J. Functional Programming **9** (1): 1–31, January 1999. Printed in the United Kingdom

The semantics of future and an application

C. FLANAGAN* and M. FELLEISEN
Rice University, 6100 Main Street, Houston, TX 77005-1892, USA

Abstract

The **future** annotation of MultiLisp provides a simple method for taming the implicit parallelism of functional programs. Prior research on **future** has concentrated on implementation and design issues, and has largely ignored the development of a semantic characterization of **future**. This paper considers an idealized functional language with **future**s and presents a series of operational semantics with increasing 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 by introducing *placeholder objects* and *touch* operations.

We use the last semantics to derive a program analysis algorithm and an optimization algorithm that removes provably redundant touch operations. Experiments with the Gambit compiler indicate that this optimization significantly reduces the overhead imposed by touch operations.

Capsule Review

Flanagan and Felleisen present four operational semantics for evaluating programs containing Halstead's **future** annotations indicating which expressions may be run in parallel. The simplest, definitional demantics views such annotations as having no semantic effect. The most complicated semantics shows how spawned threads coordinate, using environments to model substitution and continuations to model control flow. This last semantics might be the starting point of an implementation or for use in designing program analyses. I expected it to be difficult to describe the operational effects of **future** annotations. By starting with a simple operational semantics, and progressively adding the details that one would need to be concerned with in an implementation, the authors have formulated a clean and credible description of how **future** annotations should work.

1 Futures for parallel computation

Programs in functional languages offer numerous opportunities for executing program fragments 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. If such a strategy were applied indiscriminately, however, 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.

* Currently at Compaq System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.

The **future** annotations of MultiLisp and its Scheme successors (Baker & Hewitt, 1977; Halstead, 1985) provide a simple method for taming the implicit parallelism of functional programs. A programmer who believes that the parallel evaluation of some expression outweighs the overhead of creating a separate task may annotate the expression with the keyword **future**. An annotated functional program has the same observable behavior as the original program, but the run-time system may choose to evaluate the **future** expression in parallel with the rest of the program. If it does, the evaluation will proceed as if the annotated expression had immediately returned. Instead of a proper value though, it returns a placeholder, i.e. a value that contains enough information for retrieving the actual result of the annotated expression when needed. When a program operation requires specific knowledge about the value of some subcomputation but finds a placeholder, the run-time system performs a *touch* operation, which synchronizes the appropriate parallel threads, and eventually retrieves the necessary information.

Past research on **future**s has almost exclusively concentrated on the efficient implementation of the underlying task creation mechanism (Feeley, 1993; Ito & Matsui, 1989; Kranz *et al.*, 1989; Miller, 1987; Mohr *et al.*, 1990) and on the extension of the concept to languages with first-class continuations (Katz & Weise, 1990; Moreau, 1994b). In contrast, our goal is to develop a semantic framework and semantically well-founded optimizations for languages with **future**. The specific example we choose to consider is the development of an algorithm for removing provably redundant *touch* operations from programs. Our primary results are a series of semantics for a functional language with **future**s and a program analysis. The first semantics defines **future** to be a semantically-transparent annotation. The second one validates that a **future** expression interpreted as process creation is correct. The last semantics is a low-level refinement, which explicates the coordination of parallel tasks and the need for placeholder objects. This semantics is intensional enough to permit the derivation of a set-based program analysis (Heintze, 1994). The secondary result is a *touch* optimization algorithm (based on the analysis) with its correctness proof. The algorithm was added to the Gambit Scheme compiler (Feeley, 1993) and produced significant speedups on a standard set of benchmarks.

The presentation of our results proceeds as follows. The second section introduces an idealized functional language with **future**s, together with its definitional, sequential semantics that interprets **future**s as no-ops. The third section presents an equivalent parallel semantics for **future**s and the fourth section refines that semantics to introduce placeholders and *touch* operations. The fifth section discusses the cost of *touch* operations and presents a provably correct algorithm for eliminating unnecessary *touch* operations. The latter is based on the set-based analysis algorithm of the sixth section. The seventh section presents experimental results demonstrating the effectiveness of this optimization. Section eight discusses related work. For more details, we refer the interested reader to two technical reports on this work (Flanagan & Felleisen, 1994a; Flanagan & Felleisen, 1994b), which are expansions of a preliminary conference report (Flanagan & Felleisen, 1995).

Notation We use $f : A \longrightarrow B$ to denote that f is a total function from A to B, and use $\mathscr{P}_{\text{fin}}$ to denote the finite power-set constructor.

2 A functional language with future

2.1 Syntax

We formulate the semantics of **future** for an idealized, λ-calculus-like language with **future**s, conditionals 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, a set of basic constants that includes numbers and the empty list `nil`, and the term `error`, which is an error or runtime exception.

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

Fig. 1. The source language Λ.

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. For any set of terms X, we use X^0 to denote the set of closed terms in X.

2.2 Definitional semantics of future as identity

The future construct is an annotation that describes where parallelism may be usefully exploited by an implementation. This annotation is semantically transparent in that it does not change the meaning of programs, only how fast they are evaluated. We capture this definitional semantics of future as the identity operation via the abstract machine described in figure 2.

States of this abstract machine are simply closed terms, and the transition rules of the machine include the typical leftmost-outermost reductions of the lambda calculus (Plotkin, 1975; Felleisen & Friedman, 1986). For example, if the argument to the operation **car** is a pair, then the (*car*) rule extracts the first element of the pair.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Evaluator

$$eval_s : \Lambda^0 \longrightarrow Answers \cup \{\texttt{error}, \bot\}$$

$$eval_s(P) = \begin{cases} unload_s[V] & \text{if } P \longmapsto_s^* V \\ \texttt{error} & \text{if } P \longmapsto_s^* \texttt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists M_i \in State_s \text{ such that } P = M_0 \text{ and } M_i \longmapsto_s M_{i+1} \end{cases}$$

Data Specifications

S	$\in$	$State_s$	$::=$	Λ	(States)
F	$\in$	$FinalState_s$	$::=$	$V \mid \texttt{error}$	(Final States)
A	$\in$	$Answers$	$::=$	$c \mid \texttt{procedure} \mid (\textbf{cons}\ A\ A)$	(Answers)
$\mathscr{E}$	$\in$	$EvalCtxt$	$::=$	$[\,] \mid (\textbf{apply}\ \mathscr{E}\ M) \mid (\textbf{apply}\ V\ \mathscr{E}) \mid (\textbf{if}\ \mathscr{E}\ M\ M) \mid (\textbf{car}\ \mathscr{E}) \mid (\textbf{cdr}\ \mathscr{E})$	(Evaluation Contexts)

Unload Function

$$\begin{aligned} unload_s : Value^0 &\longrightarrow Answers \\ unload_s[c] &= c \\ unload_s[(\lambda x.\, M)] &= \texttt{procedure} \\ unload_s[(\textbf{cons}\ V_1\ V_2)] &= (\textbf{cons}\ unload_s[V_1]\ unload_s[V_2]) \end{aligned}$$

Transition Function

$$\mathscr{E}[\,(\textbf{apply}\ V_1\ V_2)\,] \longmapsto_s \begin{cases} \mathscr{E}[\,M[x \leftarrow V_2]\,] & \text{if } V_1 = (\lambda x.\, M) \\ \texttt{error} & \text{if } V_1 \neq (\lambda x.\, M) \end{cases} \quad (apply)$$

$$\mathscr{E}[\,(\textbf{car}\ V)\,] \longmapsto_s \begin{cases} \mathscr{E}[\,V_1\,] & \text{if } V = (\textbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } V \neq (\textbf{cons}\ V_1\ V_2) \end{cases} \quad (car)$$

$$\mathscr{E}[\,(\textbf{if}\ V\ M_1\ M_2)\,] \longmapsto_s \begin{cases} \mathscr{E}[\,M_1\,] & \text{if } V = \texttt{nil} \\ \mathscr{E}[\,M_2\,] & \text{if } V \neq \texttt{nil} \end{cases} \quad (if)$$

$$\mathscr{E}[\,\texttt{error}\,] \longmapsto_s \texttt{error} \quad \text{if } \mathscr{E} \neq [\,] \quad (error)$$

$$\mathscr{E}[\,(\textbf{future}\ M)\,] \longmapsto_s \mathscr{E}[\,(\textbf{future}\ M')\,] \quad \text{if } M \longmapsto_s M' \quad (future)$$

$$\mathscr{E}[\,(\textbf{future}\ V)\,] \longmapsto_s \mathscr{E}[\,V\,] \quad (future\text{-}id)$$

$$\mathscr{E}[\,(\textbf{future}\ \texttt{error})\,] \longmapsto_s \texttt{error} \quad (future\text{-}error)$$

Fig. 2. The sequential machine.

The transition rules also specify the error semantics of each class of expressions. Thus, if the argument to **car** is not a pair, then the (*car*) rule produces the state `error`. We omit the definition of transition rules for the operation **cdr** throughout this paper, since these rules are always analogous to those for **car**.

The only novel transition rules are the ones for **future** expressions, which interpret **future** as the identity operation. The (*future*) rule permits evaluation inside of a **future** expression. It is the only structural (inference) rule in the evaluation system.[1] Once the body of a **future** expression is first reduced to a value via the (*future*) rule, the (*future-id*) rule replaces the **future** expression with this value. The (*future-error*) rule handles the case where the evaluation of a future expression yields an error. This error is simply propagated to the enclosing context.

[1] We formulate the sequential semantics of **future** in this manner to simplify the correspondence with the parallel semantics presented in the next section.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

The definition of the transition rules relies on the notion of *evaluation contexts*. Roughly speaking, an evaluation context $\mathscr{E}$ is a term with a hole [ ] in place of the next sub-term to be evaluated; e.g. in (**car** M) the next sub-term to be evaluated is M, and thus the definition of evaluation contexts includes (**car** $\mathscr{E}$).

A sequential machine state is a *final state* if it is either a value or the special term `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 $M \in State_s$.

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

The sequential machine defines the semantics of the language via an evaluator function ($eval_s$) from closed programs to results. A result is either an answer, which is a closed value with all λ-expression replaced by the tag `procedure`,[2] or `error`, indicating that some program operation was misapplied, or $\perp$, if the program diverges.[3] The Uniform Evaluation Theorem implies that the evaluator $eval_s$ is a well-defined total function. Either the transition sequence for a program P terminates in a final state, in which case $eval_s(P)$ is an answer or `error`, or else the transition sequence is infinite, in which case $eval_s(P) = \perp$.

Theorem 2.2
The evaluator $eval_s$ is a total function.

Since the transition rules (*future*), (*future-id*), and (*future-error*) interpret **future** as the identity operation, the sequential machine correctly captures the extensional semantics of **future** as the identity operation. Any parallel implementation strategy for **future** should respect this extensional semantics.

3 Parallel operational semantics

The sequential abstract machine just described interprets **future** as an annotation, but ignores its *intension* as an advisory instruction concerning parallel evaluations. To understand this intensional aspect of **future** annotations, we need a semantics of **future** that models the parallel evaluation of **future** expressions. For this purpose, we extend the sequential machine into a parallel machine.

3.1 Specification of the parallel machine

State space The state space of the parallel abstract machine is defined in figure 3. In addition to sequential machine states, which represent single threads of control, the

[2] The evaluator removes λ-expressions from answers so that the observable behavior of programs does not depend on the terms themselves, but only on their meaning.
[3] We make divergent evaluations explicit because in the parallel semantics of the next section we need to ensure that a divergent speculative thread does not cause divergence of the overall evaluation.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Evaluator

$$eval_p : \Lambda^0 \longrightarrow Answers \cup \{\texttt{error}, \bot\}$$

$$eval_p(P) = \begin{cases} unload_p[V] & \text{if } P \longmapsto_p^* V \\ \texttt{error} & \text{if } P \longmapsto_p^* \texttt{error} \\ \bot & \text{if } \forall i \in \mathbf{N}\ \exists S_i \in State_p, n_i, m_i \in \mathbf{N} \text{ such that} \\ & m_i > 0, P = S_0 \text{ and } S_i \longmapsto_p^{n_i,m_i} S_{i+1} \end{cases}$$

Data Specifications

S	$\in$	$State_p$	$::=$	$M \mid (\textbf{f-let}\ (p\ S)\ S)$	(States)
M	$\in$	Λ_p	$::=$	$V \mid (\textbf{apply}\ M\ M) \mid \ldots$	(As for Λ)
V	$\in$	$Value_p$	$::=$	$PValue_p \mid p$	(Run-time Values)
		$PValue_p$	$::=$	$c \mid x \mid (\lambda x.\, M) \mid (\textbf{cons}\ V\ V)$	(Proper Values)
p	$\in$	$Ph\text{-}Vars$	$::=$	$\{p_1, p_2, p_3, \ldots\}$	(Placeholder Variables)
				$Ph\text{-}Vars \cap Vars = \emptyset$	
F	$\in$	$FinalState_p$	$::=$	$V \mid \texttt{error}$	(Final States)

Unload Function : $unload_p = unload_s$

Transition Rules

$$\mathscr{E}[\ (\textbf{apply}\ V_1\ V_2)\ ] \longmapsto_p^{1,1} \begin{cases} \mathscr{E}[\ M[x \leftarrow V_2]\ ] & \text{if } V_1 = (\lambda x.\, M) \\ \texttt{error} & \text{if } V_1 \neq (\lambda x.\, M), V \neq p \end{cases} \quad (apply)$$

$$\mathscr{E}[\ (\textbf{car}\ V)\ ] \longmapsto_p^{1,1} \begin{cases} \mathscr{E}[\ V_1\ ] & \text{if } V = (\textbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } V \neq (\textbf{cons}\ V_1\ V_2), V \neq p \end{cases} \quad (car)$$

$$\mathscr{E}[\ (\textbf{if}\ V\ M_1\ M_2)\ ] \longmapsto_p^{1,1} \begin{cases} \mathscr{E}[\ M_1\ ] & \text{if } V = \texttt{nil} \\ \mathscr{E}[\ M_2\ ] & \text{if } V \neq \texttt{nil}, V \neq p \end{cases} \quad (if)$$

$$\mathscr{E}[\ \texttt{error}\ ] \longmapsto_p^{1,1} \texttt{error} \quad \text{if } \mathscr{E} \neq [\ ] \quad (error)$$

$$\mathscr{E}[\ (\textbf{future}\ M)\ ] \longmapsto_p^{1,1} \mathscr{E}[\ (\textbf{future}\ M')\ ] \quad \text{if } M \longmapsto_p^{1,1} M' \quad (future)$$

$$\mathscr{E}[\ (\textbf{future}\ V)\ ] \longmapsto_p^{1,1} \mathscr{E}[\ V\ ] \quad (future\text{-}id)$$

$$\mathscr{E}[\ (\textbf{future}\ \texttt{error})\ ] \longmapsto_p^{1,1} \texttt{error} \quad (future\text{-}error)$$

$$\mathscr{E}[\ (\textbf{future}\ N)\ ] \longmapsto_p^{1,0} (\textbf{f-let}\ (p\ N)\ \mathscr{E}[\ p\ ]) \quad p \notin FP(\mathscr{E}) \quad (fork)$$

$$(\textbf{f-let}\ (p\ V)\ S) \longmapsto_p^{1,1} S[p \leftarrow V] \quad (join)$$

$$(\textbf{f-let}\ (p\ \texttt{error})\ S) \longmapsto_p^{1,1} \texttt{error} \quad (join\text{-}error)$$

$$S \longmapsto_p^{0,0} S \quad (reflexive)$$

$$S \longmapsto_p^{n,m} S'' \quad (transitive)$$
$$\text{if } S \longmapsto_p^{a,b} S',\ S' \longmapsto_p^{c,d} S'' \text{ and } a > 0,\ c > 0,\ n = a + c,\ m = b + d$$

$$(\textbf{f-let}\ (p\ S_1)\ S_2) \longmapsto_p^{n,m} (\textbf{f-let}\ (p\ S_1')\ S_2') \quad (parallel)$$
$$\text{if } S_1 \longmapsto_p^{a,m} S_1',\ S_2 \longmapsto_p^{c,d} S_2',\ n = a + c$$

Fig. 3. The parallel machine.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

parallel machine also includes states of the form (**f-let** (p S_1) S_2), which model the concurrent evaluation of **future** expressions by parallel threads. All instances of **f-let** in a state occur at the top-level, in an **f-let** tree. The **f-let** tree is a convenient method for representing the current collection of parallel threads and their interdependencies.

The *primary* sub-state S_1 of a state (**f-let** (p S_1) S_2) is initially the body of the **future** expression, and the *secondary* sub-state S_2 is initially the evaluation context surrounding the **future** expression. The placeholder variable p represents the result of S_1 in S_2. 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.

The set of values of the parallel machine includes the values of the sequential machine (constants, variables, closures and pairs), which we refer to as *proper* values. To model the implementation of futures, the parallel machine also includes a new class of values called *placeholder variables*. A placeholder variable p represents the result of a computation in progress. Once the computation terminates, all occurrences of the placeholder are replaced by the value returned by the computation. The usual conventions for binding constructs like λ and **let** apply to **f-let**. We use $S[p \leftarrow V]$ to denote the capture-free substitution of V for all free occurrences of p within a state S. The function $FP(S)$ determines the free placeholders variables in a state S, and a state is *closed* if it contains no free variables or free placeholders.

Transition rules The transition relation of the parallel machine is a predicate on quadruples $\cdot \longmapsto_p^{\cdot} \cdot$. The transition relation is **f-let** compatible and transitive-reflexive. To provide for induction proofs, it includes the definiton of two indices that measure the number of steps taken and how many of them are mandatory (i.e. , not speculative). Specifically, if $S \longmapsto_p^{n,m} S'$ holds, then the index n is the number of steps involved in the transition from S to S', and the index $m \leqslant n$ is the number of these steps that are mandatory.

The transition relation is formulated as a collection of transition rules. The rules (*apply*), (*car*), (*if*), (*error*), (*future-id*) and (*future-error*) are simply the corresponding transition rules of the sequential machine, appropriately modified where necessary to allow for undetermined placeholders. An application of one of these rules counts as a mandatory step. The (*future*) rule permits sequential evaluation inside a **future** expression. We forbid parallel evaluation inside a **future** expression in order to simplify the state space of the parallel machine.

The (*fork*) rule initiates parallel evaluation by spawning the outermost **future** expression. It may be applied whenever the current term is of the form:

$$\mathscr{E}[\ (\textbf{future}\ N)\ ]$$

The **future** annotation allows the expression N to be evaluated in parallel with the enclosing context $\mathscr{E}[\ \ ]$. The machine creates a new placeholder p to represent the

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

result of N, and initiates parallel evaluation of N and $\mathscr{E}[\ p\ ]$. We always spawn the outermost **future** expression in order to maximize the granularity of the created task,[4] and also to simplify the correctness proof of the parallel machine.

The 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 (*join*) rule 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 (*join-error*) rule discards the secondary sub-state S_2 and returns `error` as the result of the parallel state.

The rules (*reflexive*) and (*transitive*) close the relation under reflexivity and transitivity. The transition rule (*parallel*) permits concurrent evaluation of both sub-states of a parallel state (**f-let** $(p\ S_1)\ S_2$).

We write $S \longmapsto_p^* S'$ if $S \longmapsto_p^{n,m} S'$ for some $n, m \in \mathbf{N}$. A state S is in *normal form* if there is no state S' such that $S \longmapsto_p^{n,m} S'$ 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.

Nondeterminism Unlike the transition *function* of the sequential machine, which maps each state to a unique successor state, the transition *relation* of the parallel machine has an important degree of freedom. In particular, it does not specify when the (*fork*) rule applies. For example, consider the state $\mathscr{E}[$ (**future** N) $]$. An implementation of the machine may proceed either by evaluating the **future** body N via a (*future*) transition or by creating a new task via a (*fork*) transition. The choice of whether or not to apply the (*fork*) rule 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* (Kranz *et al.*, 1989; Swanson *et al.*, 1988; Gabriel & McCarthy, 1988). Alternatively, an implementation may *never* invoke the (*fork*) rule, resulting in a purely sequential evaluation. In between these two extremes lies 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* (Feeley, 1993; Mohr *et al.*, 1990), where new tasks are created via *fork* transitions only when the additional parallelism can exploit idle computing resources.

A second source of nondeterminism in the parallel 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 $eval_p$, is that the implementation must perform mandatory computation steps on a regular basis.

Evaluation The parallel machine can evaluate a program via many different transition sequences. Some of these transition sequences may be infinite, even if the

[4] This strategy is used in many implementations of **future** (Mohr *et al.*, 1990; Feeley, 1993) and of other parallel functional languages (Flanagan & Nikhil, 1996).

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

program terminates according to the sequential semantics. Consider the program

$$P = (\textbf{apply}\ (\lambda x.\,\Omega)\ (\textbf{future}\ \texttt{error}))$$

where Ω is some diverging sequential term such that $\Omega \mapsto\!\!\longrightarrow_p^{n,m} \Omega \mapsto\!\!\longrightarrow_p^{n,m} \Omega \mapsto\!\!\longrightarrow_p^{n,m} \cdots$. The sequential evaluator never executes Ω because P's result is `error`. In contrast, P admits the following infinite parallel transition sequence:

$$\begin{array}{lll} P & \mapsto\!\!\longrightarrow_p^{1,0} & (\textbf{f-let}\ (p\ \texttt{error})\ (\textbf{apply}\ (\lambda x.\,\Omega)\ p)) \qquad \text{via } (\textit{fork}) \\ & \mapsto\!\!\longrightarrow_p^{1,0} & (\textbf{f-let}\ (p\ \texttt{error})\ \Omega) \\ & \mapsto\!\!\longrightarrow_p^{n,0} & (\textbf{f-let}\ (p\ \texttt{error})\ \Omega) \qquad \text{since } \Omega \mapsto\!\!\longrightarrow_p^{n,m} \Omega \\ & \mapsto\!\!\longrightarrow_p^{n,0} & \cdots \end{array}$$

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 parallel machine excludes these *excessively speculative* or *unfair* transition sequences, and only admits *fair* transition sequences that regularly include mandatory transition steps.[5] 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 parallel machine nondeterministically 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 $eval_p$ returns $unload_p[V]$ or `error`, respectively. If the chosen transition sequence does not terminate, then $eval_p$ returns $\perp$. As we will prove below, the evaluator relation $eval_p$ is a total function and is extensionally identical to the sequential evaluator $eval_s$.

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**, **cdr**, **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 parallel 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.

[5] The concept of a mandatory step is closely related to the notion of legitimacy introduced by Katz and Weise (1990).

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

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

$$
\begin{array}{lll}
P & = & (\textbf{car}\ (\textbf{future}\ (\textbf{apply}\ (\lambda x.(\textbf{cons}\ x\ x))\ 1))) \quad (1)\\
& \longmapsto_p^{1,0} & (\textbf{f-let}\ (p\ (\textbf{apply}\ (\lambda x.(\textbf{cons}\ x\ x))\ 1))\ (\textbf{car}\ p)) \quad (2)\\
& \longmapsto_p^{1,1} & (\textbf{f-let}\ (p\ (\textbf{cons}\ 1\ 1))\ (\textbf{car}\ p)) \quad (3)\\
& \longmapsto_p^{1,1} & (\textbf{car}\ (\textbf{cons}\ 1\ 1)) \quad (4)\\
& \longmapsto_p^{1,1} & 1 \quad (5)
\end{array}
$$

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 (**car** p). The (*car*) rule cannot fire since the argument of **car** is a placeholder. Evaluation of the primary substate proceeds unhindered. Once the primary substate produces a value (line 3), the (*join*) rule synchronizes the separate threads of computation by replacing all occurrences of p by that value. After synchronization (line 4), 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 parallel 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. Since this property is crucial in proving the correctness of the machine, we formulate a matching lemma.

Lemma 3.1 (*Substitutivity; Placeholder Transparency*)
If $S_1 \longmapsto_p^{n,m} S_2$, then for any placeholder p and any $W \in Value_p$,

$$S_1[p \leftarrow W] \longmapsto_p^{n,m} S_2[p \leftarrow W]\ .$$

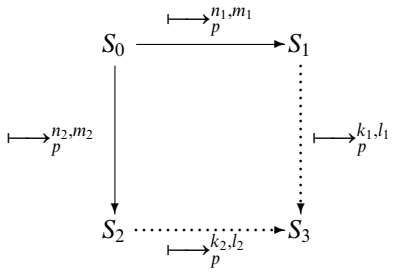
Proof
The proof is by lexicographic induction on n and on the size of S_1, and proceeds by case analysis on the transition rule used for $S_1 \longmapsto_p^{n,m} S_2$. □

3.2 Consistency of the parallel machine

The *observable* behavior of the parallel machine on a given program is deterministic, despite its nondeterminate *internal* behavior. We prove this fact in the traditional manner, using a extended form of the Diamond Lemma. The Extended Diamond Lemma states that if we reduce an initial state S_0 by two alternative transitions, producing respectively states S_1 and S_2, then there is some state S_3 that is reachable from both S_1 and S_2. Furthermore, the lemma also relates the number of steps involved in the various state transitions. This relationship among transition steps is crucial to proving that all transition sequences for a given program exhibit the same termination behavior.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Lemma 3.2 (*Extended Diamond Lemma*)
Let $S_0, S_1, S_2 \in State_p$. If $S_0 \longmapsto_p^{n_1,m_1} S_1$ and $S_0 \longmapsto_p^{n_2,m_2} S_2$, then there exists $S_3 \in State_p$ and $k_1, l_1, k_2, l_2 \in \mathbf{N}$ such that $S_1 \longmapsto_p^{k_1,l_1} S_3$ and $S_2 \longmapsto_p^{k_2,l_2} S_3$. Furthermore, the following inequalities hold:



$$\begin{aligned} m_1 + k_1 &\leqslant n_2 + l_2 \\ m_2 + k_2 &\leqslant n_1 + l_1 \\ k_1 &\leqslant n_2 \\ k_2 &\leqslant n_1 \end{aligned}$$

Proof
The proof proceeds by lexicographic induction $n_1 + n_2$ and on the size of S_0, and by case analysis on the pair of transition rules used for $S_0 \longmapsto_p^{n_1,m_1} S_1$ and $S_0 \longmapsto_p^{n_2,l_1} S_2$.

To simplify the case analysis, we define the transition relation (*seq*) as the union of (*apply*), (*car*), (*if*) and (*error*). The transition relation of the parallel machine is then the union of the ten relations (*seq*), (*future*), (*future-id*), (*future-error*), (*fork*), (*join*), (*join-error*), (*parallel*), (*reflexive*) and (*transitive*). 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	*fut*	*fut-id*	*fut-err*	*fork*	*join*	*j-err*	*par.*	*refl.*	*trans.*
(*seq*)	≡	×	×	×	×	×	×	×	1	2
(*future*)		≡	×	×	3	×	×	×	1	2
(*future-id*)			≡	×	4	×	×	×	1	2
(*future-error*)				≡	5	×	×	×	1	2
(*fork*)					≡	×	×	×	1	2
(*join*)						≡	×	6	1	2
(*join-error*)							≡	7	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.

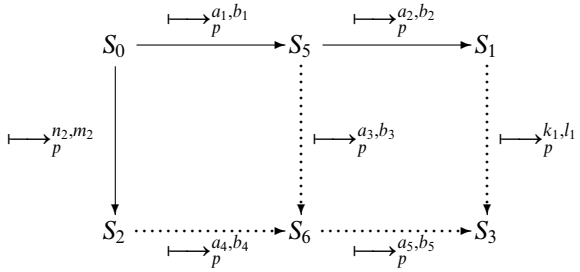
(≡) The cases marked by the symbol ≡ hold, since the relation in question is a partial function, and hence $S_1 = S_2$.

(1) For the case where $S_0 \longmapsto_p^{0,0} S_1$ via the rule *reflexive*, take $S_3 = S_2$.

(2) If $S_0 \longmapsto_p^{n_1,m_1} S_1$ via the rule *transitive*, we have that $S_0 \longmapsto_p^{a_1,b_1} S_5 \longmapsto_p^{a_2,b_2} S_1$, where $n_1 = a_1 + a_2$, $m_1 = b_1 + b_2$, $a_1 > 0$ and $b_1 > 0$. The following diagram

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

outlines our proof technique for this case:



Since $a_1 + n_2 < n_1 + n_2$, by the inductive hypothesis there exists $S_6 \in State_p$ and $a_3, b_3, a_4, b_4 \in \mathbf{N}$ such that:

$$\begin{array}{llllll} S_5 \longmapsto_p^{a_3,b_3} S_6 & b_1 + a_3 \leqslant n_2 + b_4 & a_4 \leqslant a_1 \\ S_2 \longmapsto_p^{a_4,b_4} S_6 & m_2 + a_4 \leqslant a_1 + b_3 & a_3 \leqslant n_2 \end{array}$$

Similarly, since $S_5 \longmapsto_p^{a_2,b_2} S_1$, $S_5 \longmapsto_p^{a_3,b_3} S_6$ and $a_2 + a_3 < n_1 + n_2$, by the inductive hypothesis there exists $S_3 \in State_p$ and $k_1, l_1, a_5, b_5 \in \mathbf{N}$ such that:

$$\begin{array}{llllll} S_1 \longmapsto_p^{k_1,l_1} S_3 & b_2 + k_1 \leqslant a_3 + b_5 & a_5 \leqslant a_2 \\ S_6 \longmapsto_p^{a_5,b_5} S_3 & b_3 + a_5 \leqslant a_2 + l_1 & k_1 \leqslant a_3 \end{array}$$

Let $k_2 = a_4 + a_5$ and $l_2 = b_4 + b_5$. Then $S_1 \longmapsto_p^{k_1,l_1} S_3$ and $S_2 \longmapsto_p^{k_2,l_2} S_3$. Furthermore:

$$\begin{array}{rcll} m_1 + k_1 &=& b_1 + b_2 + k_1 & \\ &=& (b_1 + a_3) + (b_2 + k_1) - a_3 & \\ &\leqslant& (n_2 + b_4) + (a_3 + b_5) - a_3 & (I.H.) \\ &=& n_2 + l_2 & \\ m_2 + k_2 &=& m_2 + a_4 + a_5 & \\ &=& (m_2 + a_4) + (b_3 + a_5) - b_3 & \\ &\leqslant& (a_1 + b_3) + (a_2 + l_1) - b_3 & (I.H.) \\ &=& n_1 + l_1 & \\ k_1 &\leqslant& a_2 \leqslant n_2 & \\ k_2 &=& a_4 + a_5 \leqslant a_1 + a_2 = n_1 & \end{array}$$

Thus the lemma holds for this case.

(3) Suppose $S_0 \longmapsto_p^{1,1} S_1$ via the rule (*future*) and $S_0 \longmapsto_p^{1,0} S_2$ via the rule (*fork*). Then:

$$\begin{array}{rcll} S_0 &=& \mathscr{E}[\ (\textbf{future}\ N)\ ] & \\ S_1 &=& \mathscr{E}[\ (\textbf{future}\ N')\ ] & \text{where } N \longmapsto_p^{1,1} N' \\ S_2 &=& (\textbf{f-let}\ (p\ N)\ \mathscr{E}[\ p\ ]) & \end{array}$$

Let $S_3 = (\textbf{f-let}\ (p\ N')\ \mathscr{E}[\ p\ ])$. Clearly $S_1 \longmapsto_p^{1,0} S_3$ via (*fork*), and $S_2 \longmapsto_p^{1,1} S_3$ via (*parallel*).

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

(4) Suppose $S_0 \longmapsto_p^{1,1} S_1$ via the rule (*future-id*) and $S_0 \longmapsto_p^{1,0} S_2$ via the rule (*fork*). Then:

$$\begin{aligned} S_0 &= \mathscr{E}[\ (\textbf{future}\ V)\ ] \\ S_1 &= \mathscr{E}[\ V\ ] \\ S_2 &= (\textbf{f-let}\ (p\ V)\ \mathscr{E}[\ p\ ]) \end{aligned}$$

Let $S_3 = S_1$. Then $S_1 \longmapsto_p^{0,0} S_3$ via (*reflexive*) and $S_2 \longmapsto_p^{1,1} S_3$ via (*join*).

(5) The case where $S_0 \longmapsto_p^{1,1} S_1$ via (*future-error*) and $S_0 \longmapsto_p^{1,0} S_2$ via (*fork*) holds by reasoning similar to the previous case.

(6) Suppose $S_0 \longmapsto_p^{1,1} S_1$ via the rule (*join*) and $S_0 \longmapsto_p^{n,0} S_2$ by the rule (*parallel*). Then:

$$\begin{aligned} S_0 &= (\textbf{f-let}\ (p\ V)\ S) \\ S_1 &= S[p \leftarrow V] \\ S_2 &= (\textbf{f-let}\ (p\ V)\ S') \qquad \text{where } S \longmapsto_p^{n,m} S' \end{aligned}$$

Pick $S_3 = S'[p \leftarrow V]$. Then $S_1 \longmapsto_p^{n,m} S_3$ by the Placeholder Transparency Lemma (3.1), and $S_2 \longmapsto_p^{1,1} S_3$ via the rule (*join*).

(7) The case where $S_0 \longmapsto_p^{1,1} S_1$ via (*join-error*) and $S_0 \longmapsto_p^{n,0} S_2$ via (*parallel*) holds by reasoning similar to the previous case.

(8) Suppose both the transitions $S_0 \longmapsto_p^{n_1,m_1} S_1$ and $S_0 \longmapsto_p^{n_2,m_2} S_2$ are via the rule (*parallel*). Then:

$$\begin{aligned} S_0 &= (\textbf{f-let}\ (p\ S_0')\ S_0'') \\ S_1 &= (\textbf{f-let}\ (p\ S_1')\ S_1'') \\ &\quad \text{where } S_0' \longmapsto_p^{a_1,m_1} S_1',\ S_0'' \longmapsto_p^{c_1,d_1} S_1'' \text{ and } n_1 = a_1 + c_1 \\ S_2 &= (\textbf{f-let}\ (p\ S_2')\ S_2'') \\ &\quad \text{where } S_0' \longmapsto_p^{a_2,m_2} S_2',\ S_0'' \longmapsto_p^{c_2,d_2} S_2'' \text{ and } n_2 = a_2 + c_2 \end{aligned}$$

Since $a_1 + a_2 \leqslant n_1 + n_2$ and S_0' is strictly smaller than S_0, by the inductive hypothesis there exists S_3' such that:

$$\begin{array}{llllll} S_1' & \longmapsto_p^{a_3,l_1} & S_3' & m_1 + a_3 \leqslant a_2 + l_2 & a_3 \leqslant a_2 \\ S_2' & \longmapsto_p^{a_4,l_2} & S_3' & m_2 + a_4 \leqslant a_1 + l_1 & a_4 \leqslant a_1 \end{array}$$

Similarly, there exists S_3'' such that:

$$\begin{array}{llllll} S_1'' & \longmapsto_p^{c_3,d_3} & S_3'' & d_1 + c_3 \leqslant c_2 + d_4 & c_3 \leqslant c_2 \\ S_2'' & \longmapsto_p^{c_4,d_4} & S_3'' & d_2 + c_4 \leqslant c_1 + d_3 & c_4 \leqslant c_1 \end{array}$$

Letting $S_3 = (\textbf{f-let}\ (p\ S_3')\ S_3'')$, we have:

$$\begin{aligned} S_1 &\longmapsto_p^{k_1,l_1} S_3 \\ S_2 &\longmapsto_p^{k_2,l_2} S_3 \end{aligned}$$

where $k_1 = a_3 + c_3$, $k_2 = a_4 + c_4$, and the indices satisfy the inequalities.

□

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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Lemma 3.3 (*Consistency of Transitions*)
Let P be a program. If $P \longmapsto_p^* S$, where S is in normal form, then:

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

Proof

1. Suppose $P \longmapsto_p^* S$ and $P \longmapsto_p^* S'$, where S, S' are in normal form. By the Extended Diamond Lemma (3.2), there exists some S_4 such that $S \longmapsto_p^* S_4$ and $S' \longmapsto_p^* S_4$. However, since S, S' are in normal form, we have that $S = S_4 = S'$.
2. We prove part 2 by contradiction. Assume that $P \longmapsto_p^{n,m} S$ where S is in normal form, and that there exists some sequence of states $S_i \in State_p$ and $n_i, m_i \in \mathbf{N}$ such that $m_i > 0$, $P = S_0$ and $S_i \longmapsto_p^{n_i,m_i} S_{i+1}$.
   Pick an integer $k > n$. Then $P \longmapsto_p^{a,b} S_{k+1}$, where $a = \Sigma_{i=0}^{i=k} n_i$ and $b = \Sigma_{i=0}^{i=k} m_i$. By the Extended Diamond Lemma (3.2), $S_{k+1} \longmapsto_p^{c,d} S$ (because S is in normal form) for some $c, d \in \mathbf{N}$, with $n \geqslant b + c$. But $b + c \geqslant b = \Sigma_{i=0}^{i=k} m_i \geqslant k + 1 > k$, since $m_i > 0$, producing the contradiction $n > k > n$.

□

The Consistency of Transitions Lemma implies that all transition sequences for a given program exhibit the same observable behavior, and hence the evaluator $eval_p$ for the parallel machine is a well-defined function.

Theorem 3.4 (*Consistency of* $eval_p$)
The relation $eval_p$ is a function.

3.3 Correctness of the parallel machine

Every transition rule of the sequential machine has a corresponding rule in the parallel machine, which implies that every transition of the sequential machine is also a transition of the parallel machine.

Lemma 3.5
Suppose $S \longmapsto_s S'$, for $S, S' \in State_s$. Then $S \longmapsto_p^{1,1} S'$.

The correspondence between transitions of the two machines (together with Theorem 3.4) implies that their respective evaluators are equivalent.

Theorem 3.6 (*Correctness of* $eval_p$)
$eval_s = eval_p$.

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

- Suppose $eval_s(P) = unload_s[V]$ because $P \longmapsto_s^* V$. Then $P \longmapsto_p^* V$, and hence $eval_p(P) = unload_p[V] = unload_s[V]$.
- Similarly, if $eval_s(P) = \texttt{error}$ because $P \longmapsto_s^* \texttt{error}$, then $P \longmapsto_p^* \texttt{error}$, and hence $eval_p(P) = \texttt{error}$.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

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

$$P \equiv S_0 \longmapsto_s S_1 \longmapsto_s \cdots \longmapsto_s S_k \longmapsto_s \cdots .$$

Then $P \equiv S_0 \longmapsto_p^{1,1} S_1 \longmapsto_p^{1,1} \cdots \longmapsto_p^{1,1} S_k \longmapsto_p^{1,1} \cdots$, and thus $eval_p(P) = \perp$.

Hence $eval_s \subseteq eval_p$. Since $eval_s$ is total and $eval_p$ is a function (from Theorem 3.4), we also have that $eval_s = eval_p$. □

The equivalence of the two evaluators implies that $eval_p$ is defined for all programs.

In summary, the parallel machine correctly implements the sequential machine in that they 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 parallel machine specifies the parallel execution behavior of programs with **future** at the source level, and hides low-level operations that are required in realistic implementations of **future**. In particular, implementations typically represent placeholders using *placeholder objects* to avoid the expensive substitution operation on placeholders (compare (*join*)). Instead, placeholder objects are mutated in place. This technique requires *touch* operations at placeholder-strict positions in computational primitives to dereference placeholder objects when necessary. Since we plan to use the semantics of **future** to develop an algorithm for removing redundant *touch* operations, we reformulate the parallel machine to expose these placeholder objects and the associated *touch* operations, and we also associate an identifying label with each expression in a placeholder-strict position. The result is an abstract machine called the *placeholder machine*.

4.1 Specification of the placeholder machine

The state space of the placeholder machine is a minor revision of that of the parallel machine – see figure 4. Instead of plain variables, placeholders are now tagged objects. An *undetermined placeholder object* (written $\langle \mathbf{ph}\ p \rangle$) is created to represent the result of each parallel task (compare (*fork*)). The placeholder variable p is used to distinguish placeholder objects from each other. When the computation associated with the placeholder object terminates, producing a value V, the undetermined placeholder object is replaced by the *determined placeholder object* $\langle \mathbf{ph}\ V \rangle$.[6]

The correctness of this technique requires that a determined placeholder object $\langle \mathbf{ph}\ V \rangle$ representing the value V is observably equivalent to the value V itself. To ensure this behavior, the primitives **car**, **cdr**, **if** and **apply** perform *touch* operations on their placeholder-strict arguments. A *touch* operation behaves as the identity

[6] The explicit replacement of undetermined placeholders by determined placeholders is still unrealistic, but suffices for our purposes. Alternatively, the explicit substitution could be avoided via a lookup rule along the lines of Felleisen and Hieb (1989).

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Evaluator

$$eval_{ph} : \Lambda^0 \longrightarrow Answers \cup \{\texttt{error}, \bot\}$$

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

where $P \in \Lambda^0$ is converted to $P' \in \Lambda^0_{ph}$ by adding labels as appropriate

Data Specifications

S	$\in$	$State_{ph}$	$::=$	$M \mid (\textbf{f-let}\ (p\ S)\ S)$	(States)
V	$\in$	$Value_{ph}$	$::=$	$PValue_{ph} \mid PHValue_{ph}$	(Run-time Values)
		$PHValue_{ph}$	$::=$	$\langle \textbf{ph}\ p \rangle \mid \langle \textbf{ph}\ V \rangle$	(Placeholder Values)
		$PValue_{ph}$	$::=$	$c \mid x \mid Cl_{ph} \mid Pair_{ph}$	(Proper Values)
		Cl_{ph}	$::=$	$(\lambda x.\, M)$	(Closures)
		$Pair_{ph}$	$::=$	$(\textbf{cons}\ V\ V)$	(Pairs)
l	$\in$	$Label$			(Labels)
M	$\in$	Λ_{ph}	$::=$	$V \mid (\textbf{apply}\ M^l\ M)$	(Terms with labels)
				$\mid (\textbf{if}\ M^l\ M\ M)$	
				$\mid (\textbf{car}\ M^l) \mid (\textbf{cdr}\ M^l)$	
				$\mid \texttt{error} \mid (\textbf{future}\ M)$	

Unload Function

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

Touch Function

$$touch_{ph} : Value_{ph} \longrightarrow PValue_{ph} \cup \{\langle \textbf{ph}\ p \rangle\}$$
$$touch_{ph}[\langle \textbf{ph}\ V \rangle] = touch_{ph}[V]$$
$$touch_{ph}[V] = V \text{ otherwise}$$

Placeholder Substitution $S[p := V]$

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

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

Transition Rules

$$\mathscr{E}[\ (\textbf{apply}\ V_1^l\ V_2)\ ] \longmapsto_{ph}^{1,1} \begin{cases} \mathscr{E}[\ M[x \leftarrow V_2]\ ] & \text{if } touch_{ph}[V_1] = (\lambda x.\, M) \\ \texttt{error} & \text{if } touch_{ph}[V_1] \notin Cl_{ph} \cup \{\langle \textbf{ph}\ p \rangle\} \end{cases} \quad (apply)$$

$$\mathscr{E}[\ (\textbf{car}\ V^l)\ ] \longmapsto_{ph}^{1,1} \begin{cases} \mathscr{E}[\ V_1\ ] & \text{if } touch_{ph}[V] = (\textbf{cons}\ V_1\ V_2) \\ \texttt{error} & \text{if } touch_{ph}[V] \notin Pair_{ph} \cup \{\langle \textbf{ph}\ p \rangle\} \end{cases} \quad (car)$$

$$\mathscr{E}[\ (\textbf{if}\ V^l\ M_1\ M_2)\ ] \longmapsto_{ph}^{1,1} \begin{cases} \mathscr{E}[\ M_1\ ] & \text{if } touch_{ph}[V] = \texttt{nil} \\ \mathscr{E}[\ M_2\ ] & \text{if } touch_{ph}[V] \notin \{\texttt{nil}, \langle \textbf{ph}\ p \rangle\} \end{cases} \quad (if)$$

$$\mathscr{E}[\ (\textbf{future}\ N)\ ] \longmapsto_{ph}^{1,0} (\textbf{f-let}\ (p\ N)\ \mathscr{E}[\ \langle \textbf{ph}\ p \rangle\ ]) \quad p \notin FP(\mathscr{E}) \quad (fork)$$

$$(\textbf{f-let}\ (p\ V)\ S) \longmapsto_{ph}^{1,1} S[p := V] \quad (join)$$

The transition rules (*error*), (*future*), (*future-id*), (*future-error*), (*join-error*), (*reflexive*), (*transitive*) and (*parallel*) are as for the parallel machine

Fig. 4. The placeholder machine.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

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 $\langle \mathbf{ph}\ p \rangle$ is *free* if p is not bound by an enclosing **f-let** expression, and a state S is *closed* if it does not contain any free variables or free undetermined placeholder objects.

Terms in the placeholder machine include labels on placeholder-strict arguments. These labels will later be used by an analysis algorithm to identify the set of possible values of each placeholder-strict argument, and, in particular, to identify those placeholder-strict arguments that never yield placeholders. The labels do not have any effect on program evaluation.

4.2 Correctness of the placeholder machine

The state space of the placeholder machine is almost identical to that of the parallel machine, with the exception of placeholder objects and labels. Therefore, each placeholder machine state corresponds to a parallel machine state. We explicate this correspondence with a translation function Θ that maps placeholder machine states to corresponding parallel machine states. The translation replaces determined placeholder objects with the appropriate value; replaces undetermined placeholder objects with the corresponding placeholder variable; and removes labels on placeholder-strict operations.

$$
\begin{array}{rcl}
\Theta : State_{ph} & \longrightarrow & State_p \\
\Theta[\![\langle \mathbf{ph}\ p \rangle]\!] & = & p \\
\Theta[\![\langle \mathbf{ph}\ V \rangle]\!] & = & \Theta[\![V]\!] \\
\Theta[\![(\mathbf{apply}\ M_1^l\ M_2)]\!] & = & (\mathbf{apply}\ \Theta[\![M_1]\!]\ \Theta[\![M_2]\!]) \\
\cdots & = & \cdots \quad \text{similar clauses for other states}
\end{array}
$$

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

The function Θ is a *bisimulation relation* for the parallel and placeholder machines. That is, every transition of the placeholder machine corresponds to a transition of the parallel machine, and *vice versa.*

Lemma 4.1 (*Bisimulation Lemma*)
Let $S_1 \in State_p$ and $S_1' \in State_{ph}$ such that $\Theta[\![S_1']\!] = S_1$.

1. If $S_1' \longmapsto_{ph}^{n,m} S_2'$, then $S_1 \longmapsto_{p}^{n,m} \Theta[\![S_2']\!]$.
2. If $S_1 \longmapsto_{p}^{n,m} S_2$, then there exists $S_2' \in State_{ph}$ such that $S_1' \longmapsto_{ph}^{n,m} S_2'$ and $\Theta[\![S_2']\!] = S_2$.

Proof
We prove the first part by lexicographic induction on n and on the size of S_1', and by case analysis of the last step in $S_1' \longmapsto_{ph}^{n,m} S_2'$. 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 $S_1 \longmapsto_{p}^{n,m} S_2$. □

The Bisimulation Lemma implies the equivalence of the placeholder and parallel machines.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

```
(define fib
  (lambda (n)
    (if (< n 2)
      1
      (+(future (fib (-n 1)))
        (fib (-n 2))))))
```

Fig. 5. The definition of *fib*.

Theorem 4.2 (*Correctness of* $eval_{ph}$)
$eval_{ph} = eval_{p}$.

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

In summary, the placeholder machine correctly implements the parallel machine in that both machines specify 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 Touch optimization

The placeholder machine performs *touch* operations on arguments in placeholder-strict positions of all program operations. These implicit *touch* operations guarantee the transparency of placeholders, which makes **future**-based parallelism so convenient to use. Unfortunately, these (compiler-inserted) *touch* operations impose a significant overhead on the execution of annotated programs. For example, an annotated doubly-recursive version of *fib* (see figure 5) performs over a million *touch* operations during the computation of (*fib* 25).

Due to the dynamic typing of Scheme, the cost of each *touch* operation depends on the program operation that invoked it. If a program operation already performs a type dispatch to ensure that its arguments have the appropriate type, e.g. **car**, **cdr**, **apply**, *etc*, then a *touch* operation is free. Put differently, an implementation of (**car** *x*) in pseudo-code is:

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

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

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

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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

is incurred. Since the vast majority of Scheme operations already perform a type-dispatch on their arguments,[7] the overhead of performing implicit *touch* operations appears to be acceptable at first glance.

Still, a standard technique for increasing execution speed in Scheme systems is to disable type-checking typically based on informal correctness arguments or based on type verifiers for the underlying sequential language (Wright & Cartwright, 1994; Fagan, 1990; Aiken *et al*., 1994). When type-checking is disabled, most program operations do *not* perform a type-dispatch on their arguments. Under these circumstances, the source code (**car** *x*) translates to the pseudo-code:

```
(unchecked-car x)
```

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

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

Performing these *placeholder?* checks can add a significant overhead to the execution time. Kranz (1989) and Feeley (1993) estimated this cost at nearly 100% of the (sequential) execution time, and our experiments confirm these results (see below).

The classical solution for avoiding this overhead is to provide a compiler switch that disables the automatic insertion of *touch*es, and a *touch* primitive so that programmers can insert *touch* operations *explicitly* where needed (Feeley, 1993; Kessler & Swanson, 1989; Swanson *et al*., 1988). We believe that this solution is flawed for two reasons. First, it clearly destroys the transparent character of **future** annotations. Instead of an annotation that only affects executions on some implementations, **future** is now a task creation construct and *touch* is a synchronization tool. Secondly, to use this solution safely, the programmer must know where placeholders can appear instead of regular values and must add *touch* operations at these places in the program. In contrast to the addition of **future** annotations, the placement of *touch* operations is far more difficult: while the former requires a prediction concerning computational intensity, the latter demands a full understanding of the data flow properties of the program. Since we believe that an accurate prediction of data flow by the programmer is only possible for small programs, we reject this traditional solution.

A better approach than explicit *touch*es is for the compiler to use information provided by a data-flow analysis of the program to remove unnecessary *touch*es wherever provably possible.[8] This approach substantially reduces the overhead of *touch* operations without sacrificing the simplicity or transparency of **future** annotations.

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

[8] This approach is similar in character to optimization techniques such as soft-typing (Wright & Cartwright, 1994; Fagan, 1990; Aiken *et al*., 1994) and tagging optimization (Henglein, 1992), which remove the type-dispatches required in dynamically-typed languages wherever possible. However, touch-optimization also applies to a statically-typed language with **future**s, where soft-typing and tagging optimization techniques are not as useful.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

$$\mathscr{E}[\ (\underline{\mathbf{car}}\ V)\ ] \longmapsto^{1,1}_{ph} \begin{cases} \mathscr{E}[\ V_1\ ] & \text{if } V = (\mathbf{cons}\ V_1\ V_2) \\ \text{unspecified} & \text{if } V \in PHValue_{ph} \\ \texttt{error} & \text{otherwise} \end{cases} \quad (\underline{car})$$

$$\mathscr{E}[\ (\underline{\mathbf{if}}\ V\ M_1\ M_2)\ ] \longmapsto^{1,1}_{ph} \begin{cases} \mathscr{E}[\ M_1\ ] & \text{if } V = \texttt{nil} \\ \text{unspecified} & \text{if } V \in PHValue_{ph} \\ \mathscr{E}[\ M_2\ ] & \text{otherwise} \end{cases} \quad (\underline{if})$$

$$\mathscr{E}[\ (\underline{\mathbf{apply}}\ V_1\ V_2)\ ] \longmapsto^{1,1}_{ph} \begin{cases} \mathscr{E}[\ M[x \leftarrow V_2]\ ] & \text{if } V_1 = (\lambda x.\, M) \\ \text{unspecified} & \text{if } V_1 \in PHValue_{ph} \\ \texttt{error} & \text{otherwise} \end{cases} \quad (\underline{apply})$$

Fig. 6. Non-*touch*ing transition rules.

5.1 Non-touching primitives

The current language does not provide primitives that do not *touch* arguments in placeholder-strict positions. To express and verify an algorithm that replaces *touch*ing primitives by non-*touch*ing primitives, we extend the language Λ_{ph} with non-touching forms of the placeholder-strict primitive operations, denoted **car**, **cdr**, **if** and **apply**, respectively:[9]

$$\begin{aligned} M \quad ::= \quad & \ldots \\ & \mid (\underline{\mathbf{car}}\ M) \mid (\underline{\mathbf{cdr}}\ M) \\ & \mid (\underline{\mathbf{if}}\ M\ M\ M) \\ & \mid (\underline{\mathbf{apply}}\ M\ M) \end{aligned}$$

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

We define the semantics of the extended language $\underline{\Lambda}$ by extending the placeholder machine with the additional transition rules described in figure 6. The evaluator for the extended language, $\underline{eval_{ph}}$, is defined in the usual way (compare figure 4). Unlike $eval_{ph}$, the evaluator $\underline{eval_{ph}}$ is no longer a function. There are programs in $\underline{\Lambda}$ for which the evaluator $\underline{eval_{ph}}$ can either return a value or can be unspecified because of the application of a non-*touch*ing operation to a placeholder. Still, the two evaluators clearly agree on programs in Λ_{ph}.

Lemma 5.1

For $P \in \Lambda_{ph}$, $\underline{eval_{ph}}(P) = eval_{ph}(P)$.

[9] Since these operations do not perform a *touch* operation on their arguments, we do not associate a label with those arguments.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

5.2 The touch optimization algorithm

The goal of *touch* optimization is to replace the touching operations **car**, **cdr**, **if** and **apply** by the corresponding non-touching operation whenever possible *without* changing the semantics of programs. For example, suppose that a program contains (**car** M^l) and we can prove that M never evaluates to a placeholder. Then we can replace (**car** M^l) with (**car** M), which the machine can execute without performing a test for placeholdership on the result of M.

This optimization technique relies on a detailed data-flow analysis of the program that determines a conservative approximation to the set of run-time values for each expression in a placeholder-strict position. We assume that the analysis returns a *placeholder table*, which is a table of labels identifying all labeled expressions in the program that may yield placeholder objects.

Definition 5.2
Placeholder table, valid placeholder table

- A *placeholder table* $\mathscr{P}$ is a subset of *Label*.
- A placeholder table $\mathscr{P}$ is *currently valid* for a state S (denoted $S \models_c \mathscr{P}$) if for every term of the form $\langle \mathbf{ph}\ p \rangle^l$ or $\langle \mathbf{ph}\ V \rangle^l$ in S, $l \in \mathscr{P}$
- A placeholder table $\mathscr{P}$ is (*always*) *valid* for a program P (denoted $P \models_a \mathscr{P}$) if $P \longmapsto^*_{ph} S$ implies that $S \models_c \mathscr{P}$.

The basic idea behind *touch* optimization is now easy to explain. If a placeholder table that is always valid for a program shows that the argument of a *touch*ing version of **car**, **cdr**, **if** or **apply** can never be a placeholder, the optimization algorithm replaces the operation with its non-*touch*ing version. The optimization algorithm $\mathscr{T}$, parameterized over a placeholder table $\mathscr{P}$, is defined in figure 7. If $\mathscr{P}$ is always valid for the program being optimized, then the touch optimization algorithm $\mathscr{T}_{\mathscr{P}}$ preserves the meaning of that program. For every transition step of the source program P there exists a corresponding transition step for the optimized program $\mathscr{T}_{\mathscr{P}}(P)$. We extend $\mathscr{T}_{\mathscr{P}}$ in a natural manner to states to aid in the proof of this correspondence.

Lemma 5.3 (*Step Correspondence*)
Let S be a placeholder machine state for which the placeholder table $\mathscr{P}$ is currently valid.

1. Suppose $S \longmapsto^{l,l}_{ph} S'$. Then $\mathscr{T}_{\mathscr{P}}(S) \longmapsto^{l,l}_{ph} \mathscr{T}_{\mathscr{P}}(S')$.
2. Suppose $\mathscr{T}_{\mathscr{P}}(S) \longmapsto^{l,l}_{ph} S''$. Then $S \longmapsto^{l,l}_{ph} S'$ where $\mathscr{T}_{\mathscr{P}}(S') = S''$.

Proof
We prove the first part by case analysis of $S \longmapsto^{l,l}_{ph} S'$, and the second part by case analysis of $\mathscr{T}_{\mathscr{P}}(S) \longmapsto^{l,l}_{ph} S''$. □

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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

$$
\begin{array}{rcll}
\mathscr{T}_{\mathscr{P}} : \Lambda_{ph} & \longrightarrow & \underline{\Lambda_{ph}} & \\
\mathscr{T}_{\mathscr{P}}[x] & = & x & \\
\mathscr{T}_{\mathscr{P}}[c] & = & c & \\
\mathscr{T}_{\mathscr{P}}[(\lambda x.\, M)] & = & (\lambda x.\, \mathscr{T}_{\mathscr{P}}[M]) & \\
\mathscr{T}_{\mathscr{P}}[(\mathbf{cons}\ M_1\ M_2)] & = & (\mathbf{cons}\ \mathscr{T}_{\mathscr{P}}[M_1]\ \mathscr{T}_{\mathscr{P}}[M_2]) & \\
\mathscr{T}_{\mathscr{P}}[(\mathbf{future}\ M)] & = & (\mathbf{future}\ \mathscr{T}_{\mathscr{P}}[M]) & \\
\mathscr{T}_{\mathscr{P}}[(\mathbf{car}\ M^{l})] & = & \begin{cases} (\underline{\mathbf{car}}\ \mathscr{T}_{\mathscr{P}}[M]) & \text{if } l \notin \mathscr{P} \\ (\mathbf{car}\ \mathscr{T}_{\mathscr{P}}[M]^{l}) & \text{if } l \in \mathscr{P} \end{cases} & \\
\mathscr{T}_{\mathscr{P}}[(\mathbf{if}\ M^{l}\ M_1\ M_2)] & = & \begin{cases} (\underline{\mathbf{if}}\ \mathscr{T}_{\mathscr{P}}[M]\ \mathscr{T}_{\mathscr{P}}[M_1]\ \mathscr{T}_{\mathscr{P}}[M_2]) & \text{if } l \notin \mathscr{P} \\ (\mathbf{if}\ \mathscr{T}_{\mathscr{P}}[M]^{l}\ \mathscr{T}_{\mathscr{P}}[M_1]\ \mathscr{T}_{\mathscr{P}}[M_2]) & \text{if } l \in \mathscr{P} \end{cases} & \\
\mathscr{T}_{\mathscr{P}}[(\mathbf{apply}\ M^{l}\ N)] & = & \begin{cases} (\underline{\mathbf{apply}}\ \mathscr{T}_{\mathscr{P}}[M]\ \mathscr{T}_{\mathscr{P}}[N]) & \text{if } l \notin \mathscr{P} \\ (\mathbf{apply}\ \mathscr{T}_{\mathscr{P}}[M]^{l}\ \mathscr{T}_{\mathscr{P}}[N]) & \text{if } l \in \mathscr{P} \end{cases} &
\end{array}
$$

Fig. 7. The touch optimization algorithm $\mathscr{T}$.

Theorem 5.4

For $P \in \Lambda^0_{ph}$, if $P \models_a \mathscr{P}$ then $\underline{eval_{ph}}(\mathscr{T}_{\mathscr{P}}(P)) = eval_{ph}(P)$.

Proof

Both the fact that $\underline{eval_{ph}}$ is well-defined on $\mathscr{T}_{\mathscr{P}}(P)$ and that the equality holds follow from Lemma 5.3. □

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

6 Set-based analysis for futures

The touch optimization algorithm just described relies on an analysis that infers an always valid placeholder table for the program to be optimized. This section presents such an analysis, which is a variant of Heintze's set-based analysis (Heintze, 1994), appropriately adapted to our language with **future**s.

Set-based analysis consists of two co-mingled phases: a *specification* phase and a *solution* phase.[10] The specification phase derives *constraints* on the sets of values that program expressions may assume. These constraints describe the data flow relationships of the analyzed program. The solution phase solves these constraints to yield an always valid placeholder table for the analyzed program.

[10] Cousot and Cousot (1995) showed that the results of set-based analysis can alternatively be computed via an abstract interpretation based on chaotic iteration.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

6.1 The constraint language

The language of *set expressions* is described by the following grammar.

$$\begin{array}{rcccl} \tau & \in & \mathit{SetExp} & = & c \mid \mathtt{dom}(\tau) \mid \mathtt{rng}(\tau) \mid \mathtt{car}(\tau) \mid \mathtt{cdr}(\tau) \mid \langle \mathbf{ph} \rangle \mid \alpha \\ \alpha, \beta, \gamma & \in & \mathit{SetVar} & \supset & \mathit{Label} \end{array}$$

Intuitively, each set expression τ denotes a set of values. A constant c denotes the corresponding singleton set. The set expressions $\mathtt{dom}(\tau)$ and $\mathtt{rng}(\tau)$ denote the argument set and result set, respectively, of functions denoted by τ. Similarly, $\mathtt{car}(\tau)$ and $\mathtt{cdr}(\tau)$ denote the first and second components, respectively, of pairs denoted by τ. The set expression $\langle \mathbf{ph} \rangle$ denotes a value set that contains placeholders. The meta-variables α, β, γ range over set variables, and we include program labels in the collection of set variables.

A *constraint* $\mathscr{C}$ is an inequality $\tau_1 \leqslant \tau_2$ relating two set expressions, and intuitively it denotes a corresponding set containment relationship between the value sets for τ_1 and τ_2. A *constraint system* $\mathscr{S}$ is a collection of constraints.

$$\begin{array}{rcccl} \mathscr{C} & \in & \mathit{Constraint} & = & \tau_1 \leqslant \tau_2 \\ \mathscr{S} & \in & \mathit{Constraint}^* & = & \mathscr{P}_{\mathrm{fin}}(\mathit{Constraint}) \end{array}$$

6.2 The specification phase

The specification phase of set-based analysis derives constraints on the sets of values that program expressions may assume. Following Aiken *et al.* (1994) and Palsberg and O'Keefe (1995), we formulate this derivation as a proof system.

The derivation proceeds in a syntax-directed manner according to the constraint derivation rules presented in figure 8. Each rule infers a judgment of the form $\Gamma \vdash M : \alpha, \mathscr{S}$, where:

1. the *derivation context* Γ maps the free variables of M to set variables;
2. α names the value set of M; and
3. $\mathscr{S}$ is a constraint system describing the data-flow relationships of M, using α.

The constraint derivation rule (*const*) generates the constraint $c \leqslant \alpha$, which ensures that the value set for a constant expression contains that constant. The (*var*) rule extracts the appropriate set variable α for a particular program variable x from the derivation context. The (*abs*) rule for functions propagates values from the function's domain into its formal parameter and from the function's body into its range. The (*app*) rule for applications propagates values from the argument expression into the domain of the applied function and from the range of that function into the result of the application expression. This rule also records values for the function sub-expression in the appropriate label. The (*cons*) rule for pairs records the possible values for the pairs components. The (*car*) rule extracts the possible values for the first component of a pair. The (*cdr*) rule is similar, but is omitted for brevity.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

$$\Gamma \vdash c : \alpha, \{c \leqslant \alpha\} \qquad (const)$$

$$\Gamma \cup \{x : \alpha\} \vdash x : \alpha, \emptyset \qquad (var)$$

$$\frac{\Gamma \cup \{x : \beta_1\} \vdash M : \beta_2, \mathscr{S}}{\Gamma \vdash (\lambda x.\, M) : \alpha, \mathscr{S} \cup \left\{ \begin{array}{rcl} \mathtt{dom}(\alpha) & \leqslant & \beta_1 \\ \beta_2 & \leqslant & \mathtt{rng}(\alpha) \end{array} \right\}} \qquad (abs)$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathscr{S}_i \quad \text{for } i = 1, 2}{\Gamma \vdash (\mathbf{apply}\ M_1^l\ M_2) : \alpha, \mathscr{S}_1 \cup \mathscr{S}_2 \cup \left\{ \begin{array}{rcl} \beta_1 & \leqslant & l \\ \beta_2 & \leqslant & \mathtt{dom}(\beta_1) \\ \mathtt{rng}(\beta_1) & \leqslant & \alpha \end{array} \right\}} \qquad (app)$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathscr{S}_i \quad \text{for } i = 1, 2}{\Gamma \vdash (\mathbf{cons}\ M_1\ M_2) : \alpha, \mathscr{S}_1 \cup \mathscr{S}_2 \cup \left\{ \begin{array}{rcl} \beta_1 & \leqslant & \mathtt{car}(\alpha) \\ \beta_2 & \leqslant & \mathtt{cdr}(\alpha) \end{array} \right\}} \qquad (cons)$$

$$\frac{\Gamma \vdash M : \beta, \mathscr{S}}{\Gamma \vdash (\mathbf{car}\ M^l) : \alpha, \mathscr{S} \cup \{\mathtt{car}(\beta) \leqslant \alpha,\ \beta \leqslant l\}} \qquad (car)$$

$$\Gamma \vdash \langle \mathbf{ph}\ p \rangle : \alpha, \{\langle \mathbf{ph} \rangle \leqslant \alpha\} \qquad (undef\text{-}ph)$$

$$\frac{\Gamma \vdash V : \beta, \mathscr{S}}{\Gamma \vdash \langle \mathbf{ph}\ V \rangle : \alpha, \mathscr{S} \cup \{\beta \leqslant \alpha,\ \langle \mathbf{ph} \rangle \leqslant \alpha\}} \qquad (def\text{-}ph)$$

$$\frac{\Gamma \vdash M : \beta, \mathscr{S}}{\Gamma \vdash (\mathbf{future}\ M) : \alpha, \mathscr{S} \cup \{\beta \leqslant \alpha,\ \langle \mathbf{ph} \rangle \leqslant \alpha\}} \qquad (future)$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathscr{S}_i \quad \text{for } i = 1, 2, 3}{\Gamma \vdash (\mathbf{if}\ M_1^l\ M_2\ M_3) : \alpha, \mathscr{S}_1 \cup \mathscr{S}_2 \cup \mathscr{S}_3 \cup \{\beta_1 \leqslant l,\ \beta_2 \leqslant \alpha,\ \beta_3 \leqslant \alpha\}} \qquad (if)$$

Fig. 8. Constraint derivation rules.

The (*undef-ph*) rule generates the constraint $\langle \mathbf{ph} \rangle \leqslant \alpha$, which ensures that the value set of an undefined placeholder contains placeholders. The (*def-ph*) rule records that the value set of a defined placeholder contains both the placeholder, and the value that the placeholder would yield after a *touch* operation. The (*future*) rule records that a **future** expression can yield both a placeholder and the value produced by the **future** body. The (*if*) rule records possible values of the test expression in the appropriate label, and also records that the **if**-expression can return the result of either the *then* or *else* subexpression.

Many of the constraint derivation rules introduce new set variables. For example, the rule (*const*) introduces the new set variable α. Whenever this rule is applied, we need to choose a fresh set variable for α that is not used elsewhere in the constraint derivation. Choosing a fresh set variable in this manner yields a more accurate analysis.

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

$$\frac{c \leqslant \beta \qquad \beta \leqslant \gamma}{c \leqslant \gamma} \quad (s_1) \qquad \frac{\langle \mathbf{ph} \rangle \leqslant \beta \qquad \beta \leqslant \gamma}{\langle \mathbf{ph} \rangle \leqslant \gamma} \quad (s_2)$$

$$\frac{\alpha \leqslant \mathtt{car}(\beta) \qquad \beta \leqslant \gamma}{\alpha \leqslant \mathtt{car}(\gamma)} \quad (s_3) \qquad \frac{\alpha \leqslant \mathtt{cdr}(\beta) \qquad \beta \leqslant \gamma}{\alpha \leqslant \mathtt{cdr}(\gamma)} \quad (s_4)$$

$$\frac{\alpha \leqslant \mathtt{rng}(\beta) \qquad \beta \leqslant \gamma}{\alpha \leqslant \mathtt{rng}(\gamma)} \quad (s_5) \qquad \frac{\mathtt{dom}(\beta) \leqslant \alpha \qquad \beta \leqslant \gamma}{\mathtt{dom}(\gamma) \leqslant \alpha} \quad (s_6)$$

$$\frac{\alpha \leqslant \mathtt{car}(\beta) \qquad \mathtt{car}(\beta) \leqslant \gamma}{\alpha \leqslant \gamma} \quad (s_7) \qquad \frac{\alpha \leqslant \mathtt{cdr}(\beta) \qquad \mathtt{cdr}(\beta) \leqslant \gamma}{\alpha \leqslant \gamma} \quad (s_8)$$

$$\frac{\alpha \leqslant \mathtt{rng}(\beta) \qquad \mathtt{rng}(\beta) \leqslant \gamma}{\alpha \leqslant \gamma} \quad (s_9) \qquad \frac{\alpha \leqslant \mathtt{dom}(\beta) \qquad \mathtt{dom}(\beta) \leqslant \gamma}{\alpha \leqslant \gamma} \quad (s_{10})$$

Fig. 9. The rules $\Theta = \{s_1, \ldots, s_{10}\}$.

6.3 Solving set constraints

To solve a constraint system, we close it under the rules Θ described in figure 9. Intuitively, these rules infer all the data-flow paths in the program, which are described by constraints of the form $\beta \leqslant \gamma$ (for $\beta, \gamma \in \mathit{SetVar}$), and propagate values along those data-flow paths. Specifically, the rules (s_1) to (s_6) propagate information about constants, placeholders, pairs, and function domains and ranges forward along the data-flow paths of the program. The rules (s_7) and (s_8) constructs the data-flow paths from the components of a pair to all accesses of that component. The rule (s_9) constructs data-flow paths from function results to corresponding call sites, and the rule (s_{10}) similarly constructs data-flow paths from actual to formal parameters for each function call. We write $\mathscr{S} \vdash_\Theta \mathscr{C}$ if $\mathscr{S}$ proves $\mathscr{C}$ via the rules Θ, and use $\mathit{close}_\Theta(\mathscr{S})$ to denote the closure of $\mathscr{S}$ under Θ, i.e. the set $\{\mathscr{C} \mid \mathscr{S} \vdash_\Theta \mathscr{C}\}$.

We use a worklist algorithm to compute the closure of $\mathscr{S}$ under Θ efficiently. The worklist keeps track of all constraints in $\mathscr{S}$ whose consequences under Θ may not be in $\mathscr{S}$. The algorithm repeatedly removes a constraint from the worklist, and for each consequences under Θ that is not already in $\mathscr{S}$, it adds that consequence both to $\mathscr{S}$ and to the worklist. The process iterates until the worklist is empty, at which point $\mathscr{S}$ is closed under Θ. The complete algorithm can be found in the first author's dissertation (Flanagan, 1997).

This closure process propagates all information concerning the possible placeholder values for labeled expressions into constraints of the form $\langle \mathbf{ph} \rangle \leqslant l$. Hence, we can infer from $\mathit{close}_\Theta(\mathscr{S})$ a placeholder table that is always valid for the analysed program, as follows.

Definition 6.1 ($\mathscr{P}(P)$)
For $P \in \Lambda_{ph}$ with $\emptyset \vdash P : \alpha, \mathscr{S}$,

$$\mathscr{P}(P) = \{l \mid [\langle \mathbf{ph} \rangle \leqslant l] \in \mathit{close}_\Theta(\mathscr{S})\} \ .$$

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Theorem 6.2 (*Correctness of* $\mathscr{P}(P)$)
The placeholder table $\mathscr{P}(P)$ is always valid for P:

$$P \models_a \mathscr{P}(P) .$$

Proof
The correctness of this theorem follows from a subject reduction proof along the lines described in the first author's dissertation (Flanagan, 1997). □

7 Experimental results

We extended the Gambit compiler (Feeley, 1993; Feeley & Miller, 1990), which makes no attempt to remove *touch* operations from programs, with a preprocessor that implements the set-based analysis algorithm and the *touch* optimization algorithm. The analysis and the optimization algorithm are as described in the previous sections extended to a sufficiently large subset of functional Scheme.[11] We used the extended Gambit compiler to test the effectiveness of *touch* optimization on the suite of benchmarks contained in Feeley's PhD thesis (1993) on a GP1000 shared-memory multiprocessor (BBN Advanced Computers, 1989). Figure 10 describes these benchmarks.

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

Fig. 10. Description of the benchmark programs.

Each benchmark was tested on the original compiler (*standard*) and on the modified compiler (*touch optimized*). The results of the test runs are documented in figure 11. The first two columns present the number of *touch* operations performed during the execution of a benchmark using the *standard* compiler (column 1), and the sequential execution overhead of these *touch* operations (column 2). To determine the absolute overhead of *touch*, we also ran the programs on a single

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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

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

Fig. 11. Benchmark results.

processor after removing all *touch* operations. The next two columns contain the corresponding measurements for the *touch optimizing* compiler. The *touch* optimization algorithm reduces the number of *touch* operations to a small fraction of the original number (column 3), thus reducing the average overhead of *touch* operations from approximately 90% to less than 10% (column 4).

The last three columns show the relative speedup of each benchmark for one, four, and 16 processor configurations, respectively. The number compares the running time of the benchmarks using the standard compiler with the optimizing compiler. As expected, the relative speedup *decreases* as the number of processors *increases*, because the execution time is then dominated by other factors, such as memory contention and communication costs. For most benchmarks, the benefit of our *touch* optimization is still substantial, producing an average speedup over the *standard* compiler of 37% on four processors, and of 20% on 16 processors. The exceptions are the last three benchmarks, `mst`, `qsort`, and `poly`. However, even Feeley (1993) described these as 'poorly parallel' programs, in which the effects of memory contention and communication costs are especially visible. It is therefore not surprising that our optimizing compiler does not improve the running time in these cases.

8 Related work

The literature on programming languages contains a number of descriptions of the semantics of parallel Scheme-like languages. The only one that directly deals with parallelism based on transparent annotations is Moreau's PhD thesis (1994b; 1994a). 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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

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 (1986); the implementation is a parallel version of the CESK machine (Felleisen & Friedman, 1987; Felleisen & Hieb, 1989) that implements placeholders as globally accessible reference cells. The equivalence proof establishes that both evaluators define the same observable equivalence relation via the construction of a number of intermediate calculi and machines. It is far more complicated than our diamond and bisimulation techniques. Moreau later adopted the simpler approach described in this paper, and extended it to handle first-class continuations and mutable state (Moreau, 1996).

Independently, Reppy (1992) and Leroy (1992) define operational semantics for ML-like languages with first-class synchronization operations. Reppy's language, Concurrent ML, can express **future** as an abstraction over the primitives of the language. The semantics is a two-level rewriting system. The first level, also a program rewriting system in the tradition of Felleisen and Friedman (1986; 1987; 1989), accounts for the sequential behavior; the second level specifies the behavior of *sets* of parallel tasks and the task communication mechanisms. Reppy proves 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 (1994) define an operational semantics for a simple functional 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 (1977)), they extend Deutsch's transition semantics (Deutsch, 1992) 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.

Wand (1995) 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 parallel and placeholder machines.

Kranz *et al.* (1989) briefly describe a simplistic algorithm for *touch* optimization based on a first-order type analysis. The algorithm lowers the *touch* overhead to 65% from 100% in standard benchmarks, that is, it is significantly less effective than our *touch* optimization. The paper does not address the semantics of **future** or the well-foundedness of the optimizations. Knopp (1989) reports the existence of a *touch* optimization algorithm based on abstract interpretation. His paper presents neither a semantics nor the abstract interpretation. He only reports the reduction of *static* counts of *touch* operations for an implementation of Common Lisp with **future**. Neither paper gives an indication concerning the expense of the analysis algorithms.[12]

Much work has been done on the static analysis of *sequential* programs. Our analysis most closely follows Heintze's work on set-based analysis for the sequential language ML (Heintze, 1992), but the extension of this technique to parallel languages requires a substantial reformulation of the derivation and correctness proof. Specifically, Heintze uses the 'natural' semantics framework to define a set-based 'natural' semantics, from which he reads off 'safeness' conditions on set environments. He then presents set constraints whose solution is the minimal safe set environment. We start from an parallel abstract machine and avoid Heintze's intermediate steps by deriving our set constraints and proving their correctness directly from the abstract machine semantics.

Other techniques for static analysis of sequential programs include abstract interpretation (Cousot & Cousot, 1977; Cousot & Cousot, 1994) and Shivers' 0CFA (Shivers, 1991). The relationship between abstract interpretation and set-based analysis was covered by Heintze (Heintze, 1992).

Sequential optimization techniques such as tagging optimization (Henglein, 1992) and soft-typing (Wright & Cartwright, 1994; Fagan, 1990; Aiken *et al.*, 1994) are similar in character to *touch* optimization. Both techniques remove the type-dispatches required for dynamic type-checking wherever possible, without changing the behavior of programs, in the same fashion as we remove *touch* operations. However, the analyses relys on conventional type inference techniques, whereas ours exploits the Scheme view of types as sets directly.

9 Conclusion

The development of a semantics for **future**s directly leads to the derivation of a powerful program analysis. The analysis is computationally inexpensive but yields enough information to eliminate numerous implicit *touch* operations. We believe that the construction of this simple *touch* optimization algorithm clearly illustrates how semantics can contribute to the development of advanced compilers.

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

https://doi.org/10.1017/S0956796899003329 Published online by Cambridge University Press

Acknowledgements

We thank Marc Feeley for discussions concerning *touch* optimizations and for his assistance in testing the effectiveness of our algorithm, Nevin Heintze for discussions on set-based analysis and for access to his implementation of set based analysis for ML, and Luc Moreau for very careful reading of our proofs. Phil Wadler and the anonymous referees made many important suggestions.

References

Aiken, A., Wimmers, E. L. and Lakshman, T. K. (1994) Soft typing with conditional types. *Principles of Programming Languages*, pp. 163–173.

Baker, H. G. and Hewitt, C. (1977) The incremental garbage collection of processes. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pp. 55–59. *SIGPLAN Notices* **12**(8).

BBN Advanced Computers (1989) *Inside the gp1000*, BBN Advanced Computers, Inc., Cambridge, MA.

Cousot, P. and Cousot, R. (1977) Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints. *Principles of Programming Languages*, pp. 238–252.

Cousot, P. and Cousot, R. (1995) Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. *Functional Programming and Computer Architecture*, pp. 170–181.

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

Deutsch, A. (1992) *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.

Fagan, M. (1990) *Soft typing.* PhD thesis, Rice University.

Feeley, M. (1993) *An efficient and general implementation of futures on large scale shared-memory multiprocessors.* PhD thesis, Department of Computer Science, Brandeis University.

Feeley, M. and Miller, J. S. (1990) A parallel virtual machine for efficient scheme compilation. *Lisp and Functional Programming*.

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

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

Felleisen, M. and Hieb, R. (1989) *The revised report on the syntactic theories of sequential control and state.* Technical Report 100, Rice University. (*Theor. Comput. Sci.* **102**, 1992.)

Flanagan, C. (1997) *Effective static debugging via componential set-based analysis.* PhD thesis, Rice University, Houston, TX.

Flanagan, C. and Felleisen, M. (1994a) *The semantics of Future.* Rice University Computer Science TR94-238.

Flanagan, C. and Felleisen, M. (1994b) *Well-founded touch optimization for futures.* Rice University Computer Science TR94-239.

Flanagan, C. and Felleisen, M. (1995) The semantics of future and its use in program optimizations. *Principles of Programming Languages*, pp. 209–220.

Flanagan, C. and Nikhil, R. S. (1996) *pHluid*: The design of a parallel functional language implementation on workstations. *International Conference on Functional Programming*.

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

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

Heintze, N. (1992) *Set based program analysis*. PhD thesis, Carnegie Mellon University.

Heintze, N. (1994) Set-based analysis of ML programs. *Lisp and Functional Programming*, pp. 306–317.

Henglein, F. (1992) Global tagging optimization by type inference. *Lisp and Functional Programming*, pp. 205–215.

Ito, T. and Halstead, R. H. (eds). (1989) *Parallel Lisp: Languages and Systems: Lecture Notes in Computer Science 441*. Springer-Verlag.

Ito, T. and Matsui, M. (1989) *A parallel Lisp language: Pailisp and its kernel specification: Lecture Notes in Computer Science 441*. Springer-Verlag, pp. 58–100.

Jagannathan, S. and Weeks, S. (1994) Analyzing stores and references in a parallel symbolic language. *Lisp and Functional Programming*, pp. 294–305.

Katz, M. and Weise, D. (1990) Continuing into the future: on the interaction of futures and first-class continuations. *Lisp and Functional Programming*.

Kessler, R. R. and Swanson, R. (1989) *Concurrent scheme: Lecture Notes in Computer Science 441*. S[pringer-Verlag, pp. 200–234.

Knopp, J. (1989) *Improving the performance of parallel Lisp by compile time analysis: Lecture Notes in Computer Science 441*. Springer-Verlag, pp. 271–277.

Kranz, D. A., Halstead, R. H. and Mohr, E. (1989) *Mul-T: A high-performance parallel Lisp: Lecture Notes in Computer Science 441*. Springer-Verlag, pp. 306–321.

Leroy, X. (1992) *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7.

Miller, J. (1987) *Multischeme: A parallel processing system*. PhD thesis, MIT.

Mohr, E., Kranz, R. and Halstead, R. (1990) Lazy task creation: A technique for increasing the granularity of parallel programs. *Lisp and Functional Programming*.

Moreau, L. (1994a) The PCKS-machine. an abstract machine for sound evaluation of parallel functional programs with first-class continuations. *European Symposium on Programming (ESOP'94): Lecture Notes in Computer Science 788*, pp. 424–438. Springer-Verlag.

Moreau, L. (1994b) *Sound evaluation of parallel functional programs with first-class continuations*. PhD thesis, Universite de Liege.

Moreau, L. (1996) The semantics of scheme with future. *International Conference on Functional Programming*, pp. 146–156.

Shivers, O. (1991) *Control-flow analysis of higher-order languages, or taming lambda*. PhD thesis, Carnegie-Mellon University.

Palsberg, J. and O'Keefe, P. (1995) A type system equivalent to flow analysis. *Principles of Programming Languages*, pp. 367–378.

Plotkin, G. D. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 125–159.

Reppy, J. H. (1992) *Higher-order concurrency*. PhD thesis, Cornell University.

Swanson, M. R., Kessler, R. R. & Lindstrom, G. (1988) An implementation of portable standard lisp on the BBN Butterfly. *Lisp and Functional Programming*, pp. 132–142.

Wand, M. (1995) *Compiler correctness for parallel languages*. Unpublished manuscript.

Wright, A. and Cartwright, R. (1994) A practical soft type system for scheme. *Lisp and Functional Programming*, pp. 250–262.