show that the lemma holds in each of the above cases.

(×) The cases marked by the symbol × are impossible, since the domains of the respective relations are disjoint.

(≡) The cases marked by the symbol ≡ hold, since the relation in question is a partial function, and hence $S_2 = S_3$.

(1) For the case where $S_1 \longmapsto_{pc}^{0,0} S_2$ via the rule *reflexive*, take $S_4 = S_2$.

(2) For the case where $S_1 \longmapsto_{pc}^{n_1,m_1} S_2$ via the rule *transitive*, we have that $S_1 \longmapsto_{pc}^{a_1,b_1} S_5 \longmapsto_{pc}^{a_2,b_2} S_2$. The following diagram outlines our proof technique for this case.

$$\begin{tikzcd}
S_1 \arrow[r, "\longmapsto_{pc}^{a_1,b_1}", phantom] \arrow[d, "n_2,m_2"] & S_5 \arrow[r, "\longmapsto_{pc}^{a_2,b_2}", phantom] \arrow[d, dotted, "a_3,b_3"] & S_2 \arrow[d, dotted, "n_3,m_3"] \\
S_3 \arrow[r, dotted, "a_4,b_4"] & S_6 \arrow[r, dotted, "a_5,b_5"] & S_4
\end{tikzcd}$$

Since $a_1 < n_1$, by the inductive hypothesis there exists $S_6 \in State_{pc}$ and $a_3, b_3, a_4, b_4 \in \mathbf{N}$ such that

$$\begin{array}{lllcrcl}
S_5 & \longmapsto_{pc}^{a_3,b_3} & S_6 & \quad & b_1 + b_3 & \leq & 2(n_2 + a_4) \\
S_3 & \longmapsto_{pc}^{a_4,b_4} & S_6 & \quad & m_2 + b_4 & \leq & 2(a_1 + a_3)
\end{array}$$

Since $S_5 \longmapsto_{pc}^{a_2,b_2} S_2$, $S_5 \longmapsto_{pc}^{a_3,b_3} S_6$ and $a_2 < n_1$, by the inductive hypothesis there exists $S_4 \in State_{pc}$ and $n_3, m_3, a_5, b_5 \in \mathbf{N}$ such that

$$\begin{array}{lllcrcl}
S_2 & \longmapsto_{pc}^{n_3,m_3} & S_4 & \quad & b_2 + m_3 & \leq & 2(a_3 + a_5) \\
S_6 & \longmapsto_{pc}^{a_5,b_5} & S_4 & \quad & b_3 + b_5 & \leq & 2(a_2 + n_3)
\end{array}$$

Let $n_4 = a_4 + a_5$ and $m_4 = b_4 + b_5$, and we have that $S_2 \longmapsto_{pc}^{n_3,m_3} S_4$ and $S_3 \longmapsto_{pc}^{n_4,m_4} S_4$. Furthermore,

$$\begin{aligned} m_1 + m_3 &= b_1 + b_2 + m_3 \\ &= (b_1 + b_3) + (b_2 + m_3) - b_3 \\ &\leq 2(n_2 + a_4) + 2(a_3 + a_5) - b_3 && (I.H.) \\ &= 2 \times n_2 + 2(a_4 + a_5) + (2 \times a_3 - b_3) \\ &\leq 2(n_2 + n_4) \end{aligned}$$

Similarly, $m_2 + m_4 \leq 2(n_1 + n_3)$.

(3) Suppose $S_1 \longmapsto_{pc}^{1,1} S_2$ via the rule (*seq*), and $S_1 \longmapsto_{pc}^{1,0} S_3$ via the rule (*fork*). Then $n_1 = m_1 = n_2 = 1$, $m_2 = 0$ and

$$\begin{aligned} S_1 &= \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ N\ ]))\ M)\ ] \\ S_3 &= (\mathbf{f\text{-}let}\ (p\ \mathcal{E}'[\ N\ ])\ \mathcal{E}[\ M[x \leftarrow p]\ ]) \end{aligned}$$

We consider two cases for $S_1 \longmapsto_{pc}^{1,1} S_2$ separately.

- Suppose $\mathcal{E}'[\ N\ ] \longmapsto_{pc}^{1,1} \mathcal{E}'[\ N'\ ]$, and $S_2 = (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ N'\ ]))\ M)$. Set $S_4 = (\mathbf{f\text{-}let}\ (p\ \mathcal{E}'[\ N'\ ])\ \mathcal{E}[\ M[x \leftarrow p]\ ])$. Clearly $S_2 \longmapsto_{pc}^{1,0} S_4$ via (*fork*), $S_3 \longmapsto_{pc}^{1,1} S_4$ via (*seq*), and the indices satisfy the inequalities.
- Alternatively, suppose $S_2 = \texttt{error}$. Then let $S_4 = \texttt{error}$. Observe that

$$\begin{aligned} S_3 &\longmapsto_{pc}^{1,1} (\mathbf{f\text{-}let}\ (p\ \texttt{error})\ \mathcal{E}[\ M[x \leftarrow p]\ ]) && \text{via } (\textit{parallel}) \text{ and } (\textit{seq}) \\ &\longmapsto_{pc}^{1,1} \texttt{error} && \text{via } (\textit{join-error}) \end{aligned}$$

  Also $n_3 = m_3 = 0$ and $m_4 = m_4 = 2$. Hence $m_2 + m_4 = 2$ and $n_1 + n_3 = 1$. This case is the reason for the potential factor of two difference between $m_2 + m_4$ and $n_1 + n_3$.

(4) Suppose $S_1 \longmapsto_{pc}^{1,1} S_2$ via the rule (*future-id*), and $S_1 \longmapsto_{pc}^{1,0} S_3$ via the rule (*fork*). There are two sub-cases to consider. Suppose the two transitions operate on the same **future** expression. Then,

$$\begin{aligned} S_1 &= \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ V))\ M)\ ] \\ S_2 &= \mathcal{E}[\ M[x \leftarrow V]\ ] \\ S_3 &= (\mathbf{f\text{-}let}\ (p\ V)\ \mathcal{E}[\ M[x \leftarrow p]\ ]) \end{aligned}$$

For this case, choose $S_4 = S_2$:

$$\begin{aligned} S_2 &\longmapsto_{pc}^{0,0} S_4 \\ S_3 &\longmapsto_{pc}^{1,1} (\mathcal{E}[\ M[x \leftarrow p]\ ])[p \leftarrow V] \\ &= \mathcal{E}[\ M[x \leftarrow V]\ ] = S_4 \\ &\quad \text{since } p \notin FP(\mathcal{E}) \cup FP(M) \end{aligned}$$

The indices clearly satisfy the inequalities.

Alternatively, suppose that the transitions operate on different **future** expressions. Then

$$\begin{array}{rcl} S_1 & = & \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ (\mathbf{let}\ (y\ (\mathbf{future}\ V))\ N)\ ]))\ M)\ ] \\ S_2 & = & \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ N[y \leftarrow V]\ ]))\ M)\ ] \\ S_3 & = & (\mathbf{f\text{-}let}\ (p\ \mathcal{E}'[\ (\mathbf{let}\ (y\ (\mathbf{future}\ V))\ N)\ ]) \\ & & \qquad \mathcal{E}[\ M[x \leftarrow p]\ ]) \\ & & \text{where } p \notin FP(\mathcal{E}) \cup FP(M) \end{array}$$

Let

$$S_4 \ = \ (\mathbf{f\text{-}let}\ (p\ \mathcal{E}'[\ N[y \leftarrow V]\ ])\ \mathcal{E}[\ M[x \leftarrow p]\ ])$$

and we have

$$S_2 \ \longmapsto_{pc}^{1,0} \ S_4 \qquad \text{via } (\mathit{fork})$$

Also, since $\mathcal{E}'[\ (\mathbf{let}\ (y\ (\mathbf{future}\ V))\ N)\ ] \longmapsto_{pc}^{1,1} \mathcal{E}'[\ N[y \leftarrow V]\ ]$ via (*future-id*), we have (by using the rule (*parallel*) for compatible closure)

$$S_3 \ \longmapsto_{pc}^{1,1} \ S_4 \qquad \text{via } (\mathit{parallel})$$

The indices clearly satisfy the inequalities.

(5) Suppose $S_1 \longmapsto_{pc}^{1,1} S_2$ via the rule (*fork*), and $S_1 \longmapsto_{pc}^{1,0} S_3$ by the rule (*fork*). If the two transition operate on the same **future** expression, then $S_2 = S_3$. Otherwise,

$$\begin{array}{rcl} S_1 & = & \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ (\mathbf{let}\ (y\ (\mathbf{future}\ N'))\ N)\ ]))\ M)\ ] \\ S_1 \longmapsto_{pc}^{1,1} S_2 & = & (\mathbf{f\text{-}let}\ (p_1\ \mathcal{E}'[\ (\mathbf{let}\ (y\ (\mathbf{future}\ N'))\ N)\ ]) \\ & & \qquad \mathcal{E}[\ M[x \leftarrow p_1]\ ]) \\ S_1 \longmapsto_{pc}^{1,1} S_3 & = & (\mathbf{f\text{-}let}\ (p_2\ N') \\ & & \qquad \mathcal{E}[\ (\mathbf{let}\ (x\ (\mathbf{future}\ \mathcal{E}'[\ N[y \leftarrow p_2]\ ]))\ M)\ ]) \end{array}$$

where $p_1 \notin FP(\mathcal{E}) \cup FP(M)$ and $p_2 \notin FP(\mathcal{E}) \cup FP(\mathcal{E}') \cup FP(N) \cup FP(M)$. Let

$$\begin{array}{l} S_4 = (\mathbf{f\text{-}let}\ (p_2\ N') \\ \qquad\qquad (\mathbf{f\text{-}let}\ (p_1\ \mathcal{E}'[\ N[y \leftarrow p_2]\ ])\ \mathcal{E}[\ M[x \leftarrow p_1]\ ])) \end{array}$$

Then

$$\begin{array}{rcll} S_2 & \longmapsto_{pc}^{1,1} & (\mathbf{f\text{-}let}\ (p_1\ (\mathbf{f\text{-}let}\ (p_2\ N')\ \mathcal{E}'[\ N[y \leftarrow p_2]\ ])) & \text{via } (\mathit{fork}) \\ & & \qquad \mathcal{E}'[\ M[x \leftarrow p_1]\ ]) & \\ & \longmapsto_{pc}^{1,0} & S_4 & \text{via } (\mathit{lift}) \\ S_3 & \longmapsto_{pc}^{1,1} & S_4 & \text{via } (\mathit{fork}) \end{array}$$

and the indices clearly satisfy the inequalities.

(6) For the case where $S_1 \longmapsto^{1,1}_{pc} S_2$ via the rule (*join*), and $S_1 \longmapsto^{n,0}_{pc} S_3$ by the rule (*parallel*), we have

$$\begin{array}{rcll} S_1 & = & (\textbf{f-let}\ (p\ V)\ S) & \\ S_2 & = & S[p \leftarrow V] & \\ S_3 & = & (\textbf{f-let}\ (p\ V)\ S') & \text{where } S \longmapsto^{n,m}_{pc} S' \end{array}$$

Pick $S_4 = S'[p \leftarrow V]$, and $S_2 \longmapsto^{n,m}_{pc} S_4$ by the Placeholder Transparency Lemma (3.2), $S_3 \longmapsto^{1,1}_{pc} S_4$ via the rule (*join*), and the indices clearly satisfy the inequalities.

(7) Suppose $S_1 \longmapsto^{1,1}_{pc} S_2$ via the rule (*join-error*), and $S_1 \longmapsto^{n,0}_{pc} S_3$ by the rule (*parallel*). Then

$$\begin{array}{rcll} S_1 & = & (\textbf{f-let}\ (p\ \texttt{error})\ S) & \\ S_2 & = & \texttt{error} & \\ S_3 & = & (\textbf{f-let}\ (p\ \texttt{error})\ S') & \text{where } S \longmapsto^{n,m}_{pc} S' \end{array}$$

Pick $S_4 = \texttt{error}$. Then $S_2 \longmapsto^{0,0}_{pc} S_4$, $S_3 \longmapsto^{1,1}_{pc} S_4$ by the rule (*join-error*), and the indices clearly satisfy the inequalities.

(8) Suppose both the transitions $S_1 \longmapsto^{n_1,m_1}_{pc} S_2$ and $S_1 \longmapsto^{n_2,m_2}_{pc} S_3$ are via the rule (*parallel*). Then

$$\begin{array}{rcl} S_1 & = & (\textbf{f-let}\ (p\ S_1')\ S_1'') \\ S_2 & = & (\textbf{f-let}\ (p\ S_2')\ S_2'') \\ & & \text{where } S_1' \longmapsto^{a_1,m_1}_{pc} S_2',\ S_1'' \longmapsto^{c_1,d_1}_{pc} S_2'' \text{ and } n_1 = a_1 + c_1 \\ S_3 & = & (\textbf{f-let}\ (p\ S_3')\ S_3'') \\ & & \text{where } S_1' \longmapsto^{a_2,m_2}_{pc} S_3',\ S_1'' \longmapsto^{c_2,d_2}_{pc} S_3'' \text{ and } n_2 = a_2 + c_2 \end{array}$$

Since $a_1 \leq n_1$, $a_2 \leq n_2$ and S_1' is strictly smaller than S_1, by the inductive hypothesis there exists S_4' such that

$$\begin{array}{lll} S_2' & \longmapsto^{a_3,b_3}_{pc} & S_4' \\ S_3' & \longmapsto^{a_4,b_4}_{pc} & S_4' \end{array} \qquad \begin{array}{rcl} m_1 + b_3 & \leq & 2(a_2 + a_4) \\ m_2 + b_4 & \leq & 2(a_1 + a_3) \end{array}$$

Similarly, there exists S_4'' such that

$$\begin{array}{lll} S_2'' & \longmapsto^{c_3,d_3}_{pc} & S_4'' \\ S_3'' & \longmapsto^{c_4,d_4}_{pc} & S_4'' \end{array} \qquad \begin{array}{rcl} d_1 + d_3 & \leq & 2(c_2 + c_4) \\ d_2 + d_4 & \leq & 2(c_1 + c_3) \end{array}$$

Letting $S_4 = (\textbf{f-let}\ (p\ S_4')\ S_4'')$, we have

$$\begin{array}{lll} S_2 & \longmapsto^{n_3,m_3}_{pc} & S_4 \\ S_3 & \longmapsto^{n_4,m_4}_{pc} & S_4 \end{array}$$

where $n_3 = a_3 + c_3$, $m_3 = b_3$, $n_4 = a_4 + c_4$, $m_4 = b_4$. Furthermore,

$$\begin{array}{rcl} m_1 + m_3 & = & m_1 + b_3 \leq 2(a_2 + a_4) \leq 2(n_2 + n_4) \\ m_2 + m_4 & = & m_2 + b_4 \leq 2(a_1 + a_3) \leq 2(n_1 + n_3). \end{array}$$

(9) Suppose $S_1 \longmapsto_{pc}^{1,0} S_2$ via the rule (*lift*), and $S_1 \longmapsto_{pc}^{n_2,m_2} S_3$ by the rule (*parallel*). Then

$$\begin{array}{rrcll} & S_1 & = & (\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ S_a)\ S_b))\ S_c) & \\ S_1 \longmapsto_{pc}^{1,0} & S_2 & = & (\textbf{f-let}\ (p_1\ S_a)\ (\textbf{f-let}\ (p_2\ S_b)\ S_c)) & p_1 \notin FV(S_c) \\ S_1 \longmapsto_{pc}^{n_2,m_2} & S_3 & = & (\textbf{f-let}\ (p_2\ S_3')\ S_3'') & \\ & & & \text{where } (\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3' & \\ & & & \text{and } S_c \longmapsto_{pc}^{c,d} S_3'',\ n_2 = a + c & \end{array}$$

We proceed by sub-case analysis of the last step in $(\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3'$.

- There is no S_3' such that $(\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3'$ via any of the rules (*bind*), (*future-id*), (*car*), (*cdr*), (*if*), (*apply*) or (*fork*).
- Suppose $(\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3'$ via (*join*). Then

$$\begin{array}{rrcll} & S_1 & = & (\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ V)\ S_b))\ S_c) & \\ S_1 \longmapsto_{pc}^{1,0} & S_2 & = & (\textbf{f-let}\ (p_1\ V)\ (\textbf{f-let}\ (p_2\ S_b)\ S_c)) & p_1 \notin FV(S_c) \\ S_1 \longmapsto_{pc}^{n_2,1} & S_3 & = & (\textbf{f-let}\ (p_2\ S_b[p_1 \leftarrow V])\ S_3'') & n_2 = 1 + c \end{array}$$

  Let $S_4 = S_3$:

$$\begin{array}{rlr} S_2 \longmapsto_{pc}^{1,1} & (\textbf{f-let}\ (p_2\ S_b[p_1 \leftarrow V])\ S_c) & \text{via } (\textit{join}) \\ \longmapsto_{pc}^{c,d} & (\textbf{f-let}\ (p_2\ S_b[p_1 \leftarrow V])\ S_3'') = S_4 & \text{via } (\textit{parallel}) \end{array}$$

  The indices clearly satisfy the inequalities.
- Suppose $(\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3'$ via (*join-error*). Then

$$\begin{array}{rrcll} & S_1 & = & (\textbf{f-let}\ (p_2\ (\textbf{f-let}\ (p_1\ \texttt{error})\ S_b))\ S_c) & \\ S_1 \longmapsto_{pc}^{1,0} & S_2 & = & (\textbf{f-let}\ (p_1\ \texttt{error})\ (\textbf{f-let}\ (p_2\ S_b)\ S_c)) & p_1 \notin FV(S_c) \\ S_1 \longmapsto_{pc}^{n_2,0} & S_3 & = & (\textbf{f-let}\ (p_2\ \texttt{error})\ S_3'') & n_2 = 1 + c \end{array}$$

  Let $S_4 = \texttt{error}$:

$$\begin{array}{rl} S_2 \longmapsto_{pc}^{1,0} & S_4 \\ S_3 \longmapsto_{pc}^{1,0} & S_4 \end{array}$$

  The indices clearly satisfy the inequalities.
- The cases where $(\textbf{f-let}\ (p_1\ S_a)\ S_b) \longmapsto_{pc}^{a,m_2} S_3'$ via the rules (*reflexive*), (*transitive*) or (*parallel*) are straightforward.

∎

The Modified Diamond Lemma implies that all transition sequences exhibit the same observable behavior.

Lemma 3.5 (Consistency of Transitions) *Let P be a program. If $P \longmapsto^*_{pc} S$, where S is in normal form, then*

1. *For all normal form S' such that $P \longmapsto^*_{pc} S'$, $S' = S$.*
2. *It is impossible that for all $i \in \mathbf{N}$ there exists $S_i \in State_{pc}$ and $n_i, m_i \in \mathbf{N}$ such that $m_i > 0$, $P = S_0$ and $S_i \longmapsto^{n_i,m_i}_{pc} S_{i+1}$.*

Proof:

1. Suppose $P \longmapsto^*_{pc} S$ and $P \longmapsto^*_{pc} S'$, where S, S' are in normal form. By the Modified Diamond Lemma (3.4), there exists some S_4 such that $S \longmapsto^*_{pc} S_4$ and $S' \longmapsto^*_{pc} S_4$. But since S, S' are in normal form, we have that $S = S_4 = S'$.
2. We prove part 2 by contradiction. Assume that $P \longmapsto^{n,m}_{pc} S$ where S is in normal form, and that there exists some sequence of states $S_i \in State_{pc}$ and $n_i, m_i \in \mathbf{N}$ such that $m_i > 0$, $P = S_0$ and $S_i \longmapsto^{n_i,m_i}_{pc} S_{i+1}$.

   Pick an integer $k > 2n$. Then $P \longmapsto^{a,b}_{pc} S_{k+1}$, where $a = \Sigma^{i=k}_{i=0} n_i$ and $b = \Sigma^{i=k}_{i=0} m_i$. By the Modified Diamond Lemma (3.4), $S_{k+1} \longmapsto^{c,d}_{pc} S$ for some $c, d \in \mathbf{N}$, with $2n \geq b + d$. But $b + d \geq b = \Sigma^{i=k}_{i=0} m_i \geq k + 1 > k$, since $m_i > 0$, producing the contradiction $2n > k > 2n$.

∎

Since all transition sequences for a given program exhibit the same observable behavior, the evaluator $eval_{pc}$ for the $P(C)$-machine is a well-defined function.

Theorem 3.6 (Consistency of $eval_{pc}$) *The relation $eval_{pc}$ is a function.*

Proof: Follows from Lemma 3.5. ∎

3.3 Correctness of the $P(C)$-Machine

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.

Lemma 3.7 *Suppose $S \longmapsto_c S'$, for $S, S' \in State_c$. Then $S \longmapsto^{1,1}_{pc} S'$.*

The correspondence between transitions of the two machines implies that their respective evaluators are equivalent.

Theorem 3.8 (Correctness of $eval_{pc}$) $eval_c = eval_{pc}$.

Proof: Let P be any program. We proceed by case analysis of the definition of $eval_c$:

- Suppose $eval_c(P) = unload_c[V]$ because

$$P \equiv S_0 \longmapsto_c S_1 \longmapsto_c \cdots \longmapsto_c S_n \equiv V$$

Then $P \longmapsto_{pc}^{n,n} V$, and hence $eval_{pc}(P) = unload_{pc}[V] = unload_c[V]$.

- Alternatively, suppose $eval_c(P) = \texttt{error}$ because

$$P \equiv S_0 \longmapsto_c S_1 \longmapsto_c \cdots \longmapsto_c S_n \equiv \texttt{error}$$

Then $P \longmapsto_{pc}^{n,n} \texttt{error}$, and hence $eval_{pc}(P) = \texttt{error}$.

- Finally, suppose $eval_c(P) = \perp$ via the infinite sequence

$$P \equiv S_0 \longmapsto_c S_1 \longmapsto_c \cdots \longmapsto_c S_n \longmapsto_c \cdots$$

Then we have that

$$P \equiv S_0 \longmapsto_{pc}^{1,1} S_1 \longmapsto_{pc}^{1,1} \cdots \longmapsto_{pc}^{1,1} S_n \longmapsto_{pc}^{1,1} \cdots$$

and hence $eval_{pc}(P) = \perp$.

Hence, for all programs P, $eval_c(P) = eval_{pc}(P)$. ∎

The equivalence of the two evaluators implies that $eval_{pc}$ is defined for all programs.

Theorem 3.9 (Completeness of $eval_{pc}$) *The relation $eval_{pc}$ is a total function.*

In summary, 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 Placeholder Object Semantics

The $P(C)$-machine specifies the parallel execution behavior of programs with **future**s at a fairly high level. It thus hides certain low-level operations that are required in the implementation of **future**s. In particular, implementations typically represent placeholders using *placeholder objects*, and avoid the need for an expensive substitution operation on placeholders (cmp. *join*) by imperative updating these placeholder objects instead. This technique requires *touch* operations within placeholder-strict primitives to dereference placeholder objects whenever necessary. Since we plan to use the semantics of **future** to prove the correctness of an algorithm for removing redundant *touch*es, we reformulate the $P(C)$-machine to expose these placeholder objects and the associated *touch* operations. The result is the $P(C_{ph})$-machine.

Evaluator:

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

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

Data Specifications:

$$\begin{array}{lllll} S & \in & State_{pc'} & ::= & M \mid \texttt{error} \mid (\textbf{f-let}\ (p\ S)\ S) & \text{(States)} \\ M & \in & \Lambda_{pc'} & ::= & V \mid (\textbf{let}\ (x\ V)\ M) \mid \ldots & \text{(As for } \Lambda_c\text{)} \\ V & \in & Value_{pc'} & ::= & PValue_{pc'} \mid \langle \textbf{ph}\ p\ \circ \rangle \mid \langle \textbf{ph}\ p\ V \rangle & \text{(Run-time Values)} \\ & & PValue_{pc'} & ::= & c \mid x \mid Cl_{pc'} \mid Pair_{pc'} & \text{(Proper Values)} \\ & & Cl_{pc'} & ::= & (\lambda x.\, M) & \text{(Closures)} \\ & & Pair_{pc'} & ::= & (\textbf{cons}\ V\ V) & \text{(Pairs)} \end{array}$$

Auxiliary Functions:

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

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

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

$$\begin{array}{rcl} M[p := V] & = & M \text{ with all free occurrences of } \langle \textbf{ph}\ p\ \circ \rangle \text{ replaced by } \langle \textbf{ph}\ p\ V \rangle \\ \texttt{error}[p := V] & = & \texttt{error} \\ (\textbf{f-let}\ (p'\ S_1)\ S_2)[p := V] & = & \begin{cases} (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2) & \text{if } p = p' \\ (\textbf{f-let}\ (p'\ S_1[p := V])\ S_2[p := V]) & \text{if } p \neq p' \end{cases} \end{array}$$

FIGURE 5: The parallel $P(C_{ph})$-machine: Evaluator and Data Specifications

4.1 Specification of the $P(C_{ph})$-machine

The state space of the $P(C_{ph})$-machine is a minor revision of that of the $P(C)$-machine: see Figure 5. Instead of plain variables, placeholders are now tagged objects. An *undetermined placeholder object* (written $\langle \textbf{ph}\ p\ \circ \rangle$) is created to represent the result of each parallel task (cmp. (*fork*)). The symbol $\circ$ indicates that the result of the computation is unknown, and p is used to distinguish distinct placeholder objects. When the computation associated with the placeholder object terminates, producing a value V, then the undetermined placeholder object is replaced by the *determined placeholder object* $\langle \textbf{ph}\ p\ V \rangle$.[4]

The correctness of this technique requires that a determined placeholder object $\langle \textbf{ph}\ p\ V \rangle$ representing the value V is observationally equivalent to the value V itself. To ensure

[4]The explicit replacement of undetermined placeholders by determined placeholders is still unrealistic, but suffices for our purposes.

Transition Rules:

$$\mathcal{E}[\,(\mathbf{let}\ (x\ V)\ M)\,] \longmapsto^{1,1}_{pc'} \mathcal{E}[\,M[x \leftarrow V]\,] \qquad (\textit{bind})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{future}\, V))\ M)\,] \longmapsto^{1,1}_{pc'} \mathcal{E}[\,M[x \leftarrow V]\,] \qquad (\textit{future-id})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{car}\ V))\ M)\,] \longmapsto^{1,1}_{pc'} \begin{cases} \mathcal{E}[\,M[x \leftarrow V_1]\,] & \text{if } \mathit{touch}_{pc'}[V] = (\mathbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } \mathit{touch}_{pc'}[V] \notin \mathit{Pair}_{pc'} \cup \{\circ\} \end{cases} \qquad (\textit{car})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{cdr}\ V))\ M)\,] \longmapsto^{1,1}_{pc'} \begin{cases} \mathcal{E}[\,M[x \leftarrow V_2]\,] & \text{if } \mathit{touch}_{pc'}[V] = (\mathbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } \mathit{touch}_{pc'}[V] \notin \mathit{Pair}_{pc'} \cup \{\circ\} \end{cases} \qquad (\textit{cdr})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{if}\ V\ M_1\ M_2))\ M)\,] \longmapsto^{1,1}_{pc'} \begin{cases} \mathcal{E}[\,(\mathbf{let}\ (x\ M_1)\ M)\,] & \text{if } \mathit{touch}_{pc'}[V] \notin \{\texttt{nil}, \circ\} \\ \mathcal{E}[\,(\mathbf{let}\ (x\ M_2)\ M)\,] & \text{if } \mathit{touch}_{pc'}[V] = \texttt{nil} \end{cases} \qquad (\textit{if})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{apply}\ V_1\ V_2))\ M)\,] \longmapsto^{1,1}_{pc'} \begin{cases} \mathcal{E}[\,(\mathbf{let}\ (x\ N[y \leftarrow V_2])\ M)\,] & \text{if } \mathit{touch}_{pc'}[V_1] = (\lambda y.\, N) \\ \texttt{error} & \text{if } \mathit{touch}_{pc'}[V_1] \notin \mathit{Cl}_{pc'} \cup \{\circ\} \end{cases} \qquad (\textit{apply})$$

$$\mathcal{E}[\,(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M)\,] \longmapsto^{1,0}_{pc'} (\mathbf{f\text{-}let}\ (p\ N)\ \mathcal{E}[\,M[x \leftarrow \langle \mathbf{ph}\ p\ \circ \rangle]\,]) \quad p \notin \mathit{FP}(\mathcal{E}) \cup \mathit{FP}(M) \qquad (\textit{fork})$$

$$(\mathbf{f\text{-}let}\ (p\ V)\ S) \longmapsto^{1,1}_{pc'} S[p := V] \qquad (\textit{join})$$

$$(\mathbf{f\text{-}let}\ (p\ \texttt{error})\ S) \longmapsto^{1,1}_{pc'} \texttt{error} \qquad (\textit{join-error})$$

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

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

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

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

FIGURE 6: The parallel $P(C_{ph})$-machine: Transition Rules

this behavior, the primitives **car**, **cdr**, **if** and **apply** perform *touch* operations on their placeholder-strict arguments. A *touch* operation behaves as the identity operation on proper values, but when applied to a placeholder object, the *touch* operation dereferences the placeholder object to retrieve the value that it represents.

An occurrence of an undetermined placeholder object $\langle \mathbf{ph}\ p\ \circ \rangle$ is *free* if p is not bound by an enclosing **flet** expression, and a state S is *closed* if it does not contain any free variables or free undetermined placeholder objects.

4.2 Correctness of the $P(C_{ph})$-machine

The state space of the $P(C_{ph})$-machine is identical to that of the $P(C)$-machine, apart from the use of placeholder objects. Therefore, each $P(C_{ph})$-machine state corresponds directly to a $P(C)$-machine state. The translation function Θ from $P(C_{ph})$-states to $P(C)$-states simply replaces determined placeholder objects by the appropriate value, and replaces undetermined placeholder objects by the corresponding placeholder variable.

Definition 4.1. (Θ)

$$\begin{array}{rcl}
\Theta : \mathit{State}_{pc'} & \longrightarrow & \mathit{State}_{pc} \\
\Theta[\![\langle \mathbf{ph}\ p\ \circ \rangle]\!] & = & p \\
\Theta[\![\langle \mathbf{ph}\ p\ V \rangle]\!] & = & \Theta[\![V]\!] \\
\Theta[\![c]\!] & = & c \\
\Theta[\![x]\!] & = & x \\
\Theta[\![(\lambda x.\, M)]\!] & = & (\lambda x.\, \Theta[\![M]\!]) \\
\cdots & = & \cdots \quad \text{similar clauses for other } \Lambda_{pc'} \text{ terms} \\
\Theta[\![\texttt{error}]\!] & = & \texttt{error} \\
\Theta[\![(\textbf{f-let}\ (p\ S_1)\ S_1)]\!] & = & (\textbf{f-let}\ (p\ \Theta[\![S_1]\!])\ \Theta[\![S_2]\!])
\end{array}$$

Θ extends in a natural manner to evaluation contexts: $\Theta[\![\ [\ ]\ ]\!] = [\ ], \ldots$ ∎

The function Θ is a *bisimulation relation*, *i.e.*, every transition of the $P(C_{ph})$-machine corresponds to a transition of the $P(C)$-machine, and vice versa. The proof of this property relies on the following lemma concerning the translation Θ.

Lemma 4.2

1. *For all* $V \in \mathit{Value}_{pc'}$*, if* $\mathit{touch}_{pc'}[V] \neq \circ$ *then* $\Theta[\![\mathit{touch}_{pc'}[V]]\!] = \Theta[\![V]\!]$.
2. *For all* $V \in \mathit{Value}_{pc'}$*,* $\mathit{unload}_{pc'}[V] = \mathit{unload}_{pc}[\Theta[\![V]\!]]$.
3. $\Theta[\![\mathcal{E}]\!]\ [\Theta[\![M]\!]] = \Theta[\![\mathcal{E}[\ M\ ]]\!]$
4. $\Theta[\![M]\!][x \leftarrow \Theta[\![V]\!]] = \Theta[\![M[x \leftarrow V]]\!]$

Proof: The four facts follow from straightforward structural induction. ∎

Theorem 4.3 (Bisimulation Theorem) *Let* $S_1 \in \mathit{State}_{pc}$ *and* $S_1' \in \mathit{State}_{pc'}$ *such that* $\Theta[\![S_1']\!] = S_1$.

1. *If* $S_1' \longmapsto_{pc'}^{n,m} S_2'$*, then* $S_1 \longmapsto_{pc}^{n,m} \Theta[\![S_2']\!]$.
2. *If* $S_1 \longmapsto_{pc}^{n,m} S_2$*, then there exists* $S_2' \in \mathit{State}_{pc'}$ *such that* $S_1' \longmapsto_{pc'}^{n,m} S_2'$ *and* $\Theta[\![S_2']\!] = S_2$.

Proof:

1. The proof of the first part proceeds by lexicographic induction on n and on the size of S'_1, and by case analysis of the last step in $S'_1 \longmapsto^{n,m}_{pc'} S'_2$. The interesting cases concern the placeholder-strict operations. We examine the case for the transition rule (*car*) in detail.

   Suppose $S'_1 \longmapsto^{1,1}_{pc'} S'_2$ via the transition rule (*car*). Then

$$\begin{array}{rcl} S'_1 & = & \mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{car}\ V))\ M)\ ] \\ S_1 & = & \Theta[\![\mathcal{E}]\!]\ [\ (\textbf{let}\ (x\ (\textbf{car}\ \Theta[\![V]\!]))\ \Theta[\![M]\!])\ ] \end{array}$$

   We consider three cases for V separately:

   - Suppose $touch_{pc'}[V] = (\textbf{cons}\ V_1\ V_2)$. Then

$$\begin{array}{rcl} S'_2 & = & \mathcal{E}[\ M[x \leftarrow V_1]\ ] \\ \Theta[\![V]\!] & = & (\textbf{cons}\ \Theta[\![V_1]\!]\ \Theta[\![V_2]\!]) \\ S_1 & \longmapsto^{1,1}_{pc} & \Theta[\![\mathcal{E}]\!]\ [\ \Theta[\![M]\!][x \leftarrow \Theta[\![V_1]\!]]\ ] \\ & = & \Theta[\![\mathcal{E}]\!]\ [\ \Theta[\![M[x \leftarrow V_1]]\!]\ ] \\ & = & \Theta[\![\mathcal{E}[\ M[x \leftarrow V_1]\ ]]\!] \\ & = & \Theta[\![S'_2]\!] \end{array}$$

   - Suppose $touch_{pc'}[V] \notin Pair_{pc'} \cup \{\circ\}$, *i.e.*, $touch_{pc'}[V] \in Const \cup Vars \cup Cl_{pc'}$. Then

$$\begin{array}{rcl} S'_2 & = & \texttt{error} \\ \Theta[\![V]\!] & \in & Const \cup Vars \cup \{(\lambda x.\, M)\} \\ S_1 & \longmapsto^{1,1}_{pc} & \texttt{error} \end{array}$$

   - The case where $touch_{pc'}[V] = \circ$ is impossible, since S'_1 would not be a (*car*)-redex.

   Thus the case for the transition rule (*car*) holds. The analysis of the transition rules (*cdr*), (*if*) and (*apply*) is similar. We consider in detail the case for the (*parallel*) rule, in order to justify our use of lexicographic induction.

   Suppose $S'_1 \longmapsto^{n,m}_{pc} S'_2$ via the rule (*parallel*). Then there exists some states $X'_1, Y'_1, X'_2, Y'_2 \in State_{pc'}$ such that:

$$\left.\begin{array}{rcl} S'_1 & = & (\textbf{f-let}\ (p'\ X'_1)\ Y'_1) \\ S'_2 & = & (\textbf{f-let}\ (p'\ X'_2)\ Y'_2) \end{array}\right. n \qquad \begin{array}{rcl} X'_1 & \longmapsto^{a,m}_{pc'} & X'_2 \\ Y'_1 & \longmapsto^{c,d}_{pc'} & Y'_2 \end{array}$$

   where $n = a + c$. Since X'_1 and Y'_1 are substates of S'_1, by the inductive hypothesis, we have:

$$\begin{array}{rcl} \Theta[\![X'_1]\!] & \longmapsto^{a,m}_{pc} & \Theta[\![X'_2]\!] \\ \Theta[\![Y'_1]\!] & \longmapsto^{c,d}_{pc} & \Theta[\![Y'_2]\!] \end{array}$$

Hence:

$$\begin{array}{rcll} S_1 & = & (\textbf{f-let}\ (p'\ \Theta[\![X_1']\!])\ \Theta[\![Y_1']\!]) & \\ & \longmapsto_{pc}^{n,m} & (\textbf{f-let}\ (p'\ \Theta[\![X_2']\!])\ \Theta[\![Y_2']\!]) & \text{via } (\mathit{parallel}) \\ & = & \Theta[\![S_2']\!] & \end{array}$$

The proofs for the remaining cases are straightforward.

2. We prove the second part by lexicographic induction on n and on the size of S_1, and by case analysis of the last step in the transition $S_1 \longmapsto_{pc}^{n,m} S_2$. Again, the interesting cases concern placeholder-strict operations, and we consider the transition rule (*car*) in detail.

   Suppose $S_1 \longmapsto_{pc}^{1,1} S_2$ via the transition rule (*car*). Then

   $$S_1 \ = \ \mathcal{E}[\ (\textbf{let}\ (x\ (\textbf{car}\ V))\ M)\ ]$$

   An examination of the definition of Θ shows that S_1' must be

   $$\mathcal{E}'[\ (\textbf{let}\ (x\ (\textbf{car}\ V'))\ M')\ ]$$

   where $\Theta[\![\mathcal{E}']\!] = \mathcal{E}, \Theta[\![V']\!] = V$ and $\Theta[\![M']\!] = M$. We consider three cases for V:

   - Suppose $V = (\textbf{cons}\ V_1\ V_2)$. It follows that $S_2 = \mathcal{E}[\ M[x \leftarrow V_1]\ ]$ and $V' = (\textbf{cons}\ V_1'\ V_2')$, where $\Theta[\![V_i']\!] = V_i$ for $i = 1, 2$. Choosing $S_2' = \mathcal{E}'[\ M'[x \leftarrow V_1']\ ]$, we have that $S_1' \longmapsto_{pc'}^{1,1} S_2'$ and $\Theta[\![S_2']\!] = S_2$.
   - Suppose V is neither a pair nor a placeholder. Then $V \in \mathit{Const} \cup \mathit{Vars} \cup \{(\lambda x.\, M)\}$, and hence $V' \in \mathit{Const} \cup \mathit{Vars} \cup \{(\lambda x.\, M)\}$. Therefore $S_2 = S_2' = \texttt{error}$, and the conclusion holds for case, too.
   - The case where V is a placeholder is impossible, since S_1 would not be a (*car*)-redex.

   Thus the theorem holds if $S_1 \longmapsto_{pc}^{n,m} S_2$ via the transition rule (*car*). The analysis of the transition rules (*cdr*), (*if*) and (*apply*) is similar. We also examine in detail the case for the (*parallel*) rule, in order to show why we choose to use lexicographic induction.

   Suppose $S_1 \longmapsto_{pc}^{n,m} S_2$ via the rule (*parallel*). Then there exists some states $X_1, Y_1, X_2, Y_2 \in \mathit{State}_{pc}$ such that:

   $$\begin{array}{rcl} S_1 & = & (\textbf{f-let}\ (p\ X_1)\ Y_1) \\ S_2 & = & (\textbf{f-let}\ (p\ X_2)\ Y_2) \end{array} \qquad \begin{array}{rcl} X_1 & \longmapsto_{pc}^{a,m} & X_2 \\ Y_1 & \longmapsto_{pc}^{c,d} & Y_2 \end{array}$$

   with $n = a + c$. Examining the definition of Θ shows that S_1' must be:

   $$(\textbf{f-let}\ (p\ X_1')\ Y_1')$$

   for some $X_1', Y_1' \in \mathit{State}_{pc'}$ such that $\Theta[\![X_1']\!] = X_1$ and $\Theta[\![Y_1']\!] = Y_1$. Since X_1 and Y_1 are substates of S_1, by induction there exists states $X_2', Y_2' \in \mathit{State}_{pc'}$ with:

   $$\begin{array}{rcl} X_1' & \longmapsto_{pc}^{a,m} & X_2' \\ Y_1' & \longmapsto_{pc}^{c,d} & Y_2' \end{array} \qquad \begin{array}{rcl} \Theta[\![X_2']\!] & = & X_2 \\ \Theta[\![Y_2']\!] & = & Y_2 \end{array}$$

Taking $S'_2 = (\textbf{f-let}\ (p\ X'_2)\ Y'_2)$, we have that $\Theta[\![S'_2]\!] = S_2$ and that:

$$\begin{array}{rcl} S'_1 & = & (\textbf{f-let}\ (p\ X'_1)\ Y'_1) \\ & \longmapsto^{n,m}_{pc} & (\textbf{f-let}\ (p\ X'_2)\ Y'_2) \\ & = & S'_2 \end{array}$$

The proofs for the remaining cases are straightforward.

∎

The Bisimulation Theorem implies the equivalence of the $P(C_{ph})$-machine and the $P(C)$-machine.

Theorem 4.4 (Correctness of $eval_{pc'}$**)** $eval_{pc'} = eval_{pc}$.

Proof: Both directions of the theorem are straightforward consequences of the Bisimulation Theorem. ∎

Since the $P(C)$-machine is consistent and complete (from Theorems 3.6 and 3.9), the equivalence of machines implies that the $P(C_{ph})$-machine is also consistent and complete.

Theorem 4.5 (Consistency and Completeness of $eval_{pc'}$**)** *The relation* $eval_{pc'}$ *is a total function.*

In summary, the $P(C_{ph})$-machine is a correct implementation of the $P(C)$-machine in that both define the same semantics for the source language. Hence the use of placeholder objects, combined with *touch* operations for placeholder-strict primitives, is a valid technique for coordinating parallel tasks.

5 The Low-Level Operational Semantics

Since optimizations heavily rely on static information about the values that variables can assume, the $P(C_{ph})$-machine is ill-suited for correctness proofs of appropriate analysis algorithms.[5] The states of the $P(C_{ph})$-machine contain no binding information relating program variables and values. Instead, the machine relies on substitution for making progress. To address this problem, we refine the $P(C_{ph})$-machine to the $P(CEK)$-machine (see Figures 7 and 8) using standard techniques [5, 9].

5.1 Specification of the $P(CEK)$-machine

The substitution operation is replaced by an *environment* in the usual manner. An environment E is a finite 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. Using function notation, we write $E(x)$ to denote the value bound to x

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

Evaluator:

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

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

Data Specifications:

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

Auxiliary Functions:

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

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

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

FIGURE 7: The $P(CEK)$-machine: Evaluator and Data Specifications

in E, $dom(E)$ to denote the domain of E, and $E|_{FV[(\lambda x.M)]}$ to denote the restriction of E to the free variables of $(\lambda x.\, M)$.

An evaluation context, which represents the control stack, is now represented by a *continuation*. A continuation consists of a sequence of *activation records*, which are similar

Transition Rules:

$$
\begin{array}{rcll}
\langle(\mathbf{let}\ (x\ c)\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow c], K\rangle & (\mathit{bind\text{-}const}) \\
\langle(\mathbf{let}\ (x\ y)\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow E(y)], K\rangle & (\mathit{bind\text{-}var}) \\
\langle(\mathbf{let}\ (x\ (\lambda y.\, N))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow \langle(\lambda y.\, N), E'\rangle], K\rangle & (\mathit{bind\text{-}lam}) \\
 & & \text{where } E' = E|_{FV[(\lambda y.\, N)]} & \\
\langle(\mathbf{let}\ (x\ (\mathbf{cons}\ y\ z))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E[x \leftarrow (\mathbf{cons}\ E(y)\ E(z))], K\rangle & (\mathit{bind\text{-}cons}) \\
\langle x, E, \langle \mathbf{ar}\ y, M, E'\rangle.K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E'[y \leftarrow E(x)], K\rangle & (\mathit{return}) \\
\langle(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & & \\
\multicolumn{3}{l}{\quad\left\{\begin{array}{ll} \langle M, E[x \leftarrow V_1], K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = (\mathbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \mathit{Pair}_{pcek} \cup \{\circ\} \end{array}\right.} & (\mathit{car}) \\
\langle(\mathbf{let}\ (x\ (\mathbf{cdr}\ y))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \text{analogous to } \mathit{car} & (\mathit{cdr}) \\
\langle(\mathbf{let}\ (x\ (\mathbf{if}\ y\ M_1\ M_2))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & & \\
\multicolumn{3}{l}{\quad\left\{\begin{array}{ll} \langle M_1, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \{\texttt{nil}, \circ\} \\ \langle M_2, E, \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = \texttt{nil} \end{array}\right.} & (\mathit{if}) \\
\langle(\mathbf{let}\ (x\ (\mathbf{apply}\ y\ z))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & & \\
\multicolumn{3}{l}{\quad\left\{\begin{array}{ll} \langle N, E'[x' \leftarrow E(z)], \langle \mathbf{ar}\ x, M, E\rangle.K\rangle & \text{if } \mathit{touch}_{pcek}[E(y)] = \langle(\lambda x'.\, N), E'\rangle \\ \texttt{error} & \text{if } \mathit{touch}_{pcek}[E(y)] \notin \mathit{Cl}_{pcek} \cup \{\circ\} \end{array}\right.} & (\mathit{apply}) \\
\langle(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M), E, K\rangle & \longmapsto^{1,1}_{pcek} & \langle N, E, \langle \mathbf{ar}\dagger\ x, M, E\rangle.K\rangle & (\mathit{future}) \\
\langle x, E, \langle \mathbf{ar}\dagger\ y, M, E'\rangle.K\rangle & \longmapsto^{1,1}_{pcek} & \langle M, E'[y \leftarrow E(x)], K\rangle & (\mathit{future\text{-}id}) \\
\langle M, E, K_1.\langle \mathbf{ar}\dagger\ x, N, E'\rangle.K_2\rangle & \longmapsto^{1,0}_{pcek} & & \\
\multicolumn{3}{l}{\quad(\mathbf{f\text{-}let}\ (p\ \langle M, E, K_1\rangle)\ \langle N, E'[x \leftarrow \langle \mathbf{ph}\ p\ \circ\rangle], K_2\rangle) \quad p \notin FP(E') \cup FP(K_2)} & (\mathit{fork}) \\
(\mathbf{f\text{-}let}\ (p\ \langle x, E, \epsilon\rangle)\ S) & \longmapsto^{1,1}_{pcek} & S[p := E(x)] & (\mathit{join}) \\
(\mathbf{f\text{-}let}\ (p\ \texttt{error})\ S) & \longmapsto^{1,1}_{pcek} & \texttt{error} & (\mathit{join\text{-}error}) \\
(\mathbf{f\text{-}let}\ (p_2\ (\mathbf{f\text{-}let}\ (p_1\ S_1)\ S_2))\ S_2) & \longmapsto^{1,1}_{pcek} & (\mathbf{f\text{-}let}\ (p_1\ S_1)\ (\mathbf{f\text{-}let}\ (p_2\ S_2)\ S_3)) & (\mathit{lift}) \\
 & & p_1 \notin FP(S_3) & \\
(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2) & \longmapsto^{a+c,b}_{pcek} & (\mathbf{f\text{-}let}\ (p\ S_1')\ S_2') & (\mathit{parallel}) \\
 & & \text{if } S_1 \longmapsto^{a,b}_{pcek} S_1' \longmapsto^{c,d}_{pcek} S_2' & \\
S & \longmapsto^{0,0}_{pcek} & S & (\mathit{reflexive}) \\
S & \longmapsto^{a+c,b+d}_{pcek} & S'' & (\mathit{transitive}) \\
 & & \text{if } S \longmapsto^{a,b}_{pcek} S',\ S' \longmapsto^{c,d}_{pcek} S'',\ a, c > 0 &
\end{array}
$$

FIGURE 8: The $P(CEK)$-machine: Transition Rules

to closures. Evaluation of an **apply** expression produces an activation record of the form $\langle \mathbf{ar}\ x, M, E\rangle$ to record the calling context. Evaluation of a **future** expression produces a *tagged* activation record of the form $\langle \mathbf{ar}\dagger\ x, M, E\rangle$. A tagged activation record represents a point where the continuation can be split into separate tasks (cmp. (*fork*)).

The transition relation $\longmapsto_{pcek}$ of the $P(CEK)$-machine is a 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 $n, m \in \mathbf{N}$. A state $S \in State_{pcek}$ is in *normal-form* if there is no non-trivial transition from S. States of the form $\langle x, E, \epsilon \rangle$ or `error` are called *final states*, and a state *blocked* if it is in normal form but not a final state.

5.2 Correctness of the $P(CEK)$-machine

The proof of correctness of the $P(CEK)$-machine uses standard proof techniques from [5], appropriately modified to account for parallel evaluation.

Theorem 5.1 (Correctness of $eval_{pcek}$) $eval_{pcek} = eval_{pc'}$

Proof: The proof of this theorem is included in appendix A. ∎

The equivalence of machines implies that the $P(CEK)$-machine is also consistent and complete, since the $P(C_{ph})$-machine is consistent and complete.

Corollary 5.2 (Consistency and Completeness of $eval_{pcek}$) *The relation $eval_{pcek}$ is a total function.*

The four evaluators we present, $eval_c$, $eval_{pc}$, $eval_{pc'}$ and $eval_{pcek}$, corresponding to the four abstract machines, are provably equivalent. The parallel execution behavior of programs on the $P(CEK)$-machine is indistinguishable from their behavior on the C-machine, showing that the $P(CEK)$-machine is a correct implementation of **future**s according to the definitional semantics of $eval_c$.

6 Related Work

The literature on programming languages contains a number of descriptions of the semantics of parallel Scheme-like languages. The only one that directly deals with parallelism based on transparent annotations is Moreau's Ph.D. thesis [20]. Moreau studies the functional core of Scheme extended with **pcall** (a construct 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, and to derive a reasonably efficient implementation. The semantics is an extension of Felleisen and Friedman's control calculus [5]; the implementation is a parallel version of the CESK machine [6] that implements placeholders as globally accessible reference cells. The equivalence proof establishes that both evaluators define the same observational equivalence relation via the construction of a number of intermediate calculi and machines. It is far more complicated than our diamond and bisimulation techniques, possibly due to the inclusion of continuations.

Independently, Reppy [21] and Leroy [17] 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. The first level, also a program rewriting system in the tradition of Felleisen and Friedman [5, 6], accounts for the sequential behavior; the second-level specifies the behavior of *sets* of parallel tasks and the task communication mechanisms. Reppy uses the semantics to prove a type soundness theorem for the extended semantics; he does not construct a low-level semantics that can serve as the basis of an implementation or a program analysis tool. 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. No attempt is made to exploit the semantics for the derivation of an analysis algorithm or a compiler optimization.

Jaganathan and Weeks [14] define an operational semantics for a simple function language extended with the **spawn** construct. They also show how the **future** annotation can be implemented using **spawn**. Since their primary goal is the derivation of a semantically well-founded abstract interpretation (in the spirit of Cousot and Cousot [2]), they extend Deutsch's transition semantics [3] to their language. The transition semantics requires the assignments of a unique label to each sub-expression of a program and expresses computation as the movement of a token from label to label. An auxiliary label on each sub-expression is used to collect information about the values of the expression. The semantics is well-suited for deriving traditional abstract interpretations, but is inappropriate for specifying a user-level semantics.

Finally, Wand [24] 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. Such a derivation consists of a staging process that separates the run-time portion of the semantic mapping from the compile-time portion. To prove the correctness of the compiler, it suffices to prove that the "composition" of the two functions yields the semantic mapping. 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 (a stronger version 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.

7 Applications of the Semantics

We believe that the low-level semantics of the $P(CEK)$-machine is a sound basis for further research into the optimization and implementation of **future**s. We have used the semantics to derive a provably-correct program optimization algorithm that removes redundant *touch* operations from programs [7, 8]. Experiments with the Gambit compiler [4] show that this optimization substantially reduces program execution times on a standard set of benchmarks.

Another important application of the low-level semantics is as a basis from which to derive a parallel and distributed implementation of Scheme with **future**s. We plan to reformulate the low-level semantics in a *message-passing framework* and to implement it in the near future.

A Correctness Proof for the $P(CEK)$-Machine

Each $P(CEK)$-machine state corresponds to a $P(C_{ph})$-machine state according to the translation Φ_s: see Figure 9. For example, the $P(C_{ph})$-machine state $\langle C, E, K\rangle$ corresponds to the $P(C_{ph})$-machine state $\mathcal{E}[\ M\ ]$, where $\mathcal{E} = \Phi_k[\![K]\!]$ is the evaluation context represented by the continuation K, and the term $M = \Phi_m[C, E]$ is the result of applying the substitutions recorded in E to the term C.

The transition relation of the $P(CEK)$-machine is simply a reformulation of the transition relation of the $P(C_{ph})$-machine. Therefore, each transition of the $P(CEK)$-machine corresponds to a transition of the $P(C_{ph})$-machine, although the number of steps involved in corresponding transitions is not identical. In particular, the $P(C_{ph})$-machine assumes an automatic division of each sequential term into an evaluation context and a redex. Since evaluation contexts may include **future** contexts of the form $(\mathbf{let}\ (x\ (\mathbf{future}\ [\ ]))\ M)$, the $P(C_{ph})$-machine can immediately initiate evaluation inside a **future** context. In contrast, the lower-level $P(CEK)$-machine must first convert all enclosing **future** contexts into tagged activation records via the transition rule (*future*). Thus, certain mandatory transitions on the $P(CEK)$-machine correspond to an indentity transition on the $P(C_{ph})$-machine. The following theorem formalizes this correspondence, and furthermore proves that there is no infinite sequence of mandatory $P(CEK)$-machine transitions that corresponds to an identity transition on the $P(C_{ph})$-machine.

$$
\begin{array}{rcl}
\Phi_m : \Lambda_a \times Env_{pcek} & \longrightarrow & \Lambda_{pc'} \\
\Phi_m[M, \{x_1 \leftarrow V_1, \ldots, x_n \leftarrow V_n\}] & = & M[x_1 \leftarrow \Phi_v[V_1]] \ldots [x_n \leftarrow \Phi_v[V_n]] \\
\\
\Phi_v : Value_{pcek} & \longrightarrow & Value_{pc'} \\
\Phi_v[c] & = & c \\
\Phi_v[\langle(\lambda x.\, M), E\rangle] & = & \Phi_m[(\lambda x.\, M), E] \\
\Phi_v[(\mathbf{cons}\ V_1\ V_2)] & = & (\mathbf{cons}\ \Phi_v[V_1]\ \Phi_v[V_2]) \\
\Phi_v[\langle \mathbf{ph}\ p\ \circ\rangle] & = & \langle \mathbf{ph}\ p\ \circ\rangle \\
\Phi_v[\langle \mathbf{ph}\ p\ V\rangle] & = & \langle \mathbf{ph}\ p\ \Phi_v[V]\rangle \\
\\
\Phi_k : Cont_{pcek} & \longrightarrow & EvalCtxt \\
\Phi_k[\![\epsilon]\!] & = & [\,] \\
\Phi_k[\![\langle \mathbf{ar}\ x, M, E\rangle.K]\!] & = & \Phi_k[\![K]\!]\ [(\mathbf{let}\ (x\ [\,])\ \Phi_m[M, E])] \\
\Phi_k[\![\langle \mathbf{ar}\dagger\ x, M, E\rangle.K]\!] & = & \Phi_k[\![K]\!]\ [(\mathbf{let}\ (x\ (\mathbf{future}\,[\,]))\ \Phi_m[M, E])] \\
\\
\Phi_s : State_{pcek} & \longrightarrow & State_{pc'} \\
\Phi_s[\langle M, E, K\rangle] & = & \Phi_k[\![K]\!]\ [\Phi_m[M, E]] \\
\Phi_s[\mathtt{error}] & = & \mathtt{error} \\
\Phi_s[(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2)] & = & (\mathbf{f\text{-}let}\ (p\ \Phi_s[S_1])\ \Phi_s[S_2])
\end{array}
$$

FIGURE 9: The Functions Φ_m, Φ_v, Φ_k and Φ_s.

Lemma A.1 (Weak Bisimulation Theorem) *Let $S_1 \in State_{pcek}$ and $S_1' \in State_{pc'}$ such that $\Phi_s[S_1] = S_1'$.*

1. *If* $S'_1 \longmapsto^{n',m'}_{pc'} S'_2$ *then there exists* $S_2 \in State_{pcek}$ *such that* $S_1 \longmapsto^{n,m}_{pcek} S_2$, $\Phi_s[S_2] = S'_2$ *and* $m \geq m'$.
2. *If* $S_1 \longmapsto^{n,m}_{pcek} S_2$ *then* $S'_1 \longmapsto^{n',m'}_{pc'} \Phi_s[S_2]$.
3. *In part 2, for each* S_1 *there exists* $m_0 \in \mathbf{N}$ *such that* $m > m_0$ *implies* $m' > 0$.

Proof:

1. The proof of the first part is by case analysis of the last step in $S'_1 \longmapsto^{n',m'}_{pc'} S'_2$, based on a similar proof in [5] for sequential programs.
2. We prove the second part by case analysis of the last step in the transition $S_1 \longmapsto^{n,m}_{pcek} S_2$.
   - Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*bind-const*). Then

$$\begin{array}{rcll}
S_1 & = & \langle(\mathbf{let}\ (x\ c)\ M), E, K\rangle & \\
S_2 & = & \langle M, E[x \leftarrow c], K\rangle & \\
\Phi_s[S_1] & = & \mathcal{E}[\ \Phi_m[(\mathbf{let}\ (x\ c)\ M), E]\ ] & \text{where } \mathcal{E} = \Phi_k[\![K]\!] \\
 & = & \mathcal{E}[\ (\mathbf{let}\ (x\ c)\ \Phi_m[M, E])\ ] & \text{renaming so } x \notin dom(E) \\
 & \longmapsto^{1,1}_{pc'} & \mathcal{E}[\ \Phi_m[M, E][x \leftarrow c]\ ] & \\
 & = & \mathcal{E}[\ \Phi_m[M, E[x \leftarrow c]]\ ] & \text{by definition of } \Phi_m \\
 & = & \Phi_s[S_2] &
\end{array}$$

   - The analysis for the rules (*bind-var*), (*bind-lam*) and (*bind-cons*) is similar.
   - Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*future*). Then

$$\begin{array}{rcl}
S_1 & = & \langle(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M), E, K\rangle \\
S_2 & = & \langle N, E, \langle\mathbf{ar}\dagger\ x, M, E\rangle.K\rangle \\
\Phi_s[S_1] & = & \Phi_k[\![K]\!]\ [\ \Phi_m[(\mathbf{let}\ (x\ (\mathbf{future}\ N))\ M), E]\ ] \\
 & = & \Phi_k[\![K]\!]\ [\ (\mathbf{let}\ (x\ (\mathbf{future}\ \Phi_m[N, E,)])\ \Phi_m[M, E])\ ] \\
 & & \text{renaming so } x \notin dom(E) \\
 & = & \Phi_k[\![\langle\mathbf{ar}\dagger\ x, M, E\rangle.K]\!]\ [\ \Phi_m[N, E]\ ] \\
 & = & \Phi_s[S_2]
\end{array}$$

   - Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*return*). Then

$$\begin{array}{rcl}
S_1 & = & \langle x, E, \langle\mathbf{ar}\ y, M, E'\rangle.K\rangle \\
S_2 & = & \langle M, E'[y \leftarrow E(x)], K\rangle \\
\Phi_s[S_1] & = & \Phi_k[\![K]\!]\ [\ (\mathbf{let}\ (y\ E(x))\ \Phi_m[M, E'])\ ] \\
 & \longmapsto^{1,1}_{pc'} & \Phi_k[\![K]\!]\ [\ \Phi_m[M, E'][y \leftarrow E(x)]\ ] \\
 & = & \Phi_k[\![K]\!]\ [\ \Phi_m[M, E'[y \leftarrow E(x)]]\ ] \\
 & = & \Phi_s[S_2]
\end{array}$$

- The analysis for the rule (*future-id*) is similar.
- Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*car*). We analyze the error and non-error cases separately.
  - Suppose $S_2 \neq$ `error`. Then

$$\begin{array}{rcl} S_1 & = & \langle(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M), E, K\rangle \\ S_2 & = & \langle M, E[x \leftarrow V_1], K\rangle \\ & & \text{where } touch_{pcek}[E(y)] = (\mathbf{cons}\ V_1\ V_2) \\ \Phi_s[S_1] & = & \Phi_k[\![K]\!]\ [\ (\mathbf{let}\ (x\ (\mathbf{car}\ (\mathbf{cons}\ \Phi_v V_1\ \Phi_v V_2)))\ \Phi_m[M, E])\ ] \\ & \longmapsto^{1,1}_{pc'} & \Phi_k[\![K]\!]\ [\ \Phi_m[M, E][x \leftarrow \Phi_v V_1]\ ] \\ & = & \Phi_k[\![K]\!]\ [\ \Phi_m[M, E[x \leftarrow V_1]]\ ] \\ & = & \Phi_s[S_2] \end{array}$$

  - Suppose $S_2 =$ `error`. Then

$$\begin{array}{rcl} S_1 & = & \langle(\mathbf{let}\ (x\ (\mathbf{car}\ y))\ M), E, K\rangle \\ S_2 & = & \texttt{error} \\ & & \text{where } touch_{pcek}[E(y)] \notin \mathit{Pair} \cup \{\circ\} \\ \Phi_s[S_1] & = & \Phi_k[\![K]\!]\ [\ (\mathbf{let}\ (x\ (\mathbf{car}\ V))\ \Phi_m[M, E])\ ] \\ & & \text{where } V \neq (\mathbf{cons}\ V_1\ V_2), V \neq p \\ & \longmapsto^{1,1}_{pc'} & \texttt{error} \\ & = & \Phi_s[S_2] \end{array}$$

- The analysis for the rules (*cdr*), (*if*) and (*apply*) is similar.
- Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*fork*). Then

$$\begin{array}{rcl} S_1 & = & \langle M, E, K_1.\langle \mathbf{ar}\dagger\ x, N, E'\rangle.K_2\rangle \\ S_2 & = & (\mathbf{f\text{-}let}\ (p\ \langle M, E, K_1\rangle)\ \langle N, E'[x \leftarrow \langle \mathbf{ph}\ p\ \circ\rangle], K_2\rangle) \\ & & \text{where } p \notin FP(E') \cup FP(K_2) \\ \Phi_s[S_1] & = & \Phi_k[\![K_2]\!]\ [\ (\mathbf{let}\ (x\ (\mathbf{future}\ \Phi_k[\![K_1]\!][\ \Phi_m[M, E]\ ]))\ \Phi_m[N, E',)]] \\ & \longmapsto^{1,1}_{pc'} & (\mathbf{f\text{-}let}\ (p\ \Phi_k[\![K_1]\!][\ \Phi_m[M, E]\ ]) \\ & & \quad \Phi_k[\![K_2]\!]\ [\ \Phi_m[N, E'][x \leftarrow p\ ]]) \\ & = & \Phi_s[S_2] \end{array}$$

- Suppose $S_1 \longmapsto^{1,1}_{pcek} S_2$ via the rule (*join*). Then

$$\begin{array}{rcl} S_1 & = & (\mathbf{f\text{-}let}\ (p\ \langle x, E, \epsilon\rangle)\ S) \\ S_2 & = & S[p := E(x)] \\ \Phi_s[S_1] & = & (\mathbf{f\text{-}let}\ (p\ \Phi_v[E(x)])\ \Phi_s[S]) \\ & \longmapsto^{1,1}_{pc'} & \Phi_s[S][p := \Phi_v[E(x)]] \\ & = & \Phi_s[S[p := E(x)]] \\ & = & \Phi_s[S_2] \end{array}$$

- The analysis of for the rule (*join-error*) is similar.
- The cases for the rules (*reflexive*), (*transitive*) and (*parallel*) are straightforward.

3. Suppose that $S_1 \longmapsto^{n,m}_{pcek} S_2$ with $m > 0$, and $\Phi_s[S_1] \longmapsto^{n',m'}_{pc'} \Phi_s[S_2]$ with $m' = 0$. From the proof of part 2, we see that the only possibility for this case is via the rule (*future*), or via the (*transitive*) or (*parallel*) closure of the (*future*) rule. The number of consecutive such transitions from S_1 is bounded by $\theta[S_1]$, where θ is the following function:

$$\begin{array}{rcl}
\theta : \mathit{State} & \longrightarrow & \mathbf{N} \\
\theta[\langle M, E, K\rangle] & = & \begin{cases} 1 + \theta[\langle N_1, E, K\rangle] & \\ \quad\quad \text{if } M = (\mathbf{let}\ (x\ (\mathbf{future}\ N_1))\ N_2) & \\ 0 \quad\quad \text{otherwise} & \end{cases} \\
\theta[\mathtt{error}] & = & 0 \\
\theta[(\mathbf{f\text{-}let}\ (p\ S_1)\ S_2)] & = & \theta[S_1]
\end{array}$$

If we pick $m_0 = \theta[S_1]$ and $m > m_0$, it is impossible that $\Phi_s[S_1] \longmapsto^{n',m'}_{pc'} \Phi_s[S_2]$ with $m' = 0$. Hence $m' > 0$.

∎

The Correspondence Lemma implies the equivalence of the $P(C_{ph})$-machine and the $P(CEK)$-machines.

Lemma A.2 $\mathit{unload}_{pcek} = \mathit{unload}_{pc'} \circ \Phi_v$.

Theorem 5.1 (Correctness of eval_{pcek}) $\mathit{eval}_{pcek} = \mathit{eval}_{pc'}$.

Proof: The proof of the left-to-right direction proceeds by case analysis on the definitions of eval_{pcek}.

- Suppose $\mathit{eval}_{pcek}(P) = \mathit{unload}_{pcek}[E(x)]$ because $\langle P, \emptyset, \epsilon\rangle \longmapsto^{*}_{pcek} \langle x, E, \epsilon\rangle$. By the Weak Bisimulation Theorem (A.1), $P \longmapsto^{*}_{pc'} V$, where $V \equiv \Phi_v[E(x)]$. Hence

$$\mathit{eval}_{pc'}(P) = \mathit{unload}_{pc'}[\Phi_v[E(x)]] = \mathit{unload}_{pcek}[E(x)] = \mathit{eval}_{pcek}(P).$$

- The case where $\mathit{eval}_{pcek}(P) = \mathtt{error}$ is similar.
- Suppose $\mathit{eval}_{pcek}(P) = \bot$ because there exists $S_i \in \mathit{State}_{pcek}$ and $n_i, m_i \in \mathbf{N}$ such that $m_i > 0$, $S_0 = \langle P, \emptyset, \epsilon\rangle$ and $S_i \longmapsto^{n_i,m_i}_{pcek} S_{i+1}$. By the Weak Bisimulation Theorem (A.1), there exists $S'_i \in \mathit{State}_{pc'}$ and $n'_i, m'_i \in \mathbf{N}$ such that $m'_i > 0$, $P = S_0$ and $S'_i \longmapsto^{n'_i,m'_i}_{pc'} S'_{i+1}$. Hence $\mathit{eval}_{pc'}(P) = \bot$.

We prove right-to-left direction by case analysis of the definition of $\mathit{eval}_{pc'}$.

- Suppose $eval_{pc'}(P) = unload_{pc'}[V]$ via the reduction $P \longmapsto^*_{pc'} V$. By the Weak Bisimulation Theorem (A.1), $\langle P, \emptyset, \epsilon\rangle \longmapsto^*_{pcek} \langle x, E, \epsilon\rangle$, where $\Phi_v[E(x)] = V$. Hence

$$\begin{aligned} eval_{pcek}(P) &= unload_{pcek}[E(x)] = unload_{pc'}[\Phi_v[E(x)]] = unload_{pc'}[V] \\ &= eval_{pc'}(P). \end{aligned}$$

- The case where $eval_{pc'}(P) = \texttt{error}$ is similar.
- Suppose $eval_{pc'}(P) = \perp$ because there exists $S_i \in State_{pc'}$ and $n_i, m_i \in \mathbf{N}$ such that $m_i > 0$, $S_0 = P$ and $S_i \longmapsto^{n_i,m_i}_{pc'} S_{i+1}$. By the Weak Bisimulation Theorem (A.1), there exists $S'_i \in State_{pcek}$ and $n'_i, m'_i \in \mathbf{N}$ such that $m'_i > 0$, $S_0 = \langle P, \emptyset, \epsilon\rangle$ and $S'_i \longmapsto^{n'_i,m'_i}_{pcek} S'_{i+1}$. Hence $eval_{pcek}(P) = \perp$.

Hence, for all programs P, $eval_c(P) = eval_{pc}(P)$. ∎

References

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

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

[3] 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.

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

[5] 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), pp. 193–219.

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

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

[8] Flanagan, C., and Felleisen, M. The semantics of future and its use in program optimizations. In *POPL* (Jan. 1995).

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

[10] Gabriel, R., and McCarthy, J. Qlisp. *Parallel Computation and Computers for Artificial Intelligence* (1988), 63–89.

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

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

[13] Ito, T., and Matsui, M. A parallel lisp language: Pailisp and its kernel specification. [12:58–100].

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

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

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

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

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

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

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

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

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

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

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