Correctness of a Distributed-Memory Model for Scheme

Luc Moreau

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

Abstract. We propose a high-level approach to program distributed applications; it is based on the annotation \texttt{future} by which the programmer specifies which expressions may be evaluated remotely in parallel. We present the CEKDS-Machine, an abstract machine with a distributed memory, able to evaluate Scheme-like \texttt{future}-based programs. In this paper, we focus on the issue of task migration and prove that task migration is transparent to the user, i.e., task migration does not change the observable behaviour of programs.

1 Introduction

Distributed systems are omnipresent: local area networks and the success of the Internet in the past years are particular illustrations of the ubiquity of distributed computing. A major research focus in this area has been the design of new languages or programming paradigms to develop distributed applications, like e.g. PV, PVM [10], MPI [8], Nexus [9], Cilk [1]. We argue that those systems were designed to build high-performance distributed applications, and that they favour efficiency over ease of programming. Therefore, these languages or paradigms overwhelm the programmer with the burden of dealing with the complexity of distribution. Some approaches even impose programming styles, with which the programmer may not be familiar: e.g. Cilk [1] demands programs written in continuation-passing style.

Mostly-functional languages like Scheme and SML have traditionally provided the programmer with abstraction, expressiveness, first-class citizenship of objects, and automatic garbage collection. We believe that there is a niche for a high-level approach to distributed computing. Following Halstead's work on MultiLisp [11], we extend a Scheme-like language with an annotation \texttt{future} by which the programmer specifies which expressions may be evaluated in parallel, possibly remotely. By definition, annotations must be transparent, i.e., annotated programs return the same result as in the absence of annotations. This approach is abstract because it hides the intricacies of distribution by giving the programmer the illusion that a distributed system is programmable as a sequential one.

We consider the idealised Scheme-like language defined in Figure 1. It is a purely functional language, extended with a primitive \texttt{makeref} to create boxes, with primitives \texttt{deref}, \texttt{setref} to read and modify them, and with a primitive \texttt{callcc} to capture first-class continuations. In addition, there is a construct \texttt{(future M)} to create a producer-consumer type of parallelism [11]. Intuitively, the evaluation of \texttt{(future M)} immediately returns an object called placeholder, while another task evaluates the argument \texttt{M} in parallel. The purpose of the latter task, called the \texttt{producer}, is to compute and then store the value of \texttt{M} in the placeholder. The task using the placeholder is called the \texttt{consumer}.

For a long time, this approach has been characterised by a lack of formal semantics due to the difficulty of providing transparent annotations for parallelism in the presence of first-class continuations and side-effects. Recently, the author [20] defined the semantics of \texttt{future} for the language of Figure 1. The goal of this paper is to extend this semantics to a distributed framework (the proof of its correctness is available in a technical report [21]).

More specifically, the contributions of this paper are the following. \textit{i)} We define a distributed architecture able to evaluate \texttt{future}-based programs; \textit{ii)} We prove that

\footnote{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.}
task migration is transparent to the programmer, i.e. task migration does not change the observable behavior of programs. This result is the reference semantics that we can use to design and prove the correctness of program optimizations. The architecture is described in Section 2, Section 3 discusses related work and is followed by concluding remarks.

2 The Architecture

In this section, we present the CEKDS Machine, an abstract machine with a distributed store, which extends Felkens' and Friedman's CEK and CEKs machines [4, 5], and the E-PCKS machine [19]; its state space is formally described in Figure 2.

Some operations like deref, i.e. reading the content of a box, are rather complex. Indeed, as deref is strict, it touches its argument, checks whether it is legitimate to access the content of the box received in argument, and finally, reads the box content. In order to distinguish these three operations, we add two primitives touch and sync to $A_{touch}$, the language accepted by the CEKDS machine; besides, we translate every program of $A_{f}$ into a term of $A_{touch}$ by the function $A_{f}$ of Figure 2, which makes the touch and sync operations explicit.

In our distributed architecture, computational resources are called sites and are uniquely identified by site names. A site has its own memory and can run several tasks that share the site memory. A world is the set of sites that can be used to evaluate a program; sites in a world communicate by exchanging messages. More specifically, in a site, we distinguish active tasks, i.e. tasks that can be run, from suspended tasks, i.e. tasks that wait for a message or a synchronization. As far as communications are concerned, a site is equipped with two spools of messages; the input spool contains pending input messages, while the output spool contains the messages that remain to be transmitted.

A store is a finite function, also represented as a set of pairs, associating locations with store contents. In our distributed architecture, each site has its own procedure of memory allocation, and its proper task naming mechanism. Hence, we use the notion of qualified location or task name, to unambiguously refer to a location or a task in the world. We now appreciate how site names uniqueness is important to define qualified locations or task names.

We abstract a task by a triple composed of a computational state, a legitimacy [15, 20] used to implement first-class continuations and side-effects, and a name. A computational state is a configuration of the CEK-machine [4], which can be either $Ev(M, p, s)$ representing the evaluation of a term $M$ in an environment $p$ with a continuation $s$, or $Ret(M, s)$ meaning the return of a value to a continuation. The continuation, implemented as a data structure called continuation code, represents what remains to do after evaluating $M$ in $Ev(M, p, s)$. In conventional languages, the continuation is nothing else but the evaluation stack. The environment is a finite function mapping variables to values.

In Figure 3, transitions between computational states specify how to evaluate the purely functional and sequential subset of the language extended with first-class continuations; details can be found in [4, 20, 22, 19]. Figure 4 shows the transitions that involve a collaboration of a task with its site. According to (fork), the evaluation of a future allocates a new placeholder $ph$ and creates a new task, with a name $\tau$, which speculatively evaluates the

---

\[ P \in A_f \quad M \in A_f \]  
\[ V \in Values \]  
\[ v \in BConst = \{ \text{true, false, int} \} \]  
\[ f \in FConst = \{ \text{cons, car, cdr, makef, deref, setsref, calloc} \} \]  
\[ x \in Vars = \{ x, y, z \ldots \} \]  

**Fig. 1. Syntax of $A_f$**

---

2 A strict function applied to a placeholder, accesses the value that the producer task has stored in the placeholder; this operation, called touching the placeholder, can suspend the current task when the placeholder has not received a value yet.
\[ W \quad ::= \quad \{ m_1, \ldots, m_n \} \quad \text{(World)} \quad \text{Explicit translation: } X : A \rightarrow A_{\text{code}} \]
\[ m \in M \quad ::= \quad \langle \ell, V, S, L, O \rangle \quad \text{(Site)} \quad X[\ell] = x \text{ if } x \in \text{Vars} \cup \text{BConst} \cup \text{NSIP} \]
\[ t \in \text{Task} \quad ::= \quad \langle \ell, C, \ell, \tau \rangle \quad \text{(Task)} \quad X[\ell] = \lambda x.(\text{car} \hspace{1mm} (\text{touch} \hspace{1mm} x)) \]
\[ \theta \in \text{Store} \quad ::= \quad \langle \alpha_1, X_1 \rangle, \ldots, \langle \alpha_n, X_n \rangle \quad \text{Store} \quad X[\alpha] = \lambda \langle \text{car} \rangle (\text{touch} \hspace{1mm} \alpha) \]
\[ X \quad ::= \quad V \mid \ell \quad \text{State Content} \quad X[\text{deref}] = \lambda x.(\text{deref} \hspace{1mm} (\text{touch} \hspace{1mm} x)) \]
\[ s \in \mathcal{S} \quad ::= \quad \{ s_1, \ldots, s_n \} \quad \text{(Site Name)} \quad X[\text{setref}] = \lambda x.(\text{setref} \hspace{1mm} (\text{touch} \hspace{1mm} s_1)) \]
\[ \tau \in \tau \quad ::= \quad \langle \tau_1, \ldots, \tau_n \rangle \quad \text{(Task Name)} \quad \text{future} \quad X[\ell] = \lambda \langle \text{ac} \rangle \langle \text{future} \rangle \]
\[ u \quad ::= \quad \langle \gamma, \delta \rangle \quad \text{(Qualified Task Name)} \quad X[M_1, M_2] = \langle \gamma, \delta, M_1 \rangle \langle \text{touch} \hspace{1mm} M_2 \rangle \]
\[ a \quad ::= \quad \langle \alpha, \delta \rangle \quad \text{(Qualified Location)} \quad X[\alpha] = \lambda \langle \text{car} \rangle \langle \text{touch} \hspace{1mm} \alpha \rangle \]
\[ X[M_1, M_2] = \langle \alpha, \delta, M_1 \rangle \langle \text{touch} \hspace{1mm} M_2 \rangle \]

\[ C \in \text{CoSt} \quad ::= \quad \text{Ev}(M, a, k) \quad \text{(Computational State)} \quad \text{Unload}(\gamma, \delta) = \epsilon \]
\[ \rho \in \text{Env} \quad ::= \quad \langle \gamma, \delta \rangle \quad \ldots \quad \langle \xi, \zeta \rangle \quad \text{(Environment)} \]
\[ I \quad ::= \quad \{ \gamma_1, \ldots, \gamma_n \} \quad \text{(Input Spool)} \quad \text{Unload}(\lambda \langle \text{ac} \rangle M, \ell) = \text{procedure} \]
\[ O \quad ::= \quad \{ \gamma_1, \ldots, \gamma_n \} \quad \text{(Output Spool)} \quad \text{Unload}(\ell, \delta) = \text{box} \]
\[ q \quad ::= \quad \{ \gamma_1, \ldots, \gamma_n \} \quad \text{(Message)} \quad \text{Unload}(\ell, \delta) = \text{application} \]
\[ S \quad ::= \quad \{ \tau_1, \ldots, \tau_n \} \quad \text{(Suspension Set)} \quad \text{Unload}(\gamma, \delta) = \text{unload} \]
\[ \text{Term} \quad ::= \quad \text{Ev}(M, a, k) \]

\[ M \in A_{\text{code}} \quad ::= \quad V \mid (M \mid M) \quad \text{(Term)} \quad \text{Free Task Name:} \quad F(T) = \tau \quad \forall \langle \tau, C, \ell, \gamma \rangle \in \tau \]
\[ V \in \text{SValue} \quad ::= \quad \ell \mid \langle \text{cons} \rangle \langle V \rangle \mid \langle \text{car} \rangle \langle \text{nat} \rangle \quad \text{(Syntactic Value)} \]
\[ W \in \text{PValue} \quad ::= \quad \ell \mid \langle \text{cons} \rangle \langle V \rangle \mid \langle \text{car} \rangle \langle \text{nat} \rangle \quad \text{(Proper Value)} \quad \text{Store Operations:} \quad \text{Unload}(\gamma, \delta) = \epsilon \]
\[ V \in \text{Value} \quad ::= \quad W \mid \ell \quad \text{(Runtime Value)} \quad \text{with } \alpha \notin \text{DOM}(\theta) \]
\[ \phi \quad ::= \quad \langle \phi \rangle \quad \text{(Placeholder) } \theta(\alpha) = V \quad \text{if } (\phi \gamma) \not\equiv \theta \]
\[ b \in \text{Box} \quad ::= \quad \langle b \rangle \quad \text{(Box) } \theta(\alpha) = V \quad \text{if } (\theta \gamma) \not\equiv \theta \]
\[ \ell \in \text{Leg} \quad ::= \quad \langle \text{leg} \rangle \quad \text{(Legitimity) } \not\equiv \theta \]
\[ c \in \text{Const} \quad ::= \quad \epsilon \quad \text{(Constant) } \]
\[ f \in \text{PApp} \quad ::= \quad \langle \text{cons} \rangle \langle \text{setref} \rangle \quad \text{(Partial Application) } \]
\[ g \in \text{AValue} \quad ::= \quad \langle \text{cons} \rangle \langle \text{setref} \rangle \quad \text{(Applicable Value) } \theta(\alpha) = \theta \]
\[ p \in \text{BConst} \quad ::= \quad \{ \text{true}, \text{false}, \text{nil}, \ldots, \text{void} \} \quad \text{(Basic Constant) } \]
\[ k \in \text{CCode} \quad ::= \quad \langle \text{_init} \rangle \quad \text{(Initiation code) } \]
\[ \kappa \in \text{CCode} \quad ::= \quad \langle \text{call} \rangle \quad \text{(Call code) } \]
\[ \rho \in \text{BConst} \quad ::= \quad \{ \text{true}, \text{false}, \text{nil}, \ldots, \text{void} \} \quad \text{(Basic Constant) } \]
\[ \text{Environment Operations:} \quad f \in \text{PApp} \quad ::= \quad \langle \text{cons} \rangle \langle \text{setref} \rangle \quad \text{(Non Primitive) } \]
\[ x \in \text{Vars} \quad ::= \quad \langle x, y, \ldots \rangle \quad \text{(User Variable) } \]
\[ \kappa \in \text{CCode} \quad ::= \quad \langle \text{init} \rangle \quad \text{(Initialization code) } \]
\[ \rho \in \text{BConst} \quad ::= \quad \{ \text{true}, \text{false}, \text{nil}, \ldots, \text{void} \} \quad \text{(Basic Constant) } \]
\[ \text{Environment Operations:} \quad f \in \text{PApp} \quad ::= \quad \langle \text{cons} \rangle \langle \text{setref} \rangle \quad \text{(Non Primitive) } \]
\[ k \in \text{CCode} \quad ::= \quad \langle \text{call} \rangle \quad \text{(Call code) } \]

\[ \text{Fig. 2: State Space of the CEKDS-machine} \]
\[
\begin{align*}
\text{Ev}((M \cdot N)_{\beta \rho \kappa}) & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} (\kappa \text{ arg } N_{\beta \rho})) & \text{(operator)} \\
\text{Ev}(\lambda x. M_{\beta \rho}) & \Rightarrow_{\text{eval}} \text{Ret}(\lambda x. M_{\beta \rho})_{\kappa} & \text{(lambda)} \\
\text{Ev}(x_{\beta \rho}) & \Rightarrow_{\text{eval}} \text{Ret}(x_{\beta \rho})_{\kappa} & \text{(variable)} \\
\text{Ret}(V_{\kappa \text{ arg } N_{\beta \rho}}) & \Rightarrow_{\text{eval}} \text{Ev}(N_{\beta \rho} (\text{fun } V)) & \text{(operand)} \\
\text{Ret}(V_{\kappa \text{ fun } (c \lambda x. M_{\beta \rho})}) & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} (x \leftarrow V_{1_{\beta \rho}})) & \text{(apply)} \\
\text{Ev}(\text{if } M_{1_{\beta \rho}} V_{2_{\beta \rho}}) & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} (\kappa \text{ cond } (M_{1_{\beta \rho}}, M_{2_{\beta \rho}}))) & \text{(predicate)} \\
\text{Ret}(V_{\kappa \text{ cond } (M_{1_{\beta \rho}}, M_{2_{\beta \rho}})}) & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} \kappa) & \text{if } V = \text{false} \\
& & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} \kappa) & \text{if } V \neq \text{false} \\
& & \Rightarrow_{\text{eval}} \text{Ev}(M_{\beta \rho} \kappa) & \text{(if then)} \\
\text{Ret}(V_{\kappa \text{ fun } \text{callc}}) & \Rightarrow_{\text{eval}} \text{Ret}(\text{callc } \kappa) & \text{(capture)} \\
\text{Ret}(V_{\kappa \text{ fun } \text{call}}) & \Rightarrow_{\text{eval}} \text{Ret}(V_{1_{\beta \rho}}) & \text{(invoke)} \\
\text{Ret}(V_{\kappa \text{ fun } V_{1_{\beta \rho}}}) & \Rightarrow_{\text{eval}} \text{Ret}(V_{1_{\beta \rho}}) & \text{if } V_{1_{\beta \rho}} \in P_{\text{App}} & \text{(partial apply)} \\
\text{Ret}(V_{\kappa \text{ fun } \text{cons } V_{1_{\beta \rho}}}) & \Rightarrow_{\text{eval}} \text{Ret}(\text{cons } V_{1_{\beta \rho}} \kappa) & \text{(cons)} \\
\text{Ret}(\text{cons } V_{1_{\beta \rho}} V_{2_{\beta \rho}} (\kappa \text{ fun } \text{car})) & \Rightarrow_{\text{eval}} \text{Ret}(V_{1_{\beta \rho}} \kappa) & \text{(car)} \\
\text{Ret}(\text{cons } V_{1_{\beta \rho}} V_{2_{\beta \rho}} (\kappa \text{ fun } \text{cdr})) & \Rightarrow_{\text{eval}} \text{Ret}(V_{2_{\beta \rho}} \kappa) & \text{(cdr)} \\
\text{Ret}(V_{\kappa \text{ fun } f}) & \Rightarrow_{\text{eval}} \text{Ret}(f_{\beta \rho}, V_{1_{\beta \rho}} \kappa) & \text{(5)} \\
\text{Ret}(V_{\kappa \text{ fun } V_{1_{\beta \rho}}}) & \Rightarrow_{\text{eval}} \text{Ret}(\text{error } \kappa) & \text{if } V_{1_{\beta \rho}} \notin \text{All values} & \text{(apply error)} \\
\end{align*}
\]

Fig. 3. Transitions between computational states

continuation of future with the placeholder \(ph\). After transition, the initial task \(\tau\) evaluates the argument of future. We shall explain later the purpose of the legitimacies \(\ell\) and \(\ell'\).

A new box can be created by applying the functional constant \texttt{makeref} on a value. As a result, a new location is allocated in the local store, and a new box object, which refers to the new location and the site name, i.e., the qualified location, is returned.

In order to illustrate the behaviour of the machine, we consider the operation of reading a box, which will lead us to explain some rules of Figures 4, 5, 6 and 7; similar comments apply to box modification. A task considers that a box is local if the site name held in the box is the name of the current site. According to \texttt{(deref local)} in Figure 4, if the box is local, the value contained in the local store at the given location can be returned. Otherwise, rule \texttt{(deref remote)} adds to the output spool a \texttt{request} addressed to the site that allocated the box; in addition, the task is suspended, which is modelled by its transfer from the set \(T\) of runnable tasks to the set \(S\) of suspended tasks. We represent requests as triples composed of the destination site, the qualified name of the requesting task, and the message itself describing the type of the request. In the present case, the message \texttt{deref(\alpha)} means that the distant site is asked to supply the content of location \(\alpha\).

Sites communicate according to the rules of Figure 5. In rule \texttt{(migrate request)}, two sites exchange \texttt{requests} by moving them from the output spool of the source site to the input spool of the destination site. Figure 6 shows how a site handles incoming requests. In the case of rule \texttt{(request deref)}, the content of the location is packed up into an answer which must return back to the task that initiated the request; we again can see the interest of the qualified task name which indicates the name of the site that emitted the request. The answer is entered in the output spool and is migrated, like a request, by rule \texttt{(migrate answer)}. Figure 7 shows the rules that handle incoming answers. The arrival of the answer \texttt{deref(V)} awakens the task waiting for this answer by transferring it back to the set of runnable tasks with the value \(V\) as the content of the box.

Rule \texttt{(fork)} allows us to create new tasks on the current site. In Figure 5, rule \texttt{(migrate task)} shows that a task may be migrated from a site 1 running more that one active task to a site 2 without any active task. We see that a task is migrated by transferring its computational state, i.e., among others its continuation, and its legitimacy.
Fig. 4. Site Transitions
\[
\{\langle (C, t_1, \tau) \rangle \cup T_1, \theta, S_1, I, T_2, O_1, \emptyset, (t_2, s_2, S_1, \tau, O_2) \} \cup W \quad \text{(migrate task)}
\]
\[
\rightarrow \langle (C, t_1, \tau_1) \rangle \cup T_2, \theta, S_1, I, T_2, O_1, \emptyset, (t_2, s_2, S_1, \tau, O_2) \} \cup W
\]
\[
\text{if } T_1 \neq \emptyset, \text{ with } \tau_1 \in \text{FN}(S_2)
\]
\[
\{\langle T_1, \theta, s_1, I, T_2, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup O_1, \langle T_2, \theta, s_2, S_2, \tau, O_2 \rangle \} \cup W \quad \text{(migrate request)}
\]
\[
\rightarrow \langle (C, t_1, \tau_1) \rangle \cup T_2, \theta, s_2, S_2, \tau, O_2, \emptyset, (t_2, s_2, S_2, \alpha, O_1) \} \cup W
\]
\[
\{\langle T_1, \theta, s_1, I, T_2, \{ \text{Ans}(\tau_2, s_2, \alpha) \} \cup O_1, \langle T_2, \theta, s_2, S_2, \tau, O_2 \rangle \} \cup W \quad \text{(migrate answer)}
\]
\[
\rightarrow \langle (C, t_1, \tau_1) \rangle \cup T_2, \theta, s_2, S_2, \tau, O_2, \emptyset, (t_2, s_2, S_2, \alpha, O_1) \} \cup W
\]

Fig. 5. Communications between sites

\[
\langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{(request done)}
\]
\[
\langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \rho) \} \cup I, O \rangle
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \rho) \} \cup O \rangle \quad \text{with } \theta_1 = \theta[\alpha := V]
\]
\[
\langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \alpha, \rho) \} \cup O \rangle \quad \text{if } \theta(\alpha) = \bot
\]
\[
\begin{aligned}
\text{with } & \tau_2^1 = (I \cup T_2), T_2^1 = (T_2 \cup T_2), S_2^1 = (S \setminus \{ t_2 \cup T_2 \}), \theta_1 = \theta[\alpha := V]\left[c_{12} := 4\right]
\text{with } & T_2 = \{ \text{Res}(s_2, \beta, \gamma, \text{touch}(\alpha)), \text{Res}(s_2, \beta, \gamma, \text{touch}(\alpha), c_{12}, \alpha, \beta, \gamma) \in S \}
\text{with } & T_2 = \{ \text{Res}(s_2, \beta, \gamma, \text{touch}(\alpha), c_{12}, \alpha, \beta, \gamma) \in S \}
\end{aligned}
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \rho) \} \cup I, O \rangle \quad \text{if } \theta(\alpha) \neq \bot
\]
\[
\langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \text{touch}(\alpha, s_2, \theta_1, s_2) \in \text{Value}(s_2) \neq s
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Ans}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \text{touch}(\alpha, s_2, \theta_1, s_2) \in \text{Value}(s_2) = s
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \text{touch}(\alpha, s_2, \theta_1, s_2) = \text{touch}(\alpha, s_2, \theta_1, s_2) \neq s
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \text{touch}(\alpha, s_2, \theta_1, s_2) = \text{touch}(\alpha, s_2, \theta_1, s_2) \neq s
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \text{touch}(\alpha, s_2, \theta_1, s_2) = \text{touch}(\alpha, s_2, \theta_1, s_2) \neq s
\]
\[
\rightarrow \langle T, \theta, s, S, I, \{ \text{Req}(\tau_2, s_2, \alpha, \rho) \} \cup I, O \rangle \quad \text{if } \theta(\alpha) = \bot, \theta(\alpha) = \bot
\]

Fig. 6. Handling of Requests

According to (fork), the effect of evaluating (future M) is to allocate a placeholder that a new task speculatively passes to its continuation. The original task, which evaluates M, acts as a producer for the placeholder value, while the new task acts as a consumer.

A producer task has obtained the value V of a future argument, when V is returned to a continuation code of the form (s det p, s); the producer task is then expected to store V
in the placeholder $p$; this operation is called \textit{determining} the placeholder. Depending on whether the current task is running on the site where the placeholder was allocated, rules \textit{(determine local)} or \textit{(determine remote)} take care of assigning the value \textit{V} to the placeholder \textit{p}. However, placeholders are not boxes because they are defined as datastructures that can receive one value \textit{at most} \([11]\); placeholders are like single-assignment variables in OC\+ [2] and PCN \([7]\). As opposed to conventional languages, the language \(L\) uses first-class continuations which allow the programmer to write expressions that \textit{“return”} multiple values; in other words, an \(L\) different values can be passed to the same continuation. As a result, we distinguish the case where a placeholder is not assigned, in rules \textit{(determine local)} and \textit{(request del first)}, from the case where it is already assigned, rules \textit{(determine local)} and \textit{(request del last)}. In the former case, the placeholder is updated and the producer task ends. In the latter case, the value is returned to the continuation of future, as if no future had existed, following Katz and Weise’s implementation \([15, 3]\). Let us observe that transitions are atomically executed in order to ensure a sound behaviour of \textit{(determine local)} and \textit{(determine local)}.

Short primitives introduce synchronizations between the consumer task of a placeholder and its producer task; they require their arguments to be proper values, i.e. values different from placeholders; short primitives are said to \textit{touch} their argument. The translation \(\mathcal{T}\) makes the touch operation explicit by the call to \textit{touch}, whose purpose is to return a proper value. We use an auxiliary function \textit{touch}, displayed in Figure 2, which touches a value with respect to a local store \(\theta\) of a site \(s\). The function \textit{touch} can return three results: a proper value, an undetermined placeholder that was allocated on site \(s\), or a placeholder that was allocated on a different site. In the first case, the touch operation succeeds \textit{(touch local)}; in the second case, the task is suspended \textit{(touch suspend)}; in the third case, a request \textit{touch} is sent to the remote site \(\textit{touch remote}\). The remote site behaves similarly: it can return a proper value, suspend the request, or pass it to another site. Tasks or requests that are suspended when touching a placeholder are reactivated when this placeholder gets determined, i.e. \textit{(determine local)} or \textit{(request del first)}. Let us observe that the touch operation can initiate exchanges of messages between sites as soon as a proper value is found, it is directly returned back to the site that started the operation, thanks to the qualified task name.

So far, our explanations have ignored legitimacies. Following Katz and Weise \([15]\), we use a notion of legitimacy to keep track of the control flow that would exist if evaluation was sequential. An initial legitimacy is allocated when we start to evaluate a program, and each new task is given a new legitimacy. Legitimacies, like placeholders, are datastructures whose only slot can receive one value at most unlike placeholders, legitimacies are not first-class values. When a placeholder gets determined, the consumer task becomes dependent on the value of the placeholder; hence, the legitimacy of the producer task, recorded in
continuation (as det ph t), is stored into the legitimacy of the consumer task. As evaluation proceeds, chains of legitimacies get formed into memory. The relation \( t_1 \sim t_2 \) states that there is a path from legitimacy \( t_1 \) to legitimacy \( t_2 \) in the local store \( \theta \) of site \( s \), which means that control has flowed from a task with legitimacy \( t_2 \) to a task with legitimacy \( t_1 \).

As we want future to be an annotation, every program should return the result that it would produce when evaluated sequentially in the absence of future. The solution adopted in our semantics is to perform causally-dependent \([24]\) box accesses in the same order as in a sequential implementation; the solution relies on legitimacies. The translation of the primitive deref, \( \text{deref} \) (sync), touches and then applies sync on its argument.

The primitive sync behaves as the identity function if the legitimacy of the current task leads to the legitimacy associated with the box. In other words, sync acts as a synchronisation barrier by ensuring that all accesses to the box (read or write) that a sequential implementation would have performed before the current access are actually done in the parallel machine, and all accesses that a sequential implementation would perform after the current one remain to be performed by the parallel machine. The primitive sync suspends a task that illegitimately tries to access a box; it will be reactivated by (determine local) or (request det first).

In order to determine when a computation ends, the initial configuration contains a box aimed at receiving the final value. Consistent box accesses guarantee that the box will receive the legitimate final value (if there exists one).

It should be observed that using legitimacies to synchronise box accesses does not impose a total order on those operations, but a partial order. This property ensures that parallelism can exist for programs written in a mostly-functional style, where one generally considers that side-effects are performed locally in different modules or functions.

\[
\forall W_1 \rightarrow^{\text{det}_{\text{det}}} W_2 \text{ if } W_1 \rightarrow_{\text{det}} W_2, \text{ or } W_1 = \{m_1\} \cup W_1, W_2 = \{m_2\} \cup W_2, m_1 \rightarrow m_2, \text{ with } m = 0 \text{ otherwise } \]

\[
\forall W_1 \rightarrow^{\text{det}_{\text{det}}} W_2 \text{ if } W_1 \rightarrow^{\text{det}_{\text{det}}} W_3 \text{ and } W_3 \rightarrow^{\text{det}_{\text{det}}} W_2 \quad \text{(transitive)}
\]

Conventions: \( W_1 \rightarrow_{\text{det}} W_2 \) if \( W_1 \rightarrow^{\text{det}} W_2 \) or \( W_1 \rightarrow^{\text{det}} W_2 \) if \( W_1 \rightarrow^{\text{det}} W_2, n \geq 0 \);

\( W_1 \rightarrow^{\text{det}} W_2 \) if \( W_1 \rightarrow^{\text{det}} W_2, n > 0 \).

Initial world for \( P \): \( \text{InitWorld}[P] = \{m_0, m_1, \ldots, m_n\} \), with

- Initial Store: \( \theta_0 = \{(a_n, \bot)\} \)
- Initial Legitimacy: \( \ell_0 = \{\text{log } a_1, a_0\} \)
- Initial Box: \( b_0 = \{(\text{box } a_0, a_0, \ell_0)\} \)
- Initial Environment: \( \rho_0 = \{(x, b_0)\} \)
- Empty Sites: \( m_i = \{(\text{ex}(\text{det}), \text{deref} (\text{sync} x, v), A[P], x, i)\}, a_0, \theta, (0,0,0,0,0) \) i = 1, ..., n

Initial World: \( \text{Final}[W] \) if \( W_1 \neq \bot \).

\[
\text{eval}_{\text{det}}(P) = \begin{cases} 
W & \text{if there exists } W_0, W_1, \text{ such that } W_0 = \text{InitWorld}[P], W_0 \rightarrow^{\text{det}} W_1, \\
\text{Final}[W] & \text{with } W = \text{Unload}(W_0, \text{det}) \cup W_1 \\
1 & \text{if } W \in \mathbb{N}, 2W_0 \in \text{World}, a_0, \ell_0 \in \mathbb{N}, W_1 \rightarrow^{\text{det}_{\text{det}}} W_2 \text{ such that } m_i > 0 \\
W_0 \rightarrow^{\text{det}_{\text{det}}} W_1 & \text{InitWorld}[P]
\end{cases}
\]

Fig. 8. Evaluation Relation

We now have all the components to define an evaluation relation that associates programs with their observable behaviour. The function \( \text{Unload} \) replaces each function, box, or continuation by a tag in addition; \( \text{Unload} \) touches every placeholder appearing in the result. As values can be spread over different sites, \( \text{Unload} \) takes the world in argument.

Divergence should be defined with the greatest care, because future has the ability to create new tasks, but the scheduler may elect to evaluate any of them. One task only is mandatory; all the others are speculative. A task with legitimacy \( \ell \) is mandatory if \( \ell \) leads
to the initial legibility $l_0$ in the current world $W_0$, which is written $l \simeq_{W_0} l_0$. Figure 8 defines the relation $\simeq_{W_0}$ [6] to denote reductions that involve $n$ steps among which $m$ are mandatory. According to the evaluation relation $eval_{W_0}$ a program is said to be divergent, i.e., its value is $\bot$, if it leads to an infinite transition sequence that regularly often contains mandatory transitions.

The soundness of the CEK-based machine is established by proving that its evaluation relation $eval_{W_0}$ is equal to the sequential evaluation function of the CEK-Machine.

**Theorem 1 (Soundness)** $eval_{W_0} = eval_{ck} \square$

### 3 Discussion and Related Work

This paper builds upon previous work about annotations for parallelism in functional languages. For a long time, research has focused on implementation issues and efficient designs [10, 3, 28, 14, 13, 11, 18]. Parallelism by annotations has been formalised recently only. Flanagan and Fleissel [6] have defined the semantics of future in a purely functional language. The author [29, 20] has proposed a semantic framework for continuations and side-effects in a language with the annotations `call`, `fork`, and `future`. This paper is the first to present a formal semantics for futures, side-effects, first-class continuations, and distribution. Our research is part of a project which aims at building a virtual multiprocessor [23], which provides a soft architecture to support distributed applications, transcending the details of the hardware architecture. The "distribution by annotation" paradigm is our contribution to this virtual multiprocessor, which provides the user with the view that a distributed network of computers is programmable as a sequential processor.

Both compile-time and runtime improvements could boost the performance of our architecture. Our semantics is a dynamic semantics and there are opportunities to improve it using static analysis. Flanagan and Fleissel's [6] touch analysis removes provably redundant touch operations in purely functional future-based programs, using a set-based analysis [12]; extending their analysis to side-effects and first-class continuations would be desirable for the CEK-based machine. Similarly, an analysis removing unnecessary sync operations would greatly reduce the cost of synchronisations associated with side-effects.

As far as the runtime system is concerned, a realistic implementation needs a distributed garbage collector; the approach "garbage collecting the world" [16] appears to be a suitable candidate. Similarly, we have to address the issue of process collection. Miller's MtsScheme [17] task collection is done during garbage collection: a task can be reclaimed if the placeholder that it determines is accessible from the gc roots.

According to Figure 4, every future creates a new task on the current site. This process creation strategy is referred to as eager task creation [11, 18, 3]. However, future-based programs can generate far more tasks than the number of sites in a CEK-based world. In order to avoid the expensive cost of task creation, a lazy task creation [18, 3] strategy can be used: it postpones the creation of a task until a processor is ready to run it. A simple modification of our rules could make this strategy explicit. Though rule (migrate task) does not enforce any migration strategy, we think that task stealing [1, 3] would be appropriate; according to this strategy, a processor that becomes idle steals a task from a heavily loaded processor.

Quine's ICSL [27, 24, 28, 25] is a dialect of Lisp offering primitives for parallelism, transparent migration of objects, and maintenance of their cache coherence over the network. Quine's purpose is different from ours; as he does not rely on transparent annotations, he does not preserve the sequential meaning of programs. However, he proposes a caching mechanism which is certainly lacking in our CEK-based machine. Although his notion of coherence is not suitable to the CEK-based machine because it is not relative to the sequential evaluation, the protocol that he proposes with lazy propagation of updated values is an interesting technique that would be worth investigating for our semantics.

### 4 Conclusion

Traditional approaches to distributed computing favour high-performance over ease of programming. We believe that there is a need for a high-level paradigm to program distributed systems. By supplying transparent annotations to create remote computations, we provide
the programmer with the illusion that a distributed system is programmable like a sequential one, because the runtime system itself takes care of tasks and data migrations, race conditions, or critical sections.

This paper is the first step in this direction; we propose an abstract machine with a distributed store and prove that task migration is transparent to the user. Work is under way to refine the semantics and to precisely investigate the issue of data migration and distributed garbage collection within this framework.

References

This article was processed using the LATEX macro package with LLNCS style