The Semantics of Scheme with Future

Luc Moreau*
University of Southampton
L.Moreau@ecs.soton.ac.uk

Abstract

We present the formal semantics of future in a Scheme-like language which has both side-effects and first-class continuations. Correctness is established by proving that programs annotated by future have the same observable behaviour as their non-annotated counterparts, even though evaluation may be parallel.

1 Introduction

MultiLisp future [1, 8] is an annotation by which the programmer indicates that some expression may be evaluated in parallel. By definition, future-based programs have the same observable behaviour as their non-annotated counterparts, i.e. annotated programs return the same result as in the absence of annotations, even though evaluation may be parallel.

It is a delicate matter to design a language with futures and *effects*, i.e. with side-effects and first-class continuations: as the values of some programs using effects may depend on the evaluation order, incautiously adding parallelism would make them non-deterministic, which would contradict the idea that future is an annotation.

So far, implementation and efficiency questions [13, 25, 8, 12, 10, 11, 3] have mainly motivated research on annotations for parallelism in Lisp-like languages. Recently only, two semantics of parallelism by annotations were proposed. The author of this paper defined a semantic framework for functional programs with first-class continuations and pcall annotations [15, 16, 17]. Flanagan and Felleisen [7] formulated the semantics of future in a purely functional language. However, the issue of future in a language that has side-effects and first-class continuations remains unaddressed.

*This research was supported in part by the Engineering and Physical Sciences Research Council, grant GR/K30773. Author's address: Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ. United Kingdom.

The goal of this paper is to formalise the semantics of future in a Scheme-like [22] language with side-effects and first-class continuations. More generally the motivation of our research is to design a distributed implementation of Scheme based on a future-like annotation to create remote computations. This paper presents the semantic foundation of our language, which serves as a guideline to build our distributed system. Distribution considerations are beyond the scope of this paper and will be covered in a forthcoming report.

The major contributions of this paper are three operational semantics which highlight different aspects of a programming language with future. The first semantics is the sequential semantics of the language which regards future as a transparent annotation without parallelism-related meaning. By transparent, we mean that the value of future is the value of its argument. The second semantics interprets future as a construct that indicates that some expression may be evaluated in parallel; it embodies the technique to coordinate effects in a semantically sound way. The first two semantics are context-rewriting machines [4, 7], which are advantageously high-level and concise. The third semantics is a lower-level refinement of the second semantics. It features an explicit shared memory à la MultiLisp [8], *placeholder* data-structures, and a notion of *legitimacy* [12] to assess the validity of results and the soundness of side-effects. All semantics are proved to be equivalent; proofs can be found in [18].

This paper is organised as follows. The three semantics are presented in Sections 2, 3, and 4, respectively. Section 5 discusses related work, and is followed by a conclusion.

2 The CS-Machine

The syntax of our idealised Scheme-like language is shown in Figure 1: Λ_f is an applied call-by-value lambda calculus extended with primitives for manipulating first-class boxes and continuations. In addition, Λ_f has a future construct written as (future M). In the sequel, we adopt Barendregt's [2] definitions and conventions on the lambda-calculus; we shall use $M[x \leftarrow V]$ to denote the substitution of V for the free occurrences of x in M. Furthermore, as the semantics that we propose generalise Flanagan and Felleisen's [7] semantics of future, we try to use their notations and termi-

$P \in \Lambda_f^0$			(Program)
$M \in \Lambda_f$	::=	$V \mid (M\ M) \mid (\mathsf{if}\ M\ M\ M) \mid (\mathsf{future}\ M)$	(Term)
$V \in Value_f$	::=	$a \mid f \mid x \mid (\lambda x.M)$	(Value)
$a \in BConst$	=	$\{\mathsf{true}, \mathsf{false}, \mathsf{nil}, 0, 1, \ldots\}$	(Basic Constant)
$f \in FConst$	=	$\{\mathsf{cons}, \mathsf{car}, \mathsf{cdr}, \mathsf{makeref}, \mathsf{deref}, \mathsf{setref!}, \mathsf{callcc}\}$	(Functional Constant)
$x \in Vars$	=	$\{x, y, z \ldots\}$	(User Variable)

Figure 1: Syntax of Λ_f

nology wherever possible.

The first operational semantics is given by the CS-machine, derived from Felleisen, Friedman, Hieb, and Sabry's [4, 6, 23] λ_v-CS-calculi. Its state space and evaluator specification are displayed in Figures 2 and 3, respectively. The CS-machine is a *context-rewriting* machine that uses the notion of evaluation context [4]: an evaluation context $\mathcal{E}$ is a term with a "hole", [ ], in place of the next subterm to evaluate.

The transition relation $\mapsto_{cs}$ maps states onto states. States are expressions of the type $(\mathsf{letref}\ \theta\ M)$. The construct $(\mathsf{letref}\ \theta\ [\,])$ corresponds to Felleisen and Hieb's [6] ρ-notation, where the store θ, a finite function represented as a set, maps box variables to values. The reader should observe right here and now that the store θ is *local* to a state. This detail will turn out to be essential, when parallelism is added to the machine. The letref construct accepts mutually referent boxes, which correspond to circular data-structures in a real memory.

The semantics is defined by a *total* evaluation function $eval_{cs}$, which associates terminating programs with *Answer*s and non-terminating ones with $\perp$. An *Answer* is a closed value where λ-abstractions, boxes and continuations are replaced by tags. The first four rules of Figure 3 deal with the purely functional subset of the language in a traditional way. When the functional constant callcc, which stands for Scheme `call-with-current-continuation`, is applied on a value, rule (*capture*) creates a first-class continuation represented as an abortive abstraction $\lambda v.\mathcal{A}\mathcal{E}[v]$, with $\mathcal{E}$ the current evaluation context. A term $\mathcal{A}\ M$, called an abort-application [4], is meant to terminate the computation and to return the value of M; its behaviour is modelled by (*abort*) which discards its evaluation context. According to rule (*makeref*), the effect of applying makeref on a value V is to extend the local store with an association between a new box variable and the value V. This box can be accessed by deref and modified by setref!. As we wish to define a total evaluation function, error situations are detected, for instance in (β_v) when a non-applicable value occurs in operator position. Error situations are reported by using the abort operator $\mathcal{A}$, which will end the computation with the distinguished constant error. Rule (*future id*) [7] guarantees that future is an annotation by requiring the value of future to be the value of its body.

We shall consider the CS-machine as the sequential semantics of the language Λ_f. Its correctness is established by the fact that its reduction rules are derived from the λ_v-CS-calculi [4, 6, 23]. The CS-machine defines future as a *transparent* operator because the value of future is the value of its argument. Hence, annotated programs have the same observable behaviour as their non-annotated counterparts.

3 The $P(CS)$-Machine

In the CS-machine, future is regarded as a transparent annotation, but the real purpose of future is to indicate that an expression may be evaluated in parallel. This section presents the $P(CS)$-machine in which future is a construct that can create parallelism. Figure 4 displays the state space of the $P(CS)$-machine. We introduce a new kind of value, called *placeholder variable*, which "represents the result of a computation that is in progress [7]". We also distinguish *proper values* from runtime values: the former are like the runtime values of the CS-machine, while the latter include placeholder variables as well.

Whenever the $P(CS)$-machine interprets future as a construct for parallelism, it creates a new state with its own local store. Each state of the $P(CS)$-machine can be viewed as a task that may be evaluated in parallel with other states. Newly created states appear inside a construct f-let of the form $(\mathsf{f\text{-}let}\ (p\ M)\ S)$. Like a let, f-let binds a placeholder variable p to the value of M in S; its intension is to model the potential evaluation of S in parallel with the state in which f-let occurs. The term M is called the *primary* term and S is the *secondary* state. The computations that S generates are *speculative* because they are not known to be needed before M returns a value; they may even be useless if M escapes by a continuation invocation. On the other hand, the evaluation of M is *mandatory* because it contributes to the value of the state in which f-let occurs.

The evaluator specification of the $P(CS)$-machine is displayed in Figure 5. The meaning of $S_1 \mapsto_{pcs}^{n,m} S_2$ is that n steps are required in the transition from S_1 to S_2, and among them $m \leq n$ steps are mandatory.

As indicated by rule (*fork*), the primary term of a f-let construct is initially the future body, whose value will be bound to a new placeholder variable p. The secondary state evaluates a term composed of the context of future, i.e. its continuation, to which the placeholder p was passed; the secondary state[1] has its *own local* store (initially empty). As far as the secondary state is concerned, the store θ of the mandatory term is regarded as a *remote* store.

According to rule (*speculative*), the $P(CS)$-machine has the potential to perform speculative computations: if a state

[1] The definition of the evaluation context $\mathcal{E}$ is explained by the fact that a f-let created by (*fork*) always appears in the body of a letref. As rule (*capture*) is the only rule to create abort-applications, and as evaluation is not allowed inside abort-applications, the argument of $\mathcal{A}$ is always of the form $\mathcal{E}[V]$.

$$
\begin{array}{lcll}
S \in State_{cs} & ::= & (\mathsf{letref}\ \theta\ M) & \text{(State)} \\
M \in \Lambda_{cs} & ::= & V & \text{(Term)} \\
 & | & (M\ M) & \\
 & | & (\mathsf{if}\ M\ M\ M) & \\
 & | & (\mathsf{future}\ M) & \\
 & | & (\mathcal{A}\ M) & \\
V \in Value_{cs} & ::= & c \mid x & \text{(Runtime Value)} \\
 & | & f_c \mid b & \\
 & | & (\lambda x.M) \mid (\mathsf{cons}\ V\ V) & \\
c \in Const & ::= & a \mid d \mid f & \text{(Constant)} \\
f_c \in PApp & ::= & (\mathsf{cons}\ V) & \text{(Partial Application)} \\
 & | & (\mathsf{setref!}\ V) & \\
g \in AValue & ::= & (\lambda x.M) & \text{(Applicable Value)} \\
 & | & f \mid f_c & \\
\mathcal{E} \in EvCon & ::= & [\,] & \text{(Evaluation Context)} \\
 & | & (V\ \mathcal{E}) & \\
 & | & (\mathcal{E}\ M) & \\
 & | & (\mathsf{if}\ \mathcal{E}\ M\ M) & \\
 & | & (\mathsf{future}\ \mathcal{E}) & \\
\theta \in Store_{pc} & ::= & (\ldots(b_i\ V_i)\ldots) & \text{(Store)} \\
A \in Answer & ::= & c & \text{(Answer)} \\
 & | & (\mathsf{cons}\ A\ A) & \\
 & | & \mathsf{procedure} & \\
 & | & \mathsf{box} & \\
b \in box\text{-}Vars & = & \{b_0, b_1, \ldots\} & \text{(Box Variable)} \\
d \in DConst & = & \{\mathsf{error}, \mathsf{void}\} & \text{(Distinguished Constant)}
\end{array}
$$

Unload function: $Unload_{cs} : Value_{cs} \to Answer$

$$
\begin{array}{rcl}
Unload_{cs}[c] & = & c \\
Unload_{cs}[(\mathsf{cons}\ V_1\ V_2)] & = & (\mathsf{cons}\ \ Unload_{cs}[V_1]\ \ Unload_{cs}[V_2]) \\
Unload_{cs}[\lambda x.M] & = & \textbf{procedure} \\
Unload_{cs}[b] & = & \textbf{box} \\
Unload_{cs}[f_c] & = & \textbf{procedure}
\end{array}
$$

Free variables (additional rule):

$$FV(b) = \{b\}$$

Set of closed terms:

$$X^0 = \{M \in X, FV(M) = \emptyset\}$$

Store update:

$\theta' = \theta[b := V]$ is defined if
$\exists\ (b\ V') \in \theta$ and $\theta' = (\theta \setminus \{(b\ V')\}) \cup \{(b\ V)\}$

Store domain:

$$DOM((\ldots(b_i\ V_i)\ldots)) = \{\ldots b_i \ldots\}$$

Figure 2: State space of the CS-machine

Transition rules: $\mapsto_{cs}$: $State_{cs} \to State_{cs}$

$$
\begin{array}{rcll}
(\mathsf{letref}\ \theta\ \mathcal{E}[(V_1\ V_2)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[M[x \leftarrow V_2]]) & \text{if } V_1 = (\lambda x.M) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V_1 \notin AValue \end{cases} & (\beta_v) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{car}\ V)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V_1]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2) \end{cases} & (car) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{cdr}\ V)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V_2]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2) \end{cases} & (cdr) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{if}\ V\ M_1\ M_2)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[M_1]) & \text{if } V \neq \mathsf{false} \\ (\mathsf{letref}\ \theta\ \mathcal{E}[M_2]) & \text{if } V = \mathsf{false} \end{cases} & (if) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{callcc}\ V)]) & \mapsto_{cs} & (\mathsf{letref}\ \theta\ \mathcal{E}[V\ (\lambda v.\mathcal{A}\ \mathcal{E}[v])]) & (capture) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ M)]) & \mapsto_{cs} & (\mathsf{letref}\ \theta\ M) & (abort) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{makeref}\ V)]) & \mapsto_{cs} & (\mathsf{letref}\ \theta'\ \mathcal{E}[b])\ \text{ with } b \notin DOM(\theta),\ \theta' = \theta \cup \{(b\ V)\} & (makeref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{deref}\ V)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V']) & \text{if } V = b,\ (b\ V') \in \theta \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq b \end{cases} & (deref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{setref!}\ V_1\ V_2)]) & \mapsto_{cs} & \begin{cases} (\mathsf{letref}\ \theta'\ \mathcal{E}[\mathsf{void}]) & \text{if } V_1 = b,\ b \in DOM(\theta),\ \theta' = \theta[b := V_2] \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V_1 \neq b \end{cases} & (setref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{future}\ V)]) & \mapsto_{cs} & (\mathsf{letref}\ \theta\ \mathcal{E}[V]) & (future\ id)
\end{array}
$$

Evaluator specification: $eval_{cs} : \Lambda_f^0 \to Answer \cup \{\bot\}$

$$
eval_{cs}(P) = \begin{cases} Unload_{cs}[V] & \text{if } (\mathsf{letref}\ ()\ P) \mapsto_{cs}^* (\mathsf{letref}\ \theta\ V) \\ \bot & \text{if } \forall i \in \mathbf{N}, \exists S_i \in State_{cs} \text{ such that} \\ & \quad (\mathsf{letref}\ ()\ P) = S_0 \text{ and } S_i \mapsto_{cs} S_{i+1} \end{cases}
$$

Figure 3: Evaluator specification of the CS-machine

S can be reduced to a state S', these reductions can also be performed if S appears inside the body of a f-let, without waiting for the value of the placeholder. The only rules that introduce speculative computations are (*fork*) and (*speculative*), as indicated by the explicit zero for the number of mandatory transitions.

We can see that future, interpreted as a task-creation mechanism, offers a possible concurrent evaluation between a mandatory term *computing* the value of a placeholder, and a speculative state *using* this value; this clearly refers to a producer-consumer type of parallelism. In the consumer, *strict* operations, i.e. the operations that need the actual value being computed by the producer, introduce synchronisations. Strict primitives, like car, cdr, deref, and setref! require their arguments to be pairs or boxes. Similarly, "strict positions" like the operator position of an application or the predicate position of a conditional demand values different from placeholders. For this reason, all strict operations verify that their argument is not a placeholder as indicated in rules (β_v), (*car*), (*cdr*), (*if*), (*deref*), (*setref*).

If a primary term has produced a value V, rule (*join*) substitutes V for the free occurrences of the placeholder variable in the secondary state. In order to preserve the states organisation, the stores of the speculative and mandatory states are merged together, after having substituted V for the free occurrences of p across the whole speculative store. In the perspective of a distributed implementation, this rule may be interpreted as follows: mutable objects allocated by a mandatory task in its local store can be remotely accessed by a speculative task, only when the mandatory task has completed its execution.

By the non-determinism which stems from rules (*fork*) and (*speculative*), the evaluator can elect to perform speculative transitions instead of mandatory ones. As speculative computations may be infinite, while mandatory ones remain finite, divergence should be defined with the greatest care. In the following example,

$$(\mathsf{callcc}\ \lambda k.\ \ ((\mathsf{future}\ (k\ 1))\ \ \Omega))$$

the final result is 1, but an unbounded number of speculative transitions can be performed to evaluate the diverging term Ω. Therefore, we say that the evaluator diverges for a program P, if P leads to an infinite transition sequence that includes mandatory computations regularly often. Even though the evaluation order can be non-deterministic, $eval_{pcs}$ defines a total function. Based on a modified Diamond technique [7, 18], the following theorem states that the observable behaviours of programs are the same in the CS- and $P(CS)$- machines.

Theorem 1 $eval_{cs} = eval_{pcs}$ □

As a corollary of Theorem 1, future interpreted as a task-creation operator is a transparent annotation, i.e. future does not change the final answer of programs with effects despite parallel evaluation.

Two different techniques are used to ensure that future remains an annotation in the presence of first-class continuations and side-effects. On the one hand, rule (*capture*) packages up the whole continuation $\mathcal{E}$ into a first-class abortive abstraction which computes a terminal value when applied. The value will be considered as a *final answer* if it is produced by a mandatory state. On the other hand, consistency of side-effects is enforced by prohibiting a speculative state from accessing the local store of a mandatory one; in other words, access to a remote store requires synchronisations. This semantics ensures that parallelism can exist between states that perform effects locally. Practically, it means that two programs written by two programmers can run in parallel if each program uses its own boxes locally. This is precisely the programming style offered by mostly-functional languages like Scheme.

Our semantics does not enforce the scheduling strategy, except that it demands to perform mandatory transitions regularly often. The rule (*fork*) models task creation in the machine, but it is the implementor's responsibility to choose the characteristics of the scheduling strategy that suit best his goal, like for instance *eager* or *lazy* task creation [14, 3], *data-centric* or *task-centric* task allocation [21].

4 The F-PCKS-Machine

In MultiLisp [8], an expression (future M) also offers a potential concurrent evaluation between the task *computing* the value of M and the task *using* it. In order to synchronise these producer and consumer tasks, MultiLisp uses *placeholder* data-structures that can be assigned *at most one value*. The process that obtains the value of M stores this value in the placeholder. Assigning a value to a placeholder is usually referred to as *determining* the placeholder to the value. In the presence of first-class continuations, the expression M could "return multiple values", i.e. the continuation of M could be passed several values, successively. In order to preserve the "determine once" paradigm of placeholders, the first value passed to the continuation of M is stored in the placeholder, while the subsequent ones re-evaluate the continuation, as if future had not existed [12].

On the contrary, the $P(CS)$-machine implements placeholders as variables. By definition of substitution, variables receive at most one value, but rule (*capture*) can duplicate their binders. As a result, the $P(CS)$-machine is unsatisfactory in a number of respects:

1. In the state $S_1 = (\mathsf{letref}\ \theta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}[\mathsf{callcc}\ V])\ S))$, the continuation to be captured is of the form

   $$\lambda v.\mathcal{A}(\mathsf{f\text{-}let}\ (p\ \mathcal{E}[v])\ S).$$

   Each invocation of this continuation reinstates the f-let construct, and possibly gives each instance of p a new value.

2. In the same state S_1, if speculative computations $S \to S'$ are performed *after* the continuation is captured, the invocation of the continuation will restore the state S and not S'. More generally, rule (*abort*) erases any speculative computation that was performed in the context of $\mathcal{A}$.

3. Even though rule (*capture*) and the rules that implement the invocation of continuations (β_v followed by

$$
\begin{array}{lcll}
S \in State_{pcs} & ::= & (\mathsf{letref}\ \theta\ M) & \text{(State)} \\
 & | & (\mathsf{letref}\ \theta\ (\mathsf{f\text{-}let}\ (p\ M)\ S)) & \\
M \in \Lambda_{pcs} & ::= & V & \text{(Term)} \\
 & | & (M\ M) & \\
 & | & (\mathsf{if}\ M\ M\ M) & \\
 & | & (\mathsf{future}\ M) & \\
 & | & (\mathcal{A}\ \mathcal{E}[V]) & \\
W \in PValue_{pcs} & ::= & c \mid x \mid f_c \mid b & \text{(Proper Value)} \\
 & | & (\lambda x.M) \mid (\mathsf{cons}\ V\ V) & \\
V \in Value_{pcs} & ::= & W \mid p & \text{(Runtime Value)} \\
\mathcal{D} \in SEvCon & ::= & [\,] & \text{(Sequential} \\
 & | & (V\ \mathcal{D}) & \text{Evaluation Context)} \\
 & | & (\mathcal{D}\ M) & \\
 & | & (\mathsf{if}\ \mathcal{D}\ M\ M) & \\
 & | & (\mathsf{future}\ \mathcal{D}) & \\
\mathcal{E} \in EvCon_{pcs} & ::= & \mathcal{D} & \text{(Evaluation Context)} \\
 & | & (\mathsf{f\text{-}let}\ (p\ \mathcal{D})\ S) & \\
p \in p\text{-}Vars & = & \{p_0, p_1, \ldots\} & \text{(Placeholder Variable)}
\end{array}
$$

Unload function:

$$
\begin{array}{rcl}
Unload_{pcs} & : & PValue_{pcs} \to Answer \\
Unload_{pcs}[W] & = & Unload_{cs}[W]
\end{array}
$$

Substitution for p variable:

$$
\begin{array}{l}
p[p \leftarrow V] = V \\
p[p' \leftarrow V] = p \text{ if } p \neq p' \\
(\mathsf{f\text{-}let}\ (p\ M)\ S)[p' \leftarrow V] \\
\qquad = (\mathsf{f\text{-}let}\ (p\ M[p' \leftarrow V])S) \text{ if } p = p' \\
(\mathsf{f\text{-}let}\ (p\ M)\ S)[p' \leftarrow V] \\
\qquad = (\mathsf{f\text{-}let}\ (p\ M[p' \leftarrow V])S[p' \leftarrow V]) \text{ if } p \neq p'
\end{array}
$$

Free placeholder:

$$
\begin{array}{l}
FP(p) = \{p\} \\
FP((\mathsf{f\text{-}let}\ (p\ M)\ S)) \\
\qquad = FP(M) \cup (FP(S) \setminus \{p\})
\end{array}
$$

Figure 4: State space of the $P(CS)$-machine (Differences with Figure 2)

Transition Rules:

$$
\begin{array}{rcll}
(\mathsf{letref}\ \theta\ \mathcal{E}[(V_1\ V_2)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[M[x \leftarrow V_2]]) & \text{if } V_1 = (\lambda x.M) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V_1 \notin AValue, V_1 \neq p \end{cases} & (\beta_v) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{car}\ V)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V_1]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2), V \neq p \end{cases} & (car) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{cdr}\ V)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V_2]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2), V \neq p \end{cases} & (cdr) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{if}\ V\ M_1\ M_2)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[M_1]) & \text{if } V \neq \mathsf{false}, V \neq p \\ (\mathsf{letref}\ \theta\ \mathcal{E}[M_2]) & \text{if } V = \mathsf{false} \end{cases} & (if) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{callcc}\ V)]) & \mapsto_{pcs}^{1,1} & (\mathsf{letref}\ \theta\ \mathcal{E}[V\ (\lambda v.\mathcal{A}\mathcal{E}[v])]) & (capture) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathcal{E}_1[V])]) & \mapsto_{pcs}^{1,1} & (\mathsf{letref}\ \theta\ \mathcal{E}_1[V]) & (abort) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{makeref}\ V)]) & \mapsto_{pcs}^{1,1} & (\mathsf{letref}\ \theta'\ \mathcal{E}[b]) \text{ with } b \notin DOM(\theta),\ \theta' = \theta \cup \{(b\ V)\} & (makeref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{deref}\ V)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta\ \mathcal{E}[V']) & \text{if } V = b, (b\ V') \in \theta \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq b, V \neq p \end{cases} & (deref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{setref!}\ V_1\ V_2)]) & \mapsto_{pcs}^{1,1} & \begin{cases} (\mathsf{letref}\ \theta'\ \mathcal{E}[\mathsf{void}]) & \text{if } V_1 = b, b \in DOM(\theta),\ \theta' = \theta[b := V_2] \\ (\mathsf{letref}\ \theta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V_1 \neq b, V_1 \neq p \end{cases} & (setref) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{future}\ V)]) & \mapsto_{pcs}^{1,1} & (\mathsf{letref}\ \theta\ \mathcal{E}[V]) & (future\ id) \\
(\mathsf{letref}\ \theta\ \mathcal{E}[(\mathsf{future}\ M)]) & \mapsto_{pcs}^{1,0} & (\mathsf{letref}\ \theta\ (\mathsf{f\text{-}let}\ (p\ M)\ (\mathsf{letref}\ ()\ \mathcal{E}[p]))) \text{ with } p \notin FP(\mathcal{E}) & (fork) \\
(\mathsf{letref}\ \theta\ (\mathsf{f\text{-}let}\ (p\ M)\ S)) & \mapsto_{pcs}^{a,0} & (\mathsf{letref}\ \theta\ (\mathsf{f\text{-}let}\ (p\ M)\ S')) \text{ if } S \mapsto_{pcs}^{a,b} S' & (speculative) \\
(\mathsf{letref}\ \theta_1\ (\mathsf{f\text{-}let}\ (p\ V)\ (\mathsf{letref}\ \theta_2\ \mathcal{E}[M]))) & \mapsto_{pcs}^{1,1} & (\mathsf{letref}\ \theta_3\ \mathcal{E}[M]\ [p \leftarrow V]) \text{ with } DOM(\theta_1) \cap DOM(\theta_2) = \emptyset, & (join) \\
 & & \quad \theta_3 = \theta_1 \cup \theta_2', \text{ and } \theta_2' = \{(b\ V_2') \mid \exists\ (b\ V_2) \in \theta_2,\ V_2' = V_2[p \leftarrow V]\} & \\
S & \mapsto_{pcs}^{0,0} & S & (reflexive) \\
S & \mapsto_{pcs}^{a+a',b+b'} & S'' \text{ if } S \mapsto_{pcs}^{a,b} S' \text{ and } S' \mapsto_{pcs}^{a',b'} S''. & (transitive)
\end{array}
$$

Evaluator Specification: $eval_{pcs} : \Lambda_f^0 \to Answer \cup \{\bot\}$

$$
eval_{pcs}(P) = \begin{cases} Unload_{pcs}[W] & \text{if } (\mathsf{letref}\ ()\ P) \mapsto_{pcs}^{*} (\mathsf{letref}\ \theta\ W) \\ \bot & \text{if } \forall i \in \mathbf{N}, \exists S_i \in State_{pcs}, n_i, m_i \in \mathbf{N} \text{ such that} \\ & \quad (\mathsf{letref}\ ()\ P) = S_0 \text{ and } S_i \mapsto_{pcs}^{n_i,m_i} S_{i+1} \text{ with } m_i > 0. \end{cases}
$$

Figure 5: Evaluator specification of the $P(CS)$-machine

abort) are simple and faithful to the semantics of continuations [4, 6, 23], they are expensive and difficult to implement. Indeed, rule (*capture*) takes a snapshot of all speculative states running in parallel with the current state, whereas rule (*invoke*) restores this snapshot.

4. When a primary term provides a value V, rule (*join*) substitutes V for the free occurrences of the placeholder variable in the f-let body, which involves a substitution across the whole speculative store. This substitution fundamentally differs from the one in the β_v-reduction: the latter corresponds to an environment extension, while the former is comparable to a side-effect.

Besides the fact that the $P(CS)$-machine does not consider placeholders as mutable data-structures, there is another reason that explains its distinct behaviour with regard to MultiLisp implementations: context-rewriting machines do not distinguish the continuation of a future from its speculative evaluation, because both are represented by the same evaluation context. Consequently, rule (*capture*) has no other choice but to take snapshots of running states.

Taking these observations into account, we have designed a lower-level refinement of the $P(CS)$-machine, which incorporates MultiLisp-like solutions. The F-PCKS-machine abstracts a MIMD architecture with a shared memory in the tradition of MultiLisp systems [8]. The F-PCKS-machine generalises the CK and CKS machines [4, 5], by providing a parallel evaluation mechanism based on a notion of task. Each task is represented by a CK-configuration (composed of a control string and a continuation), and has access to a shared memory.

The state space of the F-PCKS-machine appears in Figure 6 and its evaluator specification in Figure 7. The set of values contains a new kind of object $\langle \mathsf{ph}\ \alpha \rangle$, called *placeholder*, which refers to a location α in the shared store. A placeholder is *undetermined* if its associated location is empty; it gets *determined* to a value V by storing V in its location. In the F-PCKS-machine, care is taken not to determine a placeholder more than once. As in the $P(CS)$-machine, a strict operation must ensure that its argument is not an undetermined placeholder. The action of obtaining the value of a placeholder, called *touching* the placeholder, is implemented by the function touch_σ. Let us notice that this function is recursive because placeholders can be determined by other placeholders.

Active tasks are quadruples formed of a term, a continuation code, a *legitimacy* [12], used to coordinate effects and validate final values, and a name. A legitimacy $\langle \mathsf{leg}\ \alpha \rangle$, like a placeholder[2], is a data-structure that refers to a location in the shared store, but unlike a placeholder, it is not considered as a value because there exists no primitive to reify it to the status of value. We shall also use the terms "undetermined" and "determined" for legitimacies. In the $P(CS)$-machine, f-let is the construct that distinguishes mandatory from speculative evaluations; legitimacy plays a similar role in the F-PCKS-machine.

When starting the execution of a program, the initial task is given the initial legitimacy ℓ_i. Whenever a task τ evaluates a future, rule (*fork*) creates a task τ', and allocates a new placeholder ph and a new legitimacy ℓ_1 (both initially undetermined). After transition (*fork*), task τ still has the same legitimacy, but is now evaluating the future body with a continuation $(\kappa\ \mathbf{det}(ph, \ell_1))$, where ph is the placeholder to determine and ℓ_1 the legitimacy of τ'. Meanwhile, the task τ' begins to evaluate the continuation of future with the new placeholder ph. We know that task τ' performs a speculative computation on behalf of τ because the legitimacy of τ' is ℓ_1, and the continuation of task τ contains a code $(\kappa\ \mathbf{det}\ (ph, \ell_1))$, where ℓ_1 is explicit.

Rule (*determine*) has two roles. First, it ensures that a placeholder is given at most one value: regardless of its legitimacy, the first task producing a value for a given placeholder has the right to determine the placeholder to that value, while the other ones continue the evaluation as if no future had existed. Second, it keeps track of legitimacy as follows. If the placeholder $\langle \mathsf{ph}\ \alpha \rangle$ gets determined to a value V, the task that consumes this placeholder speculatively becomes dependent on the value V. This dependency is made explicit by giving the consuming task the legitimacy of the producing task. More precisely, the legitimacy of the consumer task is determined to the legitimacy of the producer task. So, legitimacy is passed between tasks as a token, whenever a placeholder gets determined.

When a legitimacy $\langle \mathsf{leg}\ \alpha \rangle$ gets determined, location α receives a legitimacy, which might also be determined. Hence, as evaluation proceeds, chains of legitimacies get formed in memory. We define a relation $\ell_1 \rightsquigarrow_\sigma \ell_2$ stating that there is a path from legitimacy ℓ_1 to legitimacy ℓ_2, which means that control has flowed from a task with legitimacy ℓ_2 to a task with legitimacy ℓ_1. Intuitively, legitimacy models the fact that a sequential implementation would have performed the evaluation done by the tasks with legitimacies ℓ_2 to ℓ_1. The relation $\rightsquigarrow$ is used to determine whether a final value, i.e. a value returned to the (**init**) continuation, is a valid answer. A valid answer is produced by the task whose legitimacy ℓ is such that $\ell \rightsquigarrow_\sigma \ell_i$. The initial legitimacy ℓ_i is a pre-allocated legitimacy which serves as a marker for the end of legitimacy chains. This legitimacy always remains undetermined because the initial program does not depend on any placeholder.

A first-class continuation $\langle \mathsf{co}\ \kappa \rangle$ is a pair composed of tag co and a continuation code κ. A box $\langle \mathsf{bx}\ \alpha, \ell \rangle$ is a triple composed of a tag bx, a location, and the legitimacy of the task which performed the transition (*makeref*). As legitimacy represents the sequential flow of control, it is used to coordinate side-effects soundly. Read and write operations on boxes are allowed if they are performed in the same order as a sequential implementation would have done them; i.e., the task that wishes to access a box should have a legitimacy determined (possibly through a chain) to the legitimacy of the task that created it. Said differently, a task is not allowed to access a box if it is more speculative than the one that created it, as in the $P(CS)$-machine.

[2] Katz, Weise, and Feeley [12, 3] suggest to implement legitimacy by placeholders. However, as placeholders and legitimacy have different semantic purposes, we decided to give them different representations.

$$
\begin{array}{lcll}
S \in State_{fpcks} & ::= & \langle T, \sigma \rangle & \text{(State)} \\
t \in T & ::= & \langle M, \kappa, \ell \rangle_\tau \mid \langle \ddagger, (\textbf{stop}), \ddagger \rangle_\tau & \text{(Task)} \\
M \in \Lambda_{fpcks} & ::= & V \mid (M\ M) & \text{(Term)} \\
 & \mid & (\mathsf{if}\ M\ M\ M) & \\
 & \mid & (\mathsf{future}\ M) & \\
W \in PValue_{fpcks} & ::= & c \mid x \mid (\lambda x.M) & \text{(Proper Value)} \\
 & \mid & \langle \mathsf{co}\ \kappa \rangle \mid \langle \mathsf{bx}\ \alpha, \ell \rangle & \\
 & \mid & (\mathsf{cons}\ V\ V) \mid f_c & \\
V \in Value_{fpcks} & ::= & W \mid ph & \text{(Runtime Value)} \\
\kappa \in CCode & ::= & (\textbf{init}) & \text{(Continuation code)} \\
 & \mid & (\kappa\ \textbf{fun}\ V) & \\
 & \mid & (\kappa\ \textbf{arg}\ M) & \\
 & \mid & (\kappa\ \textbf{cond}(M, M)) & \\
 & \mid & (\kappa\ \textbf{det}\ (ph, \ell)) & \\
ph \in Placeholder & ::= & \langle \mathsf{ph}\ \alpha \rangle & \text{(Placeholder)} \\
\ell \in Legitimacy & ::= & \langle \mathsf{leg}\ \alpha \rangle & \text{(Legitimacy)} \\
\sigma \in Store_{fpcks} & : & Loc \to Contents & \text{(Store)} \\
O \in Contents & ::= & V \mid \ell \mid \bot & \text{(Store Content)} \\
\alpha \in Loc & = & \{\alpha_0, \alpha_1, \ldots\} & \text{(Location)} \\
\tau \in Tid & = & \{\tau_0, \tau_1, \ldots\} & \text{(Task Identifier)}
\end{array}
$$

Mandatory descendant:

$$
\ell_0 \rightsquigarrow_\sigma \ell_1 \quad \text{if} \quad \begin{cases} \ell_0 = \ell_1, & \text{or} \\ \sigma(\alpha_0) \rightsquigarrow_\sigma \ell_1 & \text{if } \ell_0 = \langle \mathsf{leg}\ \alpha_0 \rangle \text{ and } \sigma(\alpha_0) \neq \bot \end{cases}
$$

Touch function:

$$
\begin{array}{rcl}
\text{touch} & : & Value_{fpcks} \times Store_{fpcks} \\
 & & \quad \to PValue_{fpcks} \cup \{\bot\} \\
\text{touch}_\sigma(V) & = & V \text{ if } V \neq ph \\
\text{touch}_\sigma(\langle \mathsf{ph}\ \alpha \rangle) & = & \text{touch}_\sigma(\sigma(\alpha)) \text{ if } \sigma(\alpha) \neq \bot \\
\text{touch}_\sigma(\langle \mathsf{ph}\ \alpha \rangle) & = & \bot \text{ if } \sigma(\alpha) = \bot
\end{array}
$$

Initial Legitimacy: $\ell_i = \langle \mathsf{leg}\ \alpha_i \rangle$
Initial Store: $\sigma_i = \lambda\alpha.\bot$

Unload function (differences):

$$
\begin{array}{rcl}
Unload_{fpcks}[\langle \mathsf{bx}\ \alpha, \ell \rangle] & = & \textbf{box} \\
Unload_{fpcks}[\langle \mathsf{co}\ \kappa \rangle] & = & \textbf{procedure}
\end{array}
$$

Figure 6: State space of the F-PCKS-machine

The correctness of the third semantics comes from Theorem 2 which states that the evaluators of the F-PCKS- and $P(CS)$- machines define identical functions. The proof of Theorem 2 [18] involves a translation of any F-PCKS-state into the $P(CS)$ state space. A Lemma establishes that any reachable state of the F-PCKS-machine corresponds to a state that the $P(CS)$-machine can reach.

Theorem 2 $eval_{fpcks} = eval_{pcs}$ □

The F-PCKS-machine is a formalisation of Katz and Weise's implementation schema [12]. If we consider the functional subset with first-class continuations, it avoids synchronisations and resorts to speculative computations as much as possible. It is well-known that speculative computations may lead to unintended computations, and a programmer might wish to have more control on their generation. There exist two opposite views concerning this problem: we can restrict either futures or continuations. Both approaches can be easily described from the current semantic framework. Figure 8 displays two proposed variants; in contrast, the semantics of Figures 6 and 7 will be referred to as "the *unrestrictive* semantics".

The essence of future is to initiate a computation that speculatively uses a placeholder while its actual value is being computed. The "future-*restrictive*" solution authorises a task τ to determine a placeholder allocated when evaluating an expression (future M), only if τ is the mandatory task that evaluated M. This approach restricts speculation because it forbids a task speculatively spawned by e to take advantage of the speculative computation performed with the placeholder. We implement this solution by adding a legitimacy to placeholders, and by modifying rules (*fork*) and (*determine*). Every newly allocated placeholder receives the legitimacy of the task that executes (*fork*). Rule (*determine*) is added a new side-condition so that placeholders can now be determined only when the legitimacy chain guarantees that the current task is not more speculative that the one that created the placeholder. Though placeholders look very similar to boxes, they remain different from them, because they can be mutated at most once, and because reading their content is not conditioned by the speculativeness of the task.

The "*continuation-restrictive*" solution, proposed in [16, 18], allows a task τ to invoke a continuation, if τ is not more speculative than the task that created the first-class continuation. As far as the implementation is concerned, a legitimacy is added to each first-class continuation; the legitimacy of a continuation is the one of the task that executes (*capture*). Similarly, rule (*invoke*) can be fired according to the legitimacy added to continuation points.

The "continuation-restrictive" solution bears a strong resemblance to the technique that coordinates side-effects in the F-PCKS-machine. Parallelism is still allowed between modules that do not share continuations or boxes, which is a reasonable assumption for mostly-functional languages like Scheme. The "future-restrictive" solution is more permissive because it still allows speculative invocations of continuations. The following example illustrates how it differs from the unrestricted semantics of the F-PCKS-machine:

$$
\begin{array}{l}
\mathcal{E}_1[\mathsf{future}\ \ (\mathsf{callcc}\ \lambda k. \\
\qquad\qquad (\mathsf{cons}\ (\mathsf{future}(k\ 1))\ (\mathsf{future}(k\ 2))))]
\end{array}
$$

In the unrestricted semantics, the outermost future allocates a placeholder ph_1, and creates a task to evaluate $\mathcal{E}_1[ph_1]$ speculatively. The innermost futures allocate two placeholders ph_2, ph_3, and create two additional tasks, so that $(k\ 1)$,

Transition rules: $\mapsto_{cks}$: $(Task \times Store_{fpcks}) \to (Tasks^* \times Store_{fpcs})$

$$\langle (M\ N), \kappa, \ell\rangle_\tau \to_{cks} \langle M, (\kappa\ \mathbf{arg}\ N), \ell\rangle_\tau \quad \text{if } (M\ N) \notin PApp \qquad (\textit{operator})$$

$$\langle V, (\kappa\ \mathbf{arg}\ N), \ell\rangle_\tau \to_{cks} \langle N, (\kappa\ \mathbf{fun}\ V), \ell\rangle_\tau \qquad (\textit{operand})$$

$$\langle V, (\kappa\ \mathbf{fun}\ \lambda x.M), \ell\rangle_\tau \to_{cks} \langle M[x \leftarrow V], \kappa, \ell\rangle_\tau \qquad (\beta_v)$$

$$\langle V, (\kappa\ \mathbf{fun}\ V_1), \ell\rangle_\tau \to_{cks} \langle (V_1\ V), \kappa, \ell\rangle_\tau \quad \text{if } (V_1\ V) \in PApp \qquad (\textit{partial apply})$$

$$\langle V, (\kappa\ \mathbf{fun}\ (\mathsf{cons}\ V_1)), \ell\rangle_\tau \to_{cks} \langle (\mathsf{cons}\ V_1\ V), \kappa, \ell\rangle_\tau \qquad (\textit{cons})$$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{car}), \ell\rangle_\tau \to_{cks} \begin{cases} \langle V_1, \kappa, \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) = (\mathsf{cons}\ V_1\ V_2) \\ \langle \mathsf{error}, (\kappa\ \mathbf{fun}\ \langle \mathsf{co}\ (\mathbf{init})\rangle), \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) \neq (\mathsf{cons}\ V_1\ V_2),\ \mathrm{touch}_\sigma(V) \neq \bot \end{cases} \qquad (\textit{car})$$

$$\langle (\mathsf{if}\ M\ M'\ M''), \kappa, \ell\rangle_\tau \to_{cks} \langle M, (\kappa\ \mathbf{cond}\ (M', M'')), \ell\rangle_\tau \qquad (\textit{predicate})$$

$$\langle V, (\kappa\ \mathbf{cond}\ (M, M')), \ell\rangle_\tau \to_{cks} \begin{cases} \langle M', \kappa, \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) = \mathsf{false} \\ \langle M, \kappa, \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) \neq \mathsf{false},\ \mathrm{touch}_\sigma(V) \neq \bot \end{cases} \qquad (\textit{if})$$

$$\langle \mathsf{future}\ M, \kappa, \ell\rangle_\tau \to_{cks} \{\langle M, (\kappa\ \mathbf{det}\ (ph, \ell_1)), \ell\rangle_\tau, \langle ph, \kappa, \ell_1\rangle_{\tau'}\}; \sigma[\alpha_1 \leftarrow \bot][\alpha \leftarrow \bot] \qquad (\textit{fork})$$
with $\ell_1 = \langle \mathsf{leg}\ \alpha_1\rangle, ph = \langle \mathsf{ph}\ \alpha\rangle$, a fresh $\alpha_1 \in Loc$, a fresh $\alpha \in Loc$, a new $\tau' \in Tid$

$$\langle V, (\kappa\ \mathbf{fun}\ \langle \mathsf{ph}\ \alpha\rangle), \ell\rangle_\tau \to_{cks} \begin{cases} \langle V, (\kappa\ \mathbf{fun}\ V_1), \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(\langle \mathsf{ph}\ \alpha\rangle) = V_1,\ V_1 \in AValue \\ \langle \mathsf{error}, (\kappa\ \mathbf{fun}\ \langle \mathsf{co}\ (\mathbf{init})\rangle), \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(\langle \mathsf{ph}\ \alpha\rangle) = V_1,\ V_1 \notin AValue,\ V_1 \neq \bot \end{cases} \qquad (\textit{app touch})$$

$$\langle V, (\kappa\ \mathbf{det}\ (\langle \mathsf{ph}\ \alpha\rangle, \langle \mathsf{leg}\ \alpha_1\rangle)), \ell_2\rangle_\tau \to_{cks} \begin{cases} \langle \ddagger, (\mathbf{stop}), \ddagger\rangle_\tau; \sigma[\alpha_1 \leftarrow \ell_2][\alpha \leftarrow V] & \text{if } \sigma(\alpha) = \bot \\ \langle V, \kappa, \ell_2\rangle_\tau & \text{if } \sigma(\alpha) \neq \bot \end{cases} \qquad (\textit{determine})$$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{callcc}), \ell\rangle_\tau \to_{cks} \langle \langle \mathsf{co}\ \kappa\rangle, (\kappa\ \mathbf{fun}\ V), \ell\rangle_\tau \qquad (\textit{capture})$$

$$\langle V, (\kappa'\ \mathbf{fun}\ \langle \mathsf{co}\ \kappa\rangle), \ell\rangle_\tau \to_{cks} \langle V, \kappa, \ell\rangle_\tau \qquad (\textit{invoke})$$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{makeref}), \ell\rangle_\tau \to_{cks} \langle \langle \mathsf{bx}\ \alpha, \ell\rangle, \kappa, \ell\rangle_\tau; \sigma[\alpha \leftarrow V] \qquad (\textit{makeref})$$
with a fresh $\alpha \in Loc$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{deref}), \ell\rangle_\tau \to_{cks} \begin{cases} \langle \sigma(\alpha), \kappa, \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) = \langle \mathsf{bx}\ \alpha, \ell_1\rangle,\ \ell \rightsquigarrow_\sigma \ell_1 \\ \langle \mathsf{error}, (\kappa\ \mathbf{fun}\ \langle \mathsf{co}\ (\mathbf{init})\rangle), \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V) \neq \langle \mathsf{bx}\ \alpha, \ell_1\rangle,\ \mathrm{touch}_\sigma(V) \neq \bot \end{cases} \qquad (\textit{deref})$$

$$\langle V, (\kappa'\ \mathbf{fun}\ (\mathsf{setref!}\ V_1)), \ell\rangle_\tau \to_{cks} \begin{cases} \langle \mathsf{void}, \kappa', \ell\rangle_\tau; \sigma[\alpha \leftarrow V] & \text{if } \mathrm{touch}_\sigma(V_1) = \langle \mathsf{bx}\ \alpha, \ell_1\rangle,\ \ell \rightsquigarrow_\sigma \ell_1 \\ \langle \mathsf{error}, (\kappa'\ \mathbf{fun}\ \langle \mathsf{co}\ (\mathbf{init})\rangle), \ell\rangle_\tau & \text{if } \mathrm{touch}_\sigma(V_1) \neq \langle \mathsf{bx}\ \alpha, \ell_1\rangle,\ \mathrm{touch}_\sigma(V_1) \neq \bot \end{cases} \qquad (\textit{setref})$$

Conventions:

$$\begin{array}{rll} \langle t, \sigma\rangle \mapsto_{cks} \langle \{t'\}, \sigma\rangle & \text{is written} & t \mapsto_{cks} t' \\ \langle t, \sigma\rangle \mapsto_{cks} \langle \{t'\}, \sigma'\rangle & \text{is written} & t \mapsto_{cks} t'; \sigma' \leftarrow \sigma \\ \langle t, \sigma\rangle \mapsto_{cks} \langle \{t', t''\}, \sigma'\rangle & \text{is written} & t \mapsto_{cks} \{t', t''\}; \sigma' \leftarrow \sigma \end{array}$$

Transition relation $\to_{fpcks}$: $State_{fpcks} \to State_{fpcks}$

$$\langle T, \sigma\rangle \to^{1,a}_{fpcks} \langle T', \sigma'\rangle \text{ if } \begin{array}{l} \exists t = \langle M, \kappa, \ell\rangle_\tau \in T, \text{ such that } \langle t, \sigma\rangle \mapsto_{cks} \langle T'', \sigma'\rangle, \\ T' = (T \setminus \{t\}) \cup \{T''\}, \\ a = 1 \text{ if } \ell \rightsquigarrow_\sigma \ell_i, \text{ and } a = 0 \text{ otherwise.} \end{array} \qquad (\textit{base})$$

$$S \to^{a+a', b+b'}_{fpcks} S'' \text{ if } S \to^{a,b}_{fpcks} S' \text{ and } S' \to^{a',b'}_{fpcks} S''. \qquad (\textit{transitive})$$

Evaluator Specification: $eval_{fpcks} : \Lambda^0_f \to Answer \cup \{\bot\}$

$$eval_{fpcks}(P) = \begin{cases} Unload_{fpcks}[\mathrm{touch}_\sigma(V)] & \text{if } \langle\ \{\ \langle P, (\mathbf{init}), \ell_i\rangle_{\tau_0}\ \}\ , \sigma_i\rangle \to^*_{fpcks} \langle T, \sigma\rangle, \\ & \quad \text{with } \langle V, (\mathbf{init}), \ell\rangle_\tau \in T, \text{ such that } \ell \rightsquigarrow_\sigma \ell_i \\ \bot & \text{if } \forall i \in \mathbf{N}, \exists S_i \in State_{fpcks}, n_i, m_i \in \mathbf{N}, \text{ such that} \\ & \quad S_0 = \langle\ \{\ \langle P, (\mathbf{init}), \ell_i\rangle_{\tau_0}\ \}\ , \sigma_i\rangle, S_i \to^{n_i, m_i}_{fpcks} S_{i+1}, m_i > 0 \end{cases}$$

Figure 7: Evaluator specification of the F-PCKS-machine

future-restrictive solution:

$$ph \in Placeholder \quad ::= \quad \langle \mathsf{ph}\ \alpha, \ell \rangle \quad \text{(Placeholder)}$$

$$\langle \mathsf{future}\ M, \kappa, \ell \rangle_\tau \ \mapsto_{cks} \ \{\ \langle M, (\kappa\ \mathbf{det}\ (ph, \ell_1)), \ell \rangle_\tau,\ \ \langle ph, \kappa, \ell_1 \rangle_{\tau'}\ \}; \sigma[\alpha_1 \leftarrow \bot][\alpha \leftarrow \bot] \quad (\textit{fork})$$
with $\ell_1 = \langle \mathsf{leg}\ \alpha_1 \rangle, ph = \langle \mathsf{ph}\ \alpha, \ell \rangle$, a fresh $\alpha_1 \in Loc$,
a fresh $\alpha \in Loc$, a new $\tau' \in Tid$

$$\langle V, (\kappa\ \mathbf{det}\ (\langle \mathsf{ph}\ \alpha, \ell \rangle, \langle \mathsf{leg}\ \alpha_1 \rangle)), \ell_2 \rangle_\tau \ \mapsto_{cks} \ \begin{cases} \langle \ddagger, (\mathbf{stop}), \ddagger \rangle_\tau; \sigma[\alpha_1 \leftarrow \ell_2][\alpha \leftarrow V] & \text{if } \sigma(\alpha) = \bot, \ell_2 \rightsquigarrow_\sigma \ell \\ \langle V, \kappa, \ell_2 \rangle_\tau & \text{if } \sigma(\alpha) \neq \bot \end{cases} \quad (\textit{determine})$$

continuation-restrictive solution:

$$W \in PValue_{fpcks} \quad ::= \quad \ldots \mid \langle \mathsf{co}\ \kappa, \ell \rangle \mid \ldots \quad \text{(Proper Value)}$$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{callcc}), \ell \rangle_\tau \ \mapsto_{cks} \ \langle \langle \mathsf{co}\ \kappa, \ell \rangle, (\kappa\ \mathbf{fun}\ V), \ell \rangle_\tau \quad (\textit{capture})$$

$$\langle V, (\kappa'\ \mathbf{fun}\ \langle \mathsf{co}\ \kappa, \ell_1 \rangle), \ell \rangle_\tau \ \mapsto_{cks} \ \langle V, \kappa, \ell \rangle_\tau \ \text{ if } \ \ell \rightsquigarrow_\sigma \ell_1 \quad (\textit{invoke})$$

Figure 8: Speculation controlling variants

$(k\ 2)$, and $(\mathsf{cons}\ ph_2\ ph_3)$ are all returning values for the outermost future body, in parallel. Regardless of its legitimacy, the first task that returns a value (1, 2, or $(\mathsf{cons}\ p_2\ p_3)$) stores it in ph_1, and dies; the following ones keep on evaluating the context $\mathcal{E}_1$.

On the contrary, the future-restricted semantics allows the placeholder ph_1 to be determined by 1 only; the other results, which are more speculative than 1, requires re-evaluating the context $\mathcal{E}_1$. This implementation of future favours programs using continuations in a downward way, by avoiding the speculative computation performed on $\mathcal{E}_1[ph_1]$ being "stolen" by a non-legitimate result (2 or $(\mathsf{cons}\ p_2\ p_3)$). This solution particularly makes sense when side-effects should be performed in $\mathcal{E}_1$. Indeed, as side-effects appearing in $\mathcal{E}_1$ can only be performed by the legitimate task, it is a reasonable assumption to only let the mandatory task take advantage of the speculative computation already performed.

5 Related Work and Discussion

Previously, the author [15, 16, 17, 19] studied the semantics of a functional language extended with first-class continuations, side-effects, and pcall annotations. The annotation pcall indicates that subexpressions of an application may be evaluated in parallel before applying the value of the operator on the values of the arguments. Parallelism generated by pcall is said to be of the type "fork and join", while the one generated by future, more speculative, is of the type "producer consumer". The following translation suffices to define pcall in terms of future:

$$\overline{(\mathsf{pcall}\ M\ N)} = ((\mathsf{future}\ M)\ N)$$

In [17, 18], the semantics proposed for pcall-based programs relied on the PCKS-machine. The F-PCKS-machine generalises the PCKS-machine by the language accepted and simplifies the architecture.

According to the nomenclature proposed in this paper, the semantics of pcall [15, 16, 17, 18] can be categorised as "continuation-restrictive". This means that a notation pcall based on the unrestricted future would be more speculative than in [15, 16, 17, 18].

The only other work concerning the formal semantics of annotation-based parallelism is Flanagan and Felleisen's [7] semantics of future in a purely functional language. They proposed several abstract machines with varying degrees of intensionality. Our CS- and $P(CS)$- machines extend their C- and $P(C)$- machines with side-effects and first-class continuations. Interestingly, no extra constraint is added to the functional core in order to support effects.

Our F-PCKS-machine is a lower-level refinement of their C_{ph}-machine. Indeed, as continuations were part of the language, we considered that it was crucial to understand when placeholders were allocated and determined. To that end, we have provided the F-PCKS-machine with an explicit shared memory à la MultiLisp [8]. The F-PCKS-machine is still too abstract to be considered as a real implementation. We could easily get rid of the substitution model by introducing an environment [4]. Furthermore, the level of atomicity assumed by transitions is still too high; a solution based on explicit critical sections could be derived from [16].

In addition, in order to keep the semantics as simple as possible, we have not included queues in the representation of placeholders. As a result, a task which touches an undetermined placeholder is not allowed to fire a transition rule. In order to avoid busy-waiting, it is common practice [12, 3, 8] to suspend such a task, to put it in a queue associated with the placeholder, and to resume it as soon as the placeholder gets determined.

The collection of tasks is an issue that can have a serious impact on the practicality and performance of a parallel implementation. Halstead's MultiLisp [8] does not reclaim any runnable task. The collection of tasks in Miller's MultiScheme [13] is solved during garbage collection: indeed, a task can be collected if the placeholder that it determines is accessible from the gc roots.

From their semantics of future, Flanagan and Felleisen derive a set-based analysis [9] to perform a "touch optimisation", removing provably-unnecessary touch operations. There is no doubt that such an analysis would be useful for our language. However, we feel that a major difficulty is to design an analysis which provides safe and accurate approximations of expressions, when side-effects and continuations are used. Other analysis and associated optimisations are conceivable to improve evaluation. For instance, every access to a box (read or write) requires to check the legitimacy of the running task. A static analysis and a program transformation, in the spirit of the touch optimisation, could reduce the costs of theses checks; set-based analysis [9] or inference of regions and effects [24] would be two interesting approaches to design the analysis.

The annotation future had never been studied in the presence of both side-effects and first-class continuations. Katz and Weise [12, 3] described an implementation for future and continuations, and Tinker and Katz [25] designed Paratran to deal with futures and side-effects. Our "unrestricted" semantics is based on Katz and Weise's solution. However, we are much more conservative than Paratran for side-effects. Indeed, in Paratran, side-effects are performed optimistically: a run-time system detects data dependency violations and is able to correct them by restoring a previous state by a roll-back mechanism. Unfortunately, the run-time system is so expensive that it penalises sequential programs too much. Katz and Weise also suggest a *synchronising* variant of future, which seems closer to our "future-restrictive" semantics; however, it permits less speculation because it requires killing illegitimate tasks.

Feeley's thesis [3] is a deep study of the performance of Katz and Weise's implementation schema. He observes that the implementation of this semantics of future requires capturing the continuation of each future, so that it can be called when multiple returns occur. Feeley shows that by using a lazy task creation mechanism [14], i.e. delaying the creation of the task until it needs to be transferred to another processor, the capture of continuations can be postponed until the time of steal, where it has to be done anyway. Hence, good performance can be achieved by combining the semantics with lazy task creation.

Feeley discusses the cost of supporting legitimacy. He distinguishes legitimacy propagation (rule *determine*) and legitimacy testing ($\ell \rightsquigarrow \ell_i$), and proposes techniques to reduce their costs. He also addresses the issue of the runtime cost of touching placeholders; variants that avoid recursive touching are proposed.

6 Conclusion

The design of a parallel language based on future with side-effects and first-class continuations has been a long-standing open problem. We present the first semantics of such a language and prove its correctness. Such a semantics is useful for the derivation of a proven-correct compiler and for static analysis of parallel programs. The lower-lever framework presented is also suitable to express less speculative variants of the semantics.

A distributed implementation of Scheme based on the proposed semantics is being developed using Queinnec's DMeroon [20], a library offering facilities to distribute objects, and to maintain their coherency over the network.

7 Acknowledgement

Thanks to David De Roure, David Barron, and the anonymous referees for their comments.

References

[1] Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. Technical Report AI Memo 454, M.I.T., Cambridge, Massachussets, March 1977.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.

[3] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[4] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).

[5] Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *Proc. Conf. on Parallel Architecture and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 206–223. Springer-Verlag, 1987.

[6] Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992. Technical Report 100, Rice University, June 1989.

[7] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995. Technical Reports 238, 239, Rice University, 1994.

[8] Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages*

and Systems. US/Japan Workshop on Parallel Lisp. Japan., number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.

[9] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, Florida, June 1994.

[10] Takayasu Ito and Manabu Matsui. A Parallel Lisp Language Pailisp and its Kernel Specification. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 58–100. Springer-Verlag, 1990.

[11] Takayasu Ito and Tomohiro Seino. On Pailisp Continuation and its Implementation. In *Proceedings of the ACM SIGPLAN workshop on Continuations CW92*, pages 73–90, San Francisco, June 1992.

[12] Morry Katz and Daniel Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, June 1990.

[13] James S. Miller. *MultiScheme : A Parallel Processing System Based on MIT Scheme.* PhD thesis, MIT, 1987.

[14] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation : a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[15] Luc Moreau. The PCKS-machine. An Abstract Machine for Sound Evaluation of Parallel Functional Programs with First-Class Continuations. In *European Symposium on Programming (ESOP'94)*, number 788 in Lecture Notes in Computer Science, pages 424–438, Edinburgh, Scotland, April 1994. Springer-Verlag.

[16] Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* PhD thesis, University of Liège, Service d'Informatique, Institut Montefiore B28, 4000 Liège, Belgium, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.

[17] Luc Moreau. Non-speculative and Upward Invocation of Continuations in a Parallel Language. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'95)*, number 915 in Lecture Notes in Computer Science, pages 726–740, Aarhus, Denmark, May 1995.

[18] Luc Moreau. The Semantics of Scheme with Future. Technical report, University of Southampton, 1995.

[19] Luc Moreau and Daniel Ribbens. The Semantics of pcall and fork. In R. Halstead, T. Ito, and C. Queinnec, editors, *PSLS 95 – Parallel Symbolic Langages and Systems*, Beaune, France, October 1995.

[20] Christian Queinnec. DMEROON: a Distributed Class-based Causally-coherent Data Model: Preliminary Report. In *Parallel Symbolic Languages and Systems.*, Beaune, France, October 1995.

[21] Mukund Raghavachari and Anne Rogers. A Case Study in Language Support for Irregular Parallelism: Blocked Sparsed Cholesky. In *Parallel Symbolic Languages and Systems.*, Beaune, France, October 1995.

[22] Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[23] Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic and Computation, Special Issue on Continuations*, 6(3/4):289–360, November 1993.

[24] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(2), 1992.

[25] Pete Tinker and Morry Katz. Parallel Execution of Sequential Scheme with ParaTran. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 28–39, Snowbird, Utah, July 1988.