Continuing Into the Future: the Return
(Invited Paper)

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

Abstract

future is an annotation that indicates which expressions of a program may be evaluated in parallel. By definition, future is transparent, i.e. annotated programs return the same results as in the absence of annotations. In such a framework, the interaction of parallelism and first-class continuations has been considered as a delicate matter for a long time. Indeed, unrestricted parallelism and first-class continuations may lead to non-deterministic programs, which is contradictory to the notion of annotation. In this paper, we overview the formal semantics of future and first-class continuations. The semantics is an abstract machine that models a parallel computer with a shared memory.

1 Introduction

The continuation of an expression is defined as the computation that remains to be performed after evaluating the expression [22]. Some languages, like Scheme [21, 23] or Standard ML of New-Jersey [1] provide the programmer with *first-class* continuations; in these languages, continuations have the same status as numerical values, i.e. they can be passed in argument to or returned by functions, or stored in data structures. First-class continuations are useful to program control structures like backtracking, coroutines, engines, or exceptions [6, 8, 9].

Our interest is to design a language that would allow the programmer to build parallel applications easily. In the approach called "*parallelism by annotations*", programming languages are extended with annotations by which the programmer points out the expressions that may be evaluated in parallel. By definition, annotations must be *transparent*, that is, annotated terms must return the same result as in the absence of annotations. This approach to parallelism is high-level because transparent annotations avoid the programmer to concentrate on parallelism-specific problems such as deadlocks, race conditions, and non-determinism.

The annotation future, initially proposed by Baker and Hewitt [2], is the construct providing parallelism in Halstead's MultiLisp [7]. Intuitively, an expression (future exp) immediately returns a new value, called *placeholder*, which is a data structure with one slot; in parallel, a new task is created to evaluate the argument of future and to store its value in the placeholder. Hence, future creates a "producer-consumer" type of parallelism, where a producer task computes the value of exp in parallel with a task which continues the evaluation as if it already had obtained the value of exp.

However, providing transparent annotations is not a trivial task, especially in the presence of first-class continuations. Indeed, using first-class continuations, one can write programs whose final values depend on the evaluation order. If we add parallelism to the language in a uncontrolled way, the evaluation order changes, and the value of programs may become non-deterministic, which is contrary to the notion of transparent annotation.

The simple following program illustrates the problem:

$$\text{(callcc } (\lambda\, exit.\ \text{(cons (future } (exit\ 1)\text{) (future } (exit\ 2)\text{))))} \qquad (1)$$

Each future creates a new task to evaluate its argument, and returns a new placeholder as value. As a result, this expression leads to three tasks running in parallel and respectively evaluating ($exit$ 1), ($exit$ 2), and the cons primitive. Applying the continuation $exit$ on a value has the effect of returning the value as a final result. Consequently, if parallelism is not controlled, three different results may be returned: 1, 2, or a pair.

*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.

So far, implementation and efficiency questions [3, 7, 10, 11, 12, 13, 24] have mainly motivated research on the interaction between annotation-based parallelism and first-class continuations in Lisp-like. Recently only, the author of this paper defined a semantic framework for functional programs with first-class continuations and annotations for parallelism [14, 15, 16, 17, 18].

In their LFP'90 paper, entitled "Continuing Into the Future", Katz and Weise [12] describe an implementation of future in the presence of first-class continuations. The goal of this paper is to present the formal semantics of future in this framework; its proof of correctness can be found in [17].

2 The Formal Semantics

Our purpose is to define the semantics of a functional language with first-class continuations. Its set of terms Λ_f, defined in Figure 1, is a λ-calculus extended with conditionals, constants, a primitive callcc to capture continuations, and a future construct. The semantics is formalised by the F-PCEKS-machine, which is an abstract machine that models a parallel computer with a shared memory in the tradition of MultiLisp systems. It extends Felleisen and Friedman's CEK-machine [4].

The CEK-machine [4] is an abstract machine able to evaluate sequential functional programs with first-class continuations, i.e. programs of Λ_f without the construct future. Its configuration, called a *computational state*, can be either $\mathsf{Ev}\langle M, \rho, \kappa\rangle$ or $\mathsf{Ret}\langle V, \kappa\rangle$. The first configuration models the evaluation of a term M in an environment ρ, with a continuation κ, while the second configuration designates the return of a value V to a continuation κ. Continuations, representing the rest of the computation, are encoded by a data structure called *continuation code*. A complete definition of the state space appears in Figure 1. Let us observe that closures are represented as $\langle \mathsf{cl}\ \lambda x.M, \rho\rangle$, while first-class continuation are denoted by $\langle \mathsf{co}\ \kappa\rangle$.

Transitions rules of the CEK-machine are displayed in the first part of Figure 2. Details about these transitions can be found in [4, 15]. In order to evaluate a program P, we start the CEK-machine in an initial configuration $\mathsf{Ev}\langle P, \emptyset, (\mathbf{init})\rangle$ and end the computation if a configuration $\mathsf{Ret}\langle V, (\mathbf{init})\rangle$ can be reached. The final answer is defined as $Unload[V, \emptyset]$. According to the function $Unload$, answers are values where closures and first-class continuations are replaced by a tag **procedure**.

The F-PCEKS-machine generalises the CEK-machine by providing parallelism. It is composed of several CEK-configurations that have access to a shared memory. Each CEK-configuration represents a task in the parallel machine; new tasks can be created with the future construct. The second part of Figure 2 displays the transitions related to parallelism in the F-PCEKS-machine.

In the introduction, we described future as a construct creating a producer-consumer type of parallelism. Synchronisations between the producer and consumer take place when the consumer requires the content of the placeholder, while its value is still being computed by the producer. Requiring the value of a placeholder is called *touching* the placeholder. It is performed by primitives like `+`, `*`, `car`, `cdr`, which can only be executed if they receive the actual value of their argument; these primitives are said to be *strict*. In our semantics, we suppose that a touch primitive is explicitly introduced for each strict operation by a translation $\mathcal{X}$. Let us observe that touch is also introduced in operator position of applications and in predicate position of conditionals.

A configuration, or *state*, of the F-PCEKS-machine is represented by a set of active tasks, a store, and a set of suspended tasks. Each task is a triple composed of a CEK-configuration, a legitimacy (to be explained), and a task name.

The semantics of the construct future is given by rule (*fork*). It allocates a new placeholder ph, which is a data structure represented by $\langle \mathsf{ph}\ \alpha\rangle$, referring to a new location α in the shared store. Then, it creates a new task $\mathsf{Ret}\langle ph, \kappa\rangle$, which continues the evaluation with the placeholder as if it already had received the value of the argument of future. In parallel, the initial task evaluates the argument of future, with a continuation $(\kappa\ \mathbf{det}(ph, \ell))$ which means that the value to be obtained should be stored in the placeholder. Parallelism is modelled in the machine by the possibility to evaluate any task in the set of active tasks.

When the argument of future gets evaluated, rule (*determine*) stores the value obtained into the location associated with the placeholder and removes the current task from the set of active tasks. This action is called *determining* the placeholder to the value. A placeholder whose associated location is empty is said to be *undetermined*.

Rules (*touch*) and (*touch suspend*) deal with the situation where the primitive touch is applied on a value V. Both call the auxiliary function touch on V. If V differs from a placeholder, then V is returned and evaluation proceeds on the value. Otherwise, if V is a determined placeholder, we recursively proceed on its content. Finally, if V is an undetermined placeholder, the associated producer task has not yet produced a value, and the touching

$$
\begin{array}{llll}
M_u \in \Lambda_f & ::= & V_u \mid (M_u\ M_u) \mid (\textsf{if}\ M_u\ M_u\ M_u) \mid (\textsf{future}\ M_u) & \text{(User Term)} \\
V_u \in Value_f & ::= & a \mid f \mid x \mid (\lambda x.M_u) & \text{(User Value)} \\
\mathcal{M} \in State_{fpceks} & ::= & \langle T, \sigma, S \rangle & \text{(State)} \\
t \in Task & ::= & \langle C, \ell \rangle_\tau & \text{(Task)} \\
C \in CoSt & ::= & \textsf{Ev}\langle M, \rho, \kappa \rangle \mid \textsf{Ret}\langle V, \kappa \rangle & \text{(Computational State)} \\
\rho \in Env & ::= & \{(x_1\ V_1)\ \ldots (x_n V_n)\} & \text{(Environment)} \\
S & ::= & \{t_1, \ldots, t_n\} & \text{(Suspended Tasks)} \\
T & = & \{t_1, \ldots, t_n\} & \text{(Active Tasks)} \\
M \in \Lambda_{fpceks} & ::= & V_s \mid (M\ M) \mid (\textsf{if}\ M\ M\ M) \mid (\textsf{future}\ M) & \text{(Term)} \\
V_s \in SValue_{fpceks} & ::= & c \mid x \mid (\lambda x.M) & \text{(Syntactic Value)} \\
W \in PValue_{fpceks} & ::= & c \mid \langle \textsf{cl}\ \lambda x.M, \rho \rangle \mid \langle \textsf{co}\ \kappa \rangle \mid (\textsf{cons}\ V\ V) \mid f_c & \text{(Proper Value)} \\
V \in Value_{fpceks} & ::= & W \mid ph & \text{(Runtime Value)} \\
\kappa \in CCode & ::= & (\mathbf{init}) \mid (\kappa\ \mathbf{fun}\ V) \mid (\kappa\ \mathbf{arg}\ M, \rho) & \text{(Continuation code)} \\
 & \mid & (\kappa\ \mathbf{cond}(M, M, \rho)) \mid (\kappa\ \mathbf{det}\ (ph, \ell)) & \\
ph \in Placeholder & ::= & \langle \textsf{ph}\ \alpha \rangle & \text{(Placeholder)} \\
\ell \in Legitimacy & ::= & \langle \textsf{leg}\ \alpha \rangle & \text{(Legitimacy)} \\
\sigma \in Store_{fpceks} & ::= & \{(\alpha_1\ O_1) \ldots (\alpha_n\ O_n)\} & \text{(Store)} \\
O \in Contents_{fpceks} & ::= & \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)} \\
c \in Const & ::= & a \mid d \mid f \mid f_i & \text{(Constant)} \\
f_c \in PApp & ::= & (\textsf{cons}\ V) & \text{(Partial Application)} \\
g \in AValue & ::= & \langle \textsf{cl}\ \lambda x.M, \rho \rangle \mid f \mid f_i \mid f_c \mid \langle \textsf{co}\ \kappa \rangle & \text{(Applicable Value)} \\
f_n \in NStP & = & \{\textsf{cons}, \textsf{callcc}\} & \text{(Non Strict Primitives)} \\
A \in Answer & ::= & c \mid (\textsf{cons}\ A\ A) \mid \textsf{procedure} & \text{(Answer)} \\
x \in Vars & = & \{x, y, z \ldots\} & \text{(User Variable)} \\
a \in BConst & = & \{\textsf{true}, \textsf{false}, \textsf{nil}, 0, 1, \ldots\} & \text{(Basic Constant)} \\
f \in FConst & = & \{\textsf{cons}, \textsf{car}, \textsf{cdr}, \textsf{callcc}\} & \text{(Functional Constant)} \\
f_i \in IFConst & = & \{\textsf{touch}\} & \text{(Internal Functional Constant)} \\
d \in DConst & = & \{\textsf{error}\} & \text{(Distinguished Constant)}
\end{array}
$$

Touch function:

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

Mandatory descendant:

$\ell_0 \leadsto_\sigma \ell_1$ if

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

Initial Legitimacy: $\ell_i = \langle \textsf{leg}\ \alpha_i \rangle$

Initial Store: $\{(\alpha_i\ \bot)\}$

Store extension:

$\sigma[\alpha \leftarrow V] = \sigma \cup \{(\alpha\ V)\}$ with $\alpha \notin DOM(\sigma)$

Store update:

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

Store Content:

$\sigma(\alpha) = V$ if $(\alpha\ V) \in \sigma$

$$
\begin{array}{rcl}
Unload & : & Value_{fpceks} \to Answer \\
Unload[c, \sigma] & = & c \\
Unload[(\textsf{cons}\ V_1\ V_2), \sigma] & = & (\textsf{cons}\ Unload[V_1, \sigma]\ Unload[V_2, \sigma]) \\
Unload[\langle \textsf{cl}\ \lambda x.M, \rho \rangle, \sigma] & = & \mathbf{procedure} \\
Unload[\langle \textsf{co}\ \kappa \rangle, \sigma] & = & \mathbf{procedure} \\
Unload[\langle \textsf{ph}\ \alpha \rangle, \sigma] & = & Unload[\sigma(\alpha), \sigma]
\end{array}
$$

Explicit translation: $\mathcal{X} : \Lambda_f \to \Lambda_{fpceks}$

$$
\begin{array}{rcl}
\mathcal{X}[\![\textsf{car}]\!] & = & \lambda x.(\textsf{car}\ (\textsf{touch}\ x)) \\
\mathcal{X}[\![\textsf{cdr}]\!] & = & \lambda x.(\textsf{cdr}\ (\textsf{touch}\ x)) \\
\mathcal{X}[\![(\textsf{future}\ M_u)]\!] & = & (\textsf{future}\ \mathcal{X}[\![M_u]\!]) \\
\mathcal{X}[\![(M_{u1}\ M_{u2})]\!] & = & (\lambda m_1 m_2.((\textsf{touch}\ m_1)\ m_2))\ \mathcal{X}[\![M_{u1}]\!]\ \mathcal{X}[\![M_{u2}]\!] \\
\mathcal{X}[\![(\lambda x.M)]\!] & = & (\lambda x.\mathcal{X}[\![M]\!]) \\
\mathcal{X}[\![(\textsf{if}\ M_{u1}\ M_{u2}\ M_{u3})]\!] & = & (\textsf{if}\ (\textsf{touch}\ \mathcal{X}[\![M_{u1}]\!])\ \mathcal{X}[\![M_{u2}]\!]\ \mathcal{X}[\![M_{u3}]\!]) \\
\mathcal{X}[\![x]\!] & = & x \text{ if } x \in Vars \cup BConst \cup NStP
\end{array}
$$

Environment Extension:

$\rho[X \leftarrow V] = \rho \cup \{(x\ V)\}$

Environment Access:

$\rho(x) = V$ if $(x\ V) \in \rho$

Figure 1: State space of the F-PCEKS-machine

$$
\begin{array}{rcll}
\mathsf{Ev}\langle(M\ N),\rho,\kappa\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M,\rho,(\kappa\ \mathbf{arg}\ N,\rho)\rangle & (operator)\\
\mathsf{Ev}\langle\lambda x.M,\rho,\kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle\langle\mathsf{cl}\ \lambda x.M,\rho\rangle,\kappa\rangle & (lambda)\\
\mathsf{Ev}\langle c,\rho,\kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle c,\kappa\rangle & (constant)\\
\mathsf{Ev}\langle x,\rho,\kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle\rho(x),\kappa\rangle & (variable)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{arg}\ N,\rho)\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle N,\rho,(\kappa\ \mathbf{fun}\ V)\rangle & (operand)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ \langle\mathsf{cl}\ \lambda x.M,\rho\rangle)\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M,\rho[x\leftarrow V],\kappa\rangle & (apply)\\
\mathsf{Ev}\langle(\mathsf{if}\ M\ M_1\ M_2),\rho,\kappa\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M,\rho,(\kappa\ \mathbf{cond}\ (M_1,M_2,\rho))\rangle & (predicate)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{cond}\ (M_1,M_2,\rho))\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M_2,\rho,\kappa\rangle\quad \text{if } V=\mathsf{false} & (if\ else)\\
 & \rightarrow_{cek} & \mathsf{Ev}\langle M_1,\rho,\kappa\rangle\quad \text{if } V\neq\mathsf{false} & (if\ then)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ \mathsf{callcc})\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle\langle\mathsf{co}\ \kappa\rangle,(\kappa\ \mathbf{fun}\ V)\rangle & (capture)\\
\mathsf{Ret}\langle V,(\kappa'\ \mathbf{fun}\ \langle\mathsf{co}\ \kappa\rangle)\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle V,\kappa\rangle & (invoke)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ V_1)\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle(V_1\ V),\kappa\rangle\quad \text{if } (V_1\ V)\in PApp & (partial\ apply)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ (\mathsf{cons}\ V_1))\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle(\mathsf{cons}\ V_1\ V),\kappa\rangle & (cons)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ \mathsf{car})\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle V_1,\kappa\rangle\quad \text{if } V=(\mathsf{cons}\ V_1\ V_2) & (car)\\
 & \rightarrow_{cek} & \mathsf{Ret}\langle\mathsf{error},(\kappa\ \mathbf{fun}\ \langle\mathsf{co}\ (\mathbf{init})\rangle)\rangle\quad \text{if } V\neq(\mathsf{cons}\ V_1\ V_2) & (car\ error)\\
\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ V_1)\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle\mathsf{error},(\kappa\ \mathbf{fun}\ \langle\mathsf{co}\ (\mathbf{init})\rangle)\rangle\ \text{if } V_1\notin AValue & (apply\ error)
\end{array}
$$

$$
\begin{array}{lr}
\langle\{\ \langle C,\ell\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle & \\
\quad\rightarrow_{fpceks}\quad \langle\{\ \langle C_1,\ell\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle\quad \text{if } C\rightarrow_{cek} C_1 & (sequential)\\
\langle\{\ \langle\mathsf{Ev}\langle(\mathsf{future}\ M),\rho,\kappa\rangle,\ell\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle & \\
\quad\rightarrow_{fpceks}\quad \langle\{\ \langle\mathsf{Ev}\langle M,\rho,(\kappa\ \mathbf{det}\ ph,\ell_1)\rangle,\ell\rangle_\tau,\ \ \langle\mathsf{Ret}\langle ph,\kappa\rangle,\ell_1\rangle_{\tau_1}\ \}\ \cup\ T,\sigma_1,S\rangle & (fork)\\
\qquad\qquad \text{with } ph=\langle\mathsf{ph}\ \alpha\rangle,\ell_1=\langle\mathsf{leg}\ \alpha_1\rangle,\alpha_1\notin DOM(\sigma),\alpha\notin DOM(\sigma) & \\
\qquad\qquad \text{with } \sigma_1=\sigma[\alpha\leftarrow\bot][\alpha_1\leftarrow\bot],\tau_1\notin FN(T\cup S)\cup\{\tau\} & \\
\langle\{\ \langle\mathsf{Ret}\langle V,(\kappa\ \mathbf{det}\ \langle\mathsf{ph}\ \alpha\rangle,\langle\mathsf{leg}\ \alpha_1\rangle)\rangle,\ell_2\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle & \\
\quad\rightarrow_{fpceks}\quad \langle T_1,\sigma_1,S_1\rangle,\ \ \text{if }\ \sigma(\alpha){=}\bot & (determine)\\
\qquad\qquad T_1=(T\cup T_2),S_1=(S\setminus T_2),\sigma_1=\sigma[\alpha_1:=\ell_2][\alpha:=V] & \\
\qquad\qquad \text{with } T_2=\{\langle\mathsf{Ret}\langle\langle\mathsf{ph}\ \alpha\rangle,(\kappa'\ \mathbf{fun}\ \mathsf{touch})\rangle,\ell_3\rangle_{\tau_1}\in S\} & \\
\quad\rightarrow_{fpceks}\quad \langle\{\ \langle\mathsf{Ret}\langle V,\kappa\rangle,\ell_2\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle,\ \ \text{if }\ \sigma(\alpha){\neq}\bot & (determinen)\\
\langle\{\ \langle\mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle,\ell\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle & \\
\quad\rightarrow_{fpceks}\quad \langle\{\ \langle\mathsf{Ret}\langle\mathrm{touch}_\sigma(V),\kappa\rangle,\ell\rangle_\tau\ \}\ \cup\ T,\sigma,S\rangle & (touch)\\
\qquad\qquad \text{if } \mathrm{touch}_\sigma(V)\in PValue & \\
\quad\rightarrow_{fpceks}\quad \langle T,\sigma,\{\ \langle\mathsf{Ret}\langle\langle\mathsf{ph}\ \alpha\rangle,(\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle,\ell\rangle_\tau\ \}\ \cup\ S\rangle & (touch\ suspend)\\
\qquad\qquad \text{if } \mathrm{touch}_\sigma(V)=\langle\mathsf{ph}\ \alpha\rangle,\ \sigma(\alpha){=}\bot &
\end{array}
$$

$$
\langle T,\sigma\rangle \quad \rightarrow^{1,a}_{fpceks} \quad \langle T',\sigma'\rangle \text{ if } \langle T,\sigma\rangle\rightarrow_{fpceks}\langle T',\sigma'\rangle,\ a=1 \text{ if } \ell\rightsquigarrow_\sigma \ell_i, \text{ and } a=0 \text{ otherwise.} \qquad (base)
$$

$$
\mathcal{M} \quad \rightarrow^{a+a',b+b'}_{fpceks} \quad \mathcal{M}_2 \text{ if } \mathcal{M}\rightarrow^{a,b}_{fpceks}\mathcal{M}_1 \text{ and } \mathcal{M}_1\rightarrow^{a',b'}_{fpceks}\mathcal{M}_2. \qquad (transitive)
$$

$$
eval_{fpceks}(P) = \begin{cases}
Unload[V,\sigma] & \text{if } \langle\ \{\ \langle\mathsf{Ev}\langle\mathcal{X}[\![P]\!],\emptyset,(\mathbf{init})\rangle,\ell_i\rangle_{\tau_0}\ \}\ \ ,\sigma_i,\emptyset\rangle\rightarrow^*_{fpceks}\langle T,\sigma,S\rangle,\\
 & \quad \text{with } \langle\mathsf{Ret}\langle V,(\mathbf{init})\rangle,\ell\rangle_\tau\in T, \text{ such that } \ell\rightsquigarrow_\sigma\ell_i\\
\bot & \text{if } \forall j\in\mathbb{N},\exists\mathcal{M}_j\in State_{fpceks},n_j,m_j\in\mathbb{N}, \text{ such that}\\
 & \quad \mathcal{M}_0=\langle\ \{\ \langle\mathsf{Ev}\langle\mathcal{X}[\![P]\!],\emptyset,(\mathbf{init})\rangle,\ell_i\rangle_{\tau_0}\ \}\ \ ,\sigma_i,\emptyset\rangle,\\
 & \quad \mathcal{M}_j\rightarrow^{n_j,m_j}_{fpceks}\mathcal{M}_{j+1},m_j>0
\end{cases}
$$

Figure 2: Evaluator specification of the F-PCEKS-machine

task should be suspended, by transferring it to the set of suspended task. A suspended task is awakened when the placeholder get determined by rule (*determine*).

So far, we deliberately avoided to talk about continuations, and did not comment on legitimacies and rule (*determinen*). Placeholders are data structures that can receive at most one value: once determined, a placeholder remains constant. This invariant is automatically satisfied in the absence of first-class continuations by the functional nature of the language. However, when using first-class continuations, an expression may return "several times", i.e. several values may be passed to its continuation. Therefore, in order to preserve the property of placeholders, rule (*determine*) is allowed to be fired only if the placeholder is undetermined. (We assume here the atomicity of the transitions.) If the placeholder is already determined, rule (*determinen*) evaluates the continuation as if no future had existed (cfr. Katz and Weise [12]).

However, this does not guarantee the transparency of the future construct. Indeed, the program (1) can still return several final results. Following Katz and Weise, we use a notion of legitimacy to distinguish *mandatory* from *speculative* tasks. The mandatory task is the one that performs the transition that would be performed if evaluation had been sequential; all the other tasks are speculative.

A legitimacy $\langle \mathsf{leg}\ \alpha\rangle$, like a placeholder, 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.

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 legitimacy ℓ_1 in addition to the new placeholder. After transition (*fork*), task τ still has the same legitimacy, but is now evaluating the future argument 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*) keeps track of legitimacy as follows. If the placeholder $\langle \mathsf{ph}\ \alpha\rangle$ gets determined to a value V, the task that speculatively consumes this placeholder 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 $\ell_1 \leadsto_\sigma \ell_2$ 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 $\leadsto$ is used to determine whether a final value, i.e. a value returned to the $(\mathbf{init})$ continuation, is a valid answer. A valid answer is produced by the task whose legitimacy ℓ is such that $\ell \leadsto_\sigma \ell_i$. 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.

The third part of Figure 2 defines the evaluation relation of the F-PCEKS-machine. The evaluation of a program P begins with an initial task $\langle \mathsf{Ev}\langle P, \emptyset, (\mathbf{init})\rangle, \ell_i\rangle_{\tau_0}$ and ends if a final configuration can be reached; a final configuration contains a task $\langle \mathsf{Ret}\langle V, (\mathbf{init})\rangle, \ell\rangle_\tau$, such that $\ell \leadsto_\sigma \ell_i$, i.e. the task returning the value is mandatory.

Let us note that there exist programs with a finite mandatory computation but a possible unbounded number of speculative transitions, like for instance $(\mathsf{callcc}\lambda k.\ \ ((\mathsf{future}\ (k\ 1))\ \Omega))$, where Ω is a sequential divergent term. Therefore, divergence should be defined with the greatest care: a program is said to diverge if it regularly often performs mandatory transitions. Hence, we introduce the relation $\mathcal{M}_1 \rightarrow^{n,m} \mathcal{M}_2$ to denote a transition between two configurations $\mathcal{M}_1$ and $\mathcal{M}_2$, involving n steps among which m were mandatory.

3 Related Work and Conclusion

This paper overviewed the semantics of future in the presence of first-class continuations. Details and proof of correctness can be found in [17, 18]. The same framework is also extended to deal with side-effects. Flanagan and Felleisen [5] defined the semantics of future in a purely functional language. Their goal was to derive a program optimisation, called the *touch optimisation*, which removed provably redundant touch operations. Such an optimisation would also be useful for the language we deal with.

Katz and Weise [12] introduced the notion of *legitimacy*. For a deep study of the performance of MultiLisp, we refer to Feeley's thesis [3]. Halstead [7, page 19] gave three criteria which should be satisfied by an implementation with transparent annotations for parallelism. Thanks to the proof of correctness, our semantics satisfies these criteria.

There exist other approaches to parallelism which do not preserve the sequential meaning of programs. Among others, let us cite Queinnec's ICSLAS [19, 20], and Ito's PAILISP [10, 11], which both define new semantics of continuations in a parallel framework. Let us observe that their goals differ from ours because their primitives for parallelism add expressiveness to the sequential core, like for instance Queinnec's `pcall` or Ito's *parallel-or*.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, number LNCS 274, pages 301–324. Springer–Verlag, 1987.

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

[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.

[5] 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.

[6] Daniel P. Friedman and Christopher T. Haynes. Constraining Control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 245–254, New Orleans, LA., January 1985. ACM.

[7] 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*, LNCS 441, pages 2–57. Springer-Verlag, 1990.

[8] Christopher T. Haynes and Daniel P. Friedman. Abstracting Timed Preemption with Engines. *Comput. Lang.*, 12(2):109–121, 1987.

[9] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.

[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*, LNCS 441, 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] 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)*, LNCS 788, pages 424–438, Edinburgh, 1994.

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

[16] 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)*, LNCS 915, pages 726–740, Aarhus, 1995.

[17] Luc Moreau. The Semantics of Scheme with Future. Technical Report M95/7, University of Southampton, 1995. 56 pages.

[18] Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, Philadelphia, May 1996.

[19] Christian Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[20] Christian Queinnec and David De Roure. Design of a Concurrent and Distributed Language. In A. Agarwal, R. H. Halstead, and Takayasu Ito, editors, *Proceedings of the Workshop on Parallel Symbolic Computing: Languages, Systems and Applications*, Boston, Massachussetts, October 1992.

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

[22] Christopher Strachey and Christopher P. Wadsworth. A Mathematical Semantics for Handling Full Jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

[23] Gerald Jay Sussman and Guy Lewis Steele, Jr. SCHEME: an Interpreter for Extended Lambda Calculus. Technical Report 349, MIT AI Lab, December 1975.

[24] 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.