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 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 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. 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 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 makeref to create boxes, with primitives deref, setref! to read and modify them, and with a primitive callcc to capture first-class continuations. In addition, there is a construct (future M) to create a producer-consumer type of parallelism [11]. Intuitively, the evaluation of (future M) immediately returns an object called placeholder, while another task evaluates the argument M in parallel. The purpose of the latter task, called the *producer*, is to compute and then store the value of M in the placeholder. The task using the placeholder is called the *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 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. *i)* We define a distributed architecture able to evaluate future-based programs; *ii)* We prove that

* 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.

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

Fig. 1. Syntax of Λ_f

task migration is transparent to the programmer, i.e. task migration does not change the observable behaviour of programs; *iii)* This result is the reference semantics that we can use to design and prove the correctness of program optimisations. 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 Felleisen and Friedman's CEK and CEKS machines [4, 5], and the F-PCKS-machine [20]; 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*[2], 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 Λ_{cekds}, the language accepted by the CEKDS-machine; besides, we translate every program of Λ_f into a term of Λ_{cekds} by the function $\mathcal{X}$ 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 synchronisation. 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 uniquess 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 $\mathsf{Ev}\langle M, \rho, \kappa\rangle$ representing the evaluation of a term M in an environment ρ with a continuation κ, or $\mathsf{Ret}\langle V, \kappa\rangle$ 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 $\mathsf{Ev}\langle M, \rho, \kappa\rangle$. 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 τ_1, which speculatively evaluates the

[2] A strict function applied to a placeholder, accesses the value that the producer task has stored in the placehoder; this operation, called *touching* the placeholder, can suspend the current task when the placeholder has not received a value yet.

$$
\begin{array}{lll}
\mathcal{W} & ::= \{m_1, \ldots, m_n\} & \text{(World)} \\
m \in \mathcal{M} & ::= \langle T, \theta, s, S, I, O\rangle & \text{(Site)} \\
t \in Task & ::= \langle C, \ell, \tau\rangle & \text{(Task)} \\
\theta \in Store & ::= \{(\alpha_1\ X_1)\ldots(\alpha_n\ X_n)\} & \text{(Store)} \\
X & ::= V \ \mid\ \bot \ \mid\ \ell & \text{(Store Content)} \\
s \in \mathcal{S} & = \{s_1, s_2, \ldots\} & \text{(Site Name)} \\
\tau \in \mathcal{T} & = \{\tau_1, \tau_2, \ldots\} & \text{(Task Name)} \\
u & ::= \langle \tau, s\rangle & \text{(Qualified Task Name)} \\
a & ::= \langle \alpha, s\rangle & \text{(Qualified Location)}
\end{array}
$$

Explicit translation: $\mathcal{X} : \Lambda_f \rightarrow \Lambda_{cekds}$

$$
\begin{array}{rl}
\mathcal{X}[\![x]\!] &= x \text{ if } x \in Vars \cup BConst \cup NStP \\
\mathcal{X}[\![\mathsf{car}]\!] &= \lambda x.(\mathsf{car}\ (\mathsf{touch}\ x)) \\
\mathcal{X}[\![\mathsf{cdr}]\!] &= \lambda x.(\mathsf{cdr}\ (\mathsf{touch}\ x)) \\
\mathcal{X}[\![\mathsf{deref}]\!] &= \lambda x.(\mathsf{deref}\ (\mathsf{sync}\ (\mathsf{touch}\ x))) \\
\mathcal{X}[\![\mathsf{setref!}]\!] &= \lambda x_1 x_2.(\mathsf{setref!}\ (\mathsf{sync}\ (\mathsf{touch}\ x_1))\ x_2) \\
\mathcal{X}[\![(\mathsf{future}\ M)]\!] &= (\mathsf{future}\ \mathcal{X}[\![M]\!]) \\
\mathcal{X}[\![(M_1\ M_2)]\!] &= (\lambda m_1 m_2.(\mathsf{touch}\ m_1) m_2)\mathcal{X}[\![M_1]\!]\mathcal{X}[\![M_2]\!] \\
\mathcal{X}[\![(\lambda x.M)]\!] &= (\lambda x.\mathcal{X}[\![M]\!]) \\
\mathcal{X}[\![(\mathsf{if}\ M_1\ M_2\ M_3)]\!] &= (\mathsf{if}\ (\mathsf{touch}\ \mathcal{X}[\![M_1]\!])\ \mathcal{X}[\![M_2]\!]\ \mathcal{X}[\![M_3]\!])
\end{array}
$$

$$
\begin{array}{lll}
C \in CoSt & ::= \mathsf{Ev}\langle M, \rho, \kappa\rangle & \text{(Computational State)} \\
 & \mid\ \mathsf{Ret}\langle V, \kappa\rangle & \\
\rho \in Env & ::= \{(x_1\ V_1)\ \ldots(x_n V_n)\} & \text{(Environment)} \\
I & = \{q_1, \ldots, q_n\} & \text{(Input Spool)} \\
O & = \{q_1, \ldots, q_n\} & \text{(Output Spool)} \\
q & ::= R \ \mid\ A & \text{(Message)} \\
T & = \{t_1, \ldots, t_n\} & \text{(Active Tasks Set)} \\
S & = \{t_1, \ldots, t_n\} & \text{(Suspend Set)} \\
 & \cup\ \{R_1, \ldots, R_m\} &
\end{array}
$$

$$
\begin{array}{rl}
Unload[c, \mathcal{W}] &= c \\
Unload[(\mathsf{cons}\ V_1\ V_2), \mathcal{W}] &= (\mathsf{cons}\ Unload[V_1, \mathcal{W}] \\
 & \qquad Unload[V_2, \mathcal{W}]) \\
Unload[\lambda x.M, \mathcal{W}] &= \textbf{procedure} \\
Unload[b, \mathcal{W}] &= \textbf{box} \\
Unload[f_c, \mathcal{W}] &= \textbf{procedure} \\
Unload[\langle \mathsf{co}\ \kappa\rangle, \mathcal{W}] &= \textbf{cont} \\
Unload[\langle \mathsf{ph}\ \alpha, s\rangle, \mathcal{W}] &= Unload[\mathcal{W}[\langle \alpha, s\rangle], \mathcal{W}]
\end{array}
$$

$$
\begin{array}{lll}
M \in \Lambda_{cekds} & ::= V_s \ \mid\ (M\ M) & \text{(Term)} \\
 & \mid\ (\mathsf{if}\ M\ M\ M) \ \mid\ (\mathsf{future}\ M) & \\
V_s \in SValue & ::= c \ \mid\ x \ \mid\ (\lambda x.M) & \text{(Syntactic Value)} \\
W \in PValue & ::= c \ \mid\ \langle \mathsf{cl}\ \lambda x.M, \rho\rangle \ \mid\ f_c & \text{(Proper Value)} \\
 & \mid\ (\mathsf{cons}\ V\ V) \ \mid\ \langle \mathsf{co}\ \kappa\rangle \ \mid\ b & \\
V \in Value & ::= W \ \mid\ ph & \text{(Runtime Value)} \\
ph & ::= \langle \mathsf{ph}\ \alpha, s\rangle & \text{(Placeholder)} \\
b \in Box & ::= \langle \mathsf{bx}\ \alpha, s, \ell\rangle & \text{(Box)} \\
\ell \in Leg & ::= \langle \mathsf{leg}\ \alpha, s\rangle & \text{(Legitimacy)} \\
c \in Const & ::= p \ \mid\ f & \text{(Constant)} \\
f_c \in PApp & ::= (\mathsf{cons}\ V) \ \mid\ (\mathsf{setref!}\ V) & \text{(Partial Application)} \\
g \in AValue & ::= \langle \mathsf{cl}\ \lambda x.M, \rho\rangle \ \mid\ f \ \mid\ f_c \ \mid \langle \mathsf{co}\ \kappa\rangle & \text{(Applicable Val.)} \\
p \in BConst & = \{\mathsf{true}, \mathsf{false}, \mathsf{nil}, 0, 1, \ldots, \mathsf{void}\} & \text{(Basic Constant)} \\
f \in FConst & = \{\mathsf{cons}, \mathsf{car}, \mathsf{cdr}, \mathsf{makeref}, \mathsf{deref}, & \text{(Func. Cstnt)} \\
 & \quad \mathsf{setref!}, \mathsf{callcc}, \mathsf{sync}, \mathsf{touch}\} & \\
f_n \in NStP & = \{\mathsf{cons}, \mathsf{makeref}, \mathsf{callcc}\} & \text{(Non Strict Primitives)} \\
x \in Vars & = \{x, y, z \ldots\} & \text{(User Variable)} \\
\kappa \in CCode & ::= (\mathbf{init}) \ \mid\ (\kappa\ \mathbf{fun}\ V) & \text{(Continuation code)} \\
 & \mid\ (\kappa\ \mathbf{arg}\ M\ \rho) \ \mid\ (\kappa\ \mathbf{cond}(M, M, \rho)) & \\
 & \mid\ (\kappa\ \mathbf{det}\ (ph, \ell)) \ \mid\ (\kappa\ \mathsf{leg}\ (\ell, \ell)) & \\
R \in Req & ::= \mathsf{Req}\langle s, u, rc\rangle & \text{(Request)} \\
rc \in RC & ::= \mathsf{rtouch}(\alpha) \ \mid\ \mathsf{rdet}(\alpha, V, \alpha, \ell) & \text{(Request Contents)} \\
 & \mid\ \mathsf{deref}(\alpha) \ \mid\ \mathsf{rset}(\alpha, V) \ \mid\ \mathsf{rleg}(\ell, \ell) & \\
A \in Ans & ::= \mathsf{Ans}\langle u, ac\rangle & \text{(Answer)} \\
ac \in AC & ::= \mathsf{rtouch}(V) \ \mid\ \mathsf{rdet}(X) & \text{(Answer Contents)} \\
 & \mid\ \mathsf{rset}(X) \ \mid\ \mathsf{deref}(V) \ \mid\ \mathsf{rleg} &
\end{array}
$$

Free Task Name:

$$FN(T) = \{\tau, \langle C, \ell, \tau\rangle \in T\}$$

Store Operations:

$$
\begin{array}{l}
\theta \uplus \{(\alpha\ V)\} = \theta \cup\ \{(\alpha\ V)\} \\
\qquad \text{with } \alpha \notin DOM(\theta) \\
\theta(\alpha) = V \text{ if } (\alpha\ V) \in \theta \\
\theta[\alpha := V] = (\theta \setminus \{(\alpha\ \theta(\alpha))\}) \\
\qquad \cup \{(\alpha\ V)\}
\end{array}
$$

Global reference:

$$
\begin{array}{l}
\mathcal{W}[\langle \alpha, s\rangle] = \theta(s) \\
\qquad \text{if } \exists \langle T, s, \theta, S, I, O\rangle \in \mathcal{W}
\end{array}
$$

Environment Operations:

$$
\begin{array}{l}
\rho(x) = V \text{ if } (x\ V) \in \rho \\
\rho[x \leftarrow V] = (\rho \setminus \{(x\ V')\}) \\
\qquad \cup \{(x\ V)\} \\
\qquad \text{if } (x\ V') \in \rho \\
\rho[x \leftarrow V] = \rho \cup \{(x\ V)\} \\
\qquad \text{if } x \notin DOM(\rho)
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{ltouch}(W, \theta, s) &= W \\
\mathsf{ltouch}(\langle \mathsf{ph}\ \alpha, s\rangle, \theta, s) &= \mathsf{ltouch}(\theta(\alpha), \theta, s) \text{ if } \theta(\alpha) \neq \bot \\
\mathsf{ltouch}(\langle \mathsf{ph}\ \alpha, s\rangle, \theta, s) &= \langle \mathsf{ph}\ \alpha, s\rangle \text{ if } \theta(\alpha) = \bot \\
\mathsf{ltouch}(\langle \mathsf{ph}\ \alpha, s_1\rangle, \theta, s) &= \langle \mathsf{ph}\ \alpha, s_1\rangle \text{ if } s_1 \neq s
\end{array}
$$

$$
\begin{array}{l}
\ell \rightsquigarrow^s_\theta \ell \\
\langle \mathsf{leg}\ \alpha, s\rangle \rightsquigarrow^s_\theta \ell \text{ if } \theta(\alpha) \neq \bot \text{ and } \theta(\alpha) \rightsquigarrow^s_\theta \ell \\
\ell \rightsquigarrow_{\mathcal{W}} \ell \\
\langle \mathsf{leg}\ \alpha, s_1\rangle \rightsquigarrow_{\mathcal{W}} \ell \text{ if } \theta_{s_1}(\alpha) \neq \bot \\
\qquad \text{and } \mathcal{W}[\langle \alpha, s_1\rangle] \rightsquigarrow_{\mathcal{W}} \ell
\end{array}
$$

Fig. 2. State Space of the CEKDS-machine

$$
\begin{array}{rll}
\mathsf{Ev}\langle (M\ N), \rho, \kappa\rangle & \rightarrow_{cek} \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{arg}\ N, \rho)\rangle & (\mathit{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 & (\mathit{lambda})\\
\mathsf{Ev}\langle c, \rho, \kappa\rangle & \rightarrow_{cek} \mathsf{Ret}\langle c, \kappa\rangle & (\mathit{constant})\\
\mathsf{Ev}\langle x, \rho, \kappa\rangle & \rightarrow_{cek} \mathsf{Ret}\langle \rho(x), \kappa\rangle & (\mathit{variable})\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ N, \rho)\rangle & \rightarrow_{cek} \mathsf{Ev}\langle N, \rho, (\kappa\ \mathbf{fun}\ V)\rangle & (\mathit{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 & (\mathit{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 & (\mathit{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} & (\mathit{if\ else})\\
 & \rightarrow_{cek} \mathsf{Ev}\langle M_1, \rho, \kappa\rangle \quad \text{if } V \neq \mathsf{false} & (\mathit{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 & (\mathit{capture})\\
\mathsf{Ret}\langle V, (\kappa'\ \mathbf{fun}\ \langle \mathsf{co}\ \kappa\rangle)\rangle & \rightarrow_{cek} \mathsf{Ret}\langle V, \kappa\rangle & (\mathit{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 \in PApp & (\mathit{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 & (\mathit{cons})\\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \mathbf{fun}\ \mathsf{car})\rangle & \rightarrow_{cek} \mathsf{Ret}\langle V_1, \kappa\rangle & (\mathit{car})\\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \mathbf{fun}\ \mathsf{cdr})\rangle & \rightarrow_{cek} \mathsf{Ret}\langle V_2, \kappa\rangle & (\mathit{cdr})\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ f)\rangle & \rightarrow_{cek} \mathsf{Ret}\langle \delta(f, V), \kappa\rangle & (\delta)\\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ V_1)\rangle & \rightarrow_{cek} \mathsf{Ret}\langle \mathsf{error}, (\mathbf{init})\rangle \text{ if } V_1 \notin AValue & (\mathit{apply\ error})
\end{array}
$$

Fig. 3. Transitions between computational states

continuation of future with the placeholder ph. After transition, the initial task τ evaluates the argument of future. We shall explain later the purpose of the legitimacies ℓ and ℓ_1.

A new box can be created by applying the functional constant 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 (*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 (*deref remote*) adds to the output spool a *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 deref(α) means that the distant site is asked to supply the content of location α.

Sites communicate according to the rules of Figure 5. In rule (*migrate request*), two sites exchange 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 (*request deref*), the content of the location is packaged 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 (*migrate answer*). Figure 7 shows the rules that handle incoming answers. The arrival of the answer 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 (*fork*) allows us to create new tasks on the current site. In Figure 5, rule (*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.

$$\langle \{\ \langle C, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle C_1, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle \quad \text{if } C \rightarrow_{cek} C_1 \qquad (\textit{sequential})$$

$$\langle \{\ \langle \mathsf{Ev}\langle (\mathsf{future}\ M), \rho, \kappa\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{det}\ ph, \ell_1)\rangle, \ell, \tau\rangle,\ \ \langle \mathsf{Ret}\langle ph, \kappa\rangle, \ell_1, \tau_1\rangle\ \} \ \cup\ T, \theta_1, s, S, I, O\rangle \qquad (\textit{fork})$$

$$\quad\quad \text{with } ph = \langle \mathsf{ph}\ \alpha, s\rangle, \ell_1 = \langle \mathsf{leg}\ \alpha_1, s\rangle, \theta_1 = \theta\ \uplus\ \{\ (\alpha\ \bot)\ \ (\alpha_1\ \bot)\ \}, \tau_1 \notin FN(T \cup S) \cup \{\tau\}$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ \mathsf{makeref})\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle b, \kappa\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta_1, s, S, I, O\rangle \text{ with } b = \langle \mathsf{bx}\ \alpha, s, \ell\rangle, \theta_1 = \theta \uplus \{(\alpha\ V)\} \qquad (\textit{makeref})$$

$$\langle \{\ \langle \mathsf{Ret}\langle \langle \mathsf{bx}\ \alpha, s, \ell\rangle, (\kappa\ \mathbf{fun}\ \mathsf{deref})\rangle, \ell_1, \tau\rangle\ \} \ \cup\ T, \theta, s_1, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle \theta(\alpha), \kappa\rangle, \ell_1, \tau\rangle\ \} \ \cup\ T, \theta, s_1, S, I, O\rangle \quad \text{if } s_1 = s \qquad (\textit{deref local})$$

$$\quad \rightarrow_s \langle T, \theta, s_1, \{\ \langle \mathsf{Ret}\langle \langle \mathsf{bx}\ \alpha, s, \ell\rangle, (\kappa\ \mathbf{fun}\ \mathsf{deref})\rangle, \ell_1, \tau\rangle\ \} \ \cup\ S, I, O_1\rangle \quad \text{if } s_1 \neq s \qquad (\textit{deref remote})$$

$$\quad\quad \text{with } O_1 = \{\ \mathsf{Req}\langle s, \langle \tau, s_1\rangle, \mathsf{deref}(\alpha)\rangle\ \} \ \cup\ O$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ (\mathsf{setref}\ \langle \mathsf{bx}\ \alpha, s, \ell\rangle))\rangle, \ell_1, \tau\rangle\ \} \ \cup\ T, \theta, s_1, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \ell_1, \tau\rangle\ \} \ \cup\ T, \theta[\alpha := V], s_1, S, I, O\rangle \quad \text{if } s_1 = s \qquad (\textit{setref local})$$

$$\quad \rightarrow_s \langle T, \theta, s_1, \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ (\mathsf{setref}\ \langle \mathsf{bx}\ \alpha, s, \ell\rangle))\rangle, \ell_1, \tau\rangle\ \} \ \cup\ S, I, O_1\rangle \qquad (\textit{setref remote})$$

$$\quad\quad \text{if } s_1 \neq s, \text{ with } O_1 = \{\ \mathsf{Req}\langle s, \langle \tau, s_1\rangle, \mathsf{rset}(\alpha, V)\rangle\ \} \ \cup\ O$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{det}\ \langle \mathsf{ph}\ \alpha, s\rangle, \langle \mathsf{leg}\ \alpha_1, s\rangle)\rangle, \ell_2, \tau\rangle\ \} \ \cup\ T, \theta, s_2, S, I, O\rangle$$

$$\quad \rightarrow_s \langle T_1, \theta_1, s_2, S_1, I_1, O\rangle \quad \text{if } s_2 = s,\ \theta(\alpha) {=} \bot \qquad (\textit{determine local})$$

$$\quad\quad \text{with } I_1 = (I \cup I_2), T_1 = (T \cup T_2), S_1 = (S \setminus (I_2 \cup T_2)), \theta_1 = \theta[\alpha_1 := \ell_2][\alpha := V]$$

$$\quad\quad \text{with } I_2 = \{\mathsf{Req}\langle s_3, \langle \tau_1, s_4\rangle, \mathsf{rtouch}(\alpha)\rangle, \mathsf{Req}\langle s_3, \langle \tau_1, s_4\rangle, \mathsf{rleg}(\langle \mathsf{leg}\ \alpha_1, s\rangle, \ell_4)\rangle \in S\}$$

$$\quad\quad \text{with } T_2 = \{\langle \mathsf{Ret}\langle \langle \mathsf{ph}\ \alpha, s\rangle, (\kappa'\ \mathbf{fun}\ \mathsf{touch})\rangle, \ell_3, \tau_1\rangle\ \in\ S\}\ \ \cup$$

$$\quad\quad\quad\quad \{\langle \mathsf{Ret}\langle V_1, (\kappa'\ \mathsf{leg}\ (\langle \mathsf{leg}\ \alpha_1, s\rangle, \ell_4))\rangle, \ell_5, \tau_1\rangle \in S\}$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle V, \kappa\rangle, \ell_2, \tau\rangle\ \} \ \cup\ T, \theta, s_2, S, I, O\rangle \quad \text{if } s_2 = s,\ \theta(\alpha) {\neq} \bot \qquad (\textit{determine localn})$$

$$\quad \rightarrow_s \langle T, \theta, s_2, \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{det}\ \langle \mathsf{ph}\ \alpha, s\rangle, \langle \mathsf{leg}\ \alpha_1, s\rangle)\rangle, \ell_2, \tau\rangle\ \} \ \cup\ S, I, O_1\rangle \qquad (\textit{determine remote})$$

$$\quad\quad \text{if } s_2 \neq s \text{ with } O_1 = \{\ \mathsf{Req}\langle s, \langle \tau, s_2\rangle, \mathsf{rdet}(\alpha, V, \alpha_1, \ell_2)\rangle\ \} \ \cup\ O$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle \mathsf{ltouch}(V, \theta, s), \kappa\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle \qquad (\textit{touch local})$$

$$\quad\quad \text{if } \mathsf{ltouch}(V, \theta, s) \in PValue$$

$$\quad \rightarrow_s \langle T, \theta, s, \{\ \langle \mathsf{Ret}\langle \langle \mathsf{ph}\ \alpha, s_1\rangle, (\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle, \ell, \tau\rangle\ \} \ \cup\ S, I, O_1\rangle \qquad (\textit{touch remote})$$

$$\quad\quad \text{if } \mathsf{ltouch}(V, \theta, s) = \langle \mathsf{ph}\ \alpha, s_1\rangle,\ s \neq s_1,\ \text{with } O_1 = \{\mathsf{Req}\langle s_1, \langle \tau, s\rangle, \mathsf{rtouch}(\alpha)\rangle\} \cup O$$

$$\quad \rightarrow_s \langle T, \theta, s, \{\ \langle \mathsf{Ret}\langle \langle \mathsf{ph}\ \alpha, s_1\rangle, (\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle, \ell, \tau\rangle\ \} \ \cup\ S, I, O\rangle \qquad (\textit{touch suspend})$$

$$\quad\quad \text{if } \mathsf{ltouch}(V, \theta, s) = \langle \mathsf{ph}\ \alpha, s_1\rangle,\ s = s_1,\ \theta(\alpha) {=} \bot$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ \mathsf{sync})\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathsf{leg}\ (\ell, \ell_1))\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle \quad \text{if } V = \langle \mathsf{bx}\ \alpha, s_1, \ell_1\rangle \qquad (\textit{synchronise})$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle \mathsf{error}, (\mathbf{init})\rangle, \ell, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle \quad \text{if } V \notin Box \qquad (\textit{synchronise error})$$

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathsf{leg}\ (\ell, \ell_1))\rangle, \ell_2, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle$$

$$\quad \rightarrow_s \langle \{\ \langle \mathsf{Ret}\langle V, \kappa\rangle, \ell_2, \tau\rangle\ \} \ \cup\ T, \theta, s, S, I, O\rangle \quad \text{if } \ell \leadsto^s_\theta \ell_1 \qquad (\textit{leg local})$$

$$\quad \rightarrow_s \langle T, \theta, s, \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathsf{leg}\ (\langle \mathsf{leg}\ \alpha, s_1\rangle, \ell_1))\rangle, \ell_2, \tau\rangle\ \} \ \cup\ S, I, O_1\rangle \qquad (\textit{leg remote})$$

$$\quad\quad \text{if } \ell \leadsto^s_\theta \langle \mathsf{leg}\ \alpha, s_1\rangle,\ s_1 \neq s,\ \text{with } O_1 = \{\ \mathsf{Req}\langle s_1, \langle \tau, s\rangle, \mathsf{rleg}(\langle \mathsf{leg}\ \alpha, s_1\rangle, \ell_1)\rangle\ \} \ \cup\ O$$

$$\quad \rightarrow_s \langle T, \theta, s, \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathsf{leg}\ (\langle \mathsf{leg}\ \alpha, s\rangle, \ell_1))\rangle, \ell_2, \tau\rangle\ \} \ \cup\ S, I, O\rangle \qquad (\textit{leg suspend})$$

$$\quad\quad \text{if } \ell \leadsto^s_\theta \langle \mathsf{leg}\ \alpha, s\rangle,\ \ell \not\leadsto^s_\theta \ell_1,\ \theta(\alpha) {=} \bot$$

Fig. 4. Site Transitions

$$\{\langle\{\ \langle C_1,\ell_1,\tau_1\rangle\ \}\ \cup\ T_1,\theta,s_1,S_1,I_1,O_1\rangle,\langle\emptyset,\theta_2,s_2,S_2,I_2,O_2\rangle\}\ \cup\ \mathcal{W} \qquad (\textit{migrate task})$$
$$\rightarrow_c \{\langle T_1,\theta,s_1,S_1,I_1,O_1\rangle,\langle\{\ \langle C_1,\ell_1,\tau_2\rangle\ \},\theta_2,s_2,S_2,I_2,O_2\rangle\}\ \cup\ \mathcal{W}$$
$$\text{if } T_1 \neq \emptyset, \text{ with } \tau_2 \notin FN(S_2)$$
$$\{\langle T_1,\theta_1,s_1,S_1,I_1,\{\ \mathsf{Req}\langle s_2,\langle\tau_3,s_3\rangle,rc\rangle\ \}\ \cup\ O_1\rangle,\langle T_2,\theta_2,s_2,S_2,I_2,O_2\rangle\}\ \cup\ \mathcal{W} \qquad (\textit{migrate request})$$
$$\rightarrow_c \{\langle T_1,\theta_1,s_1,S_1,I_1,O_1\rangle,\langle T_2,\theta_2,s_2,S_2,\{\ \mathsf{Req}\langle s_2,\langle\tau_3,s_3\rangle,rc\rangle\ \}\ \cup\ I_2,O_2\rangle\}\ \cup\ \mathcal{W}$$
$$\{\langle T_1,\theta_1,s_1,S_1,I_1,\{\ \mathsf{Ans}\langle\tau_2,s_2,ac\rangle\ \}\ \cup\ O_1\rangle,\langle T_2,\theta_2,s_2,S_2,I_2,O_2\rangle\}\ \cup\ \mathcal{W} \qquad (\textit{migrate answer})$$
$$\rightarrow_c \{\langle T_1,\theta_1,s_1,S_1,I_1,O_1\rangle,\langle T_2,\theta_2,s_2,S_2,\{\ \mathsf{Ans}\langle\tau_2,s_2,ac\rangle\ \}\ \cup\ I_2,O_2\rangle\}\ \cup\ \mathcal{W}$$

Fig. 5. Communications between sites

$$\langle T,\theta,s,S,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{deref}(\alpha)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{deref}(\theta(\alpha))\rangle\ \}\ \cup\ O\rangle \qquad (\textit{request deref})$$
$$\langle T,\theta,s,S,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rset}(\alpha,V)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T,\theta_1,s,S,I,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rset}\rangle\ \}\ \cup\ O\rangle \text{ with } \theta_1=\theta[\alpha:=V] \qquad (\textit{request set})$$
$$\langle T,\theta,s,S,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rdet}(\alpha,V,\alpha_1,\ell)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T_1,\theta_1,s,S_1,I_1,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rdet}(\theta(\alpha))\rangle\ \}\ \cup\ O\rangle \text{ if } \theta(\alpha)=\perp \qquad (\textit{request det first})$$
$$\text{with } I_1=(I\ \cup\ I_2),T_1=(T\cup T_2),S_1=(S\setminus(I_2\ \cup\ T_2)),\theta_1=\theta[\alpha:=V][\alpha_1:=\ell]$$
$$\text{with } I_2=\{\mathsf{Req}\langle s_3,\langle\tau_1,s_4\rangle,\mathsf{rtouch}(\alpha)\rangle,\mathsf{Req}\langle s_3,\langle\tau_1,s_4\rangle,\mathsf{rleg}(\langle\mathsf{leg}\ \alpha_1,s\rangle,\ell_1)\rangle\in S\}$$
$$\text{with } T_2=\{\langle\mathsf{Ret}\langle\langle\mathsf{ph}\ \alpha,s\rangle,(\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle,\ell_1,\tau_1\rangle\in S\}\ \cup$$
$$\{\langle\mathsf{Ret}\langle V_2,(\kappa\ \mathsf{leg}\ (\langle\mathsf{leg}\ \alpha_1,s\rangle,\ell_1))\rangle,\ell_2,\tau_1\rangle\in S\}$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rdet}(\theta(\alpha))\rangle\ \}\ \cup\ O\rangle \text{ if } \theta(\alpha)\neq\perp \qquad (\textit{request det mult})$$
$$\langle T,\theta,s,S,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rtouch}(\alpha)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rtouch}(\mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s))\rangle\ \}\ \cup\ O\rangle \qquad (\textit{request touch local})$$
$$\text{if } \mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s)\in PValue, s_2\neq s$$
$$\rightarrow_s \langle T,\theta,s,S,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rtouch}(\mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s))\rangle\ \}\ \cup\ I,O\rangle \qquad (\textit{request touch local}')$$
$$\text{if } \mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s)\in PValue,\ s_2=s$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Req}\langle s_3,\langle\tau,s_2\rangle,\mathsf{rtouch}(\alpha_1)\rangle\ \}\ \cup\ O\rangle \qquad (\textit{request touch remote})$$
$$\text{if } \mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s)=\langle\mathsf{ph}\ \alpha_1,s_3\rangle,\ s\neq s_3$$
$$\rightarrow_s \langle T,\theta,s,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rtouch}(\alpha_1)\rangle\ \}\ \cup\ S,I,O\rangle \qquad (\textit{request touch suspend})$$
$$\text{if } \mathsf{ltouch}(\langle\mathsf{ph}\ \alpha,s\rangle,\theta,s)=\langle\mathsf{ph}\ \alpha_1,s\rangle,\ \theta(\alpha_1)=\perp$$
$$\langle T,\theta,s,S,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rleg}(\ell,\ell_1)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rleg}\rangle\ \}\ \cup\ O\rangle \text{ if } \ell\leadsto^s_\theta\ell_1,s_2\neq s \qquad (\textit{request leg local})$$
$$\rightarrow_s \langle T,\theta,s,S,\{\ \mathsf{Ans}\langle\tau,s_2,\mathsf{rleg}\rangle\ \}\ \cup\ I,O\rangle \text{ if } \ell\leadsto^s_\theta\ell_1,s_2=s \qquad (\textit{request leg local}')$$
$$\rightarrow_s \langle T,\theta,s,S,I,\{\ \mathsf{Req}\langle s_3,\langle\tau,s_2\rangle,\mathsf{rleg}(\langle\mathsf{leg}\ \alpha,s_3\rangle,\ell_1)\rangle\ \}\ \cup\ O\rangle \qquad (\textit{request leg remote})$$
$$\text{if } \ell\leadsto^s_\theta\langle\mathsf{leg}\ \alpha,s_3\rangle,\ s\neq s_3$$
$$\rightarrow_s \langle T,\theta,s,\{\ \mathsf{Req}\langle s,\langle\tau,s_2\rangle,\mathsf{rleg}(\langle\mathsf{leg}\ \alpha,s_3\rangle,\ell_1)\rangle\ \}\ \cup\ S,I,O\rangle \qquad (\textit{request leg suspend})$$
$$\text{if } \ell\leadsto^s_\theta\langle\mathsf{leg}\ \alpha,s\rangle,\ \theta(\alpha)=\perp,\ \ell\not\leadsto^s_\theta\ell_1$$

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 (κ **det** ph,ℓ); the producer task is then expected to store V

$$\langle T,\theta,s,\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ \mathsf{deref})\rangle,\ell,\tau\rangle\ \}\ \cup\ S,\{\ \mathsf{Ans}\langle \tau,s,\mathsf{deref}(V_1)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle\{\ \langle \mathsf{Ret}\langle V_1,\kappa\rangle,\ell,\tau\rangle\ \}\ \cup\ T,\theta,s,S,I,O\rangle \qquad (\textit{answer deref})$$
$$\langle T,\theta,s,\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathbf{fun}\ (\mathsf{setref}\ b))\rangle,\ell,\tau\rangle\ \}\ \cup\ S,\{\ \mathsf{Ans}\langle \tau,s,\mathsf{rset}\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle\{\ \langle \mathsf{Ret}\langle \mathsf{void},\kappa\rangle,\ell,\tau\rangle\ \}\ \cup\ T,\theta,s,S,I,O\rangle \qquad (\textit{answer rset})$$
$$\langle T,\theta,s,\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathbf{det}\ ph,\ell)\rangle,\ell_1,\tau\rangle\ \}\ \cup\ S,\{\ \mathsf{Ans}\langle \tau,s,\mathsf{rdet}(X)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle T,\theta,s,S,I,O\rangle \quad \text{if} \quad X=\perp \qquad (\textit{answer det first})$$
$$\rightarrow_s \langle\{\ \langle \mathsf{Ret}\langle V,\kappa\rangle,\ell_1,\tau\rangle\ \}\ \cup\ T,\theta,s,S,I,O\rangle \quad \text{if} \quad X\neq\perp \qquad (\textit{answer det mult})$$
$$\langle T,\theta,s,\{\ \langle \mathsf{Ret}\langle ph,(\kappa\ \mathbf{fun}\ \mathsf{touch})\rangle,\ell,\tau\rangle\ \}\ \cup\ S,\{\ \mathsf{Ans}\langle \tau,s,\mathsf{rtouch}(V)\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle\{\ \langle \mathsf{Ret}\langle V,\kappa\rangle,\ell,\tau\rangle\ \}\ \cup\ T,\theta,s,S,I,O\rangle \qquad (\textit{answer touch})$$
$$\langle T,\theta,s,\{\ \langle \mathsf{Ret}\langle V,(\kappa\ \mathsf{leg}\ (\ell,\ell_1))\rangle,\ell_2,\tau\rangle\ \}\ \cup\ S,\{\ \mathsf{Ans}\langle \tau,s,\mathsf{rleg}\rangle\ \}\ \cup\ I,O\rangle$$
$$\rightarrow_s \langle\{\ \langle \mathsf{Ret}\langle \mathsf{V},\kappa\rangle,\ell_2,\tau\rangle\ \}\ \cup\ T,\theta,s,S,I,O\rangle \qquad (\textit{answer leg})$$

Fig. 7. Handling of Answers

in the placeholder ph; this operation is called *determining* the placeholder. Depending on whether the current task is running on the site where the placeholder was allocated, rules (*determine local*) or (*determine remote*) take care of assigning the value V to the placeholder ph. However, placeholders are not boxes because they are defined as datastructures that can receive one value *at most* [11]; placeholders are like single-assignment variables in CC+ [2] and PCN [7]. As opposed to conventional languages, the language Λ_f has first-class continuations which allow the programmer to write expressions that "return" multiple values; in other words, in Λ_f, different values can be passed to the same continuation. As a result, we distinguish the case where a placeholder is not assigned, in rules (*determine local*) and (*request det first*), from the case where it is already assigned, rules (*determine localn*) and (*request det mult*). In the former case, the placeholder is updated and the producer task ends its evaluation. 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, 20]. Let us observe that transitions are atomically executed in order to ensure a sound behaviour of (*determine local*) and (*determine localn*).

Strict primitives introduce synchronisations 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; strict primitives are said to *touch* their argument. The translation $\mathcal{X}$ makes the touch action explicit by the call to touch, whose purpose is to return a *proper value*. We use an auxiliary function ltouch, displayed in Figure 2, which touches a value with respect to a local store θ of a site s. The function ltouch 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 (*touch local*); in the second case, the task is suspended as long as the placeholder remains undetermined (*touch suspend*); in the third case, a request $\mathsf{rtouch}(\alpha)$ is sent to the remote site (*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, cfr. (*determine local*) or (*request det 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 the continuation (κ **det** *ph* ℓ), is stored into the legitimacy of the consumer task. As evaluation proceeds, chains of legitimacies get formed into memory. The relation $\ell_1 \leadsto^s_\theta \ell_2$ states that there is a path from legitimacy ℓ_1 to legitimacy ℓ_2 in the local store θ of site s, which means that control has flowed from a task with legitimacy ℓ_2 to a task with legitimacy ℓ_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, $\lambda x.(\mathsf{deref}\ (\mathsf{sync}\ (\mathsf{touch}\ x)))$, 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.

$\mathcal{W}_1 \to^{1,m}_{ds} \mathcal{W}_2$ if $\quad \mathcal{W}_1 \to_c \mathcal{W}_2$, or $\mathcal{W}_1 = \{m_1\} \cup \mathcal{W}$, $\mathcal{W}_2 = \{m_2\} \cup \mathcal{W}$, $m_1 \to_s m_2$,
with $m = 1$ if $\ell \leadsto_{\mathcal{W}_1} \ell_0$, with ℓ the legitimacy of the task related to the transition
$m = 0$ otherwise

$$\mathcal{W}_1 \to^{n+n',m+m'}_{ds} \mathcal{W}_2 \text{ if } \mathcal{W}_1 \to^{n,m}_{ds} \mathcal{W}_3 \text{ and } \mathcal{W}_3 \to^{n',m'}_{ds} \mathcal{W}_2 \qquad \text{(transitive)}$$

Conventions: $\mathcal{W}_1 \to_{ds} \mathcal{W}_2$ if $\mathcal{W}_1 \to^{1,m}_{ds} \mathcal{W}_2$; $\mathcal{W}_1 \to^{*}_{ds} \mathcal{W}_2$ if $\mathcal{W}_1 \to^{n,m}_{ds} \mathcal{W}_2, n \geq 0$;
$\mathcal{W}_1 \to^{+}_{ds} \mathcal{W}_2$ if $\mathcal{W}_1 \to^{n,m}_{ds} \mathcal{W}_2, n > 0$.

Initial world for P, $InitWorld[P] = \{m_0, m_1, \ldots, m_n\}$, with

Initial Store: $\theta_0 = \{(\alpha_0\ \bot)\ (\alpha_1\ \bot)\}$ Initial Legitimacy: $\ell_0 = \langle \mathsf{leg}\ \alpha_1, s_0\rangle$
Initial Box: $b_0 = \langle \mathsf{bx}\ \alpha_0, s_0, \ell_0\rangle$ Initial Environment: $\rho_0 = \{(x\ b_0)\}$

$$m_0 = \langle\{\langle \mathsf{Ev}\langle((\lambda v.(\mathsf{setref!}\ (\mathsf{sync}\ x)\ v))\ \mathcal{X}[\![P]\!]), \rho_0, (\mathbf{init})\rangle, \ell_0, \tau_0\rangle\}, s_0, \theta_0, \emptyset, \emptyset, \emptyset\rangle$$

Empty Sites: $m_i = \langle \emptyset, s_i, \emptyset, \emptyset, \emptyset, \emptyset\rangle \quad i = 1, \ldots, n$

Final World: $Final[\mathcal{W}_f]$ if $\mathcal{W}_f[\langle \alpha_0, s_0\rangle] \neq \bot$.

$$\mathrm{eval}_{ds}(P) = \begin{cases} W & \text{if there exists } \mathcal{W}_0, \mathcal{W}_f \text{, such that } \mathcal{W}_0 = InitWorld[P],\ \mathcal{W}_0 \to^{*}_{ds} \mathcal{W}_f, \\ & Final[\mathcal{W}_f] \text{ with } W = Unload(\mathcal{W}_f[\langle \alpha_0, s_0\rangle], \mathcal{W}_f) \\ \bot & \text{if } \forall i \in \mathbf{N}, \exists \mathcal{W}_i \in World, n_i, m_i \in \mathbf{N}, \mathcal{W}_i \to^{n_i,m_i}_{ds} \mathcal{W}_{i+1} \text{ such that } m_i > 0 \\ & \mathcal{W}_0 = InitWorld[P] \end{cases}$$

Fig. 8. Evaluation Relation

We now have all the components to define an evaluation relation that associates programs with their observable behaviour. The function *Unload* replaces each function, box, or continuation by a tag; in addition, *Unload* touches every placeholder appearing in the result. As values can be spread over different sites, *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 ℓ is mandatory if ℓ leads to the initial legitimacy ℓ_0 in the current world $\mathcal{W}$, which is written $\ell \leadsto_{\mathcal{W}} \ell_0$. Figure 8 defines the relation $\rightarrow_{ds}^{n,m}$ [6] to denote reductions that involve n steps among which m are mandatory. According to the evaluation relation eval_{ds}, a program is said to be divergent, i.e. its value is $\perp$, if it leads to an infinite transition sequence that regularly often contains mandatory transitions.

The soundness of the CEKDS-machine is estalished by proving that its evaluation relation eval_{ds} is equal to the sequetial evaluation function of the CEK-Machine.

Theorem 1 (Soundness) $\mathrm{eval}_{ds} = \mathrm{eval}_{ss}$ $\square$

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 [15, 3, 28, 14, 13, 11, 18]. Parallelism by annotations has been formalised recently only. Flanagan and Felleisen [6] have defined the semantics of future in a purely functional language. The author [19, 20] has proposed a semantic framework for continuations and side-effects in a language with the annotations pcall, fork, and future. This paper is the first to present a formal semantics for futures, side-effect, first-class continuations, and distribution. Our research is part of a project which aims at building a Virtual Multicomputer [23], which provides a soft architecture to support distributed applications, transcending the details of 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 Felleisen's [6] *touch analysis* remove 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 CEKDS-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 collection; 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 MultiScheme [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 CEKDS-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.

Queinnec's ICSLA [27, 24, 26, 25] is a dialect of Lisp offering primitives for parallelism, transparent migration of objects, and maintenance of their cache coherence over the network. Queinnec'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 CEKDS machine. Although his notion of coherency is not suitable to the CEKDS 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 task 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 this semantics and to precisely investigate the issue of data migration and distributed garbage collection within this framework.

References

1. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, C. E. Leiserson, Keith H. Randall, and Y. Zhou. Cilk: an Efficient Multithreaded Runtime System. In *PPOPP'95*, 1995.
2. K. M. Chandy and Kesselman C. CC++: A Declarative, Concurrent, Object Oriented Programming Notation. Technical Report CS-92-01, California Institute of Technology, 1992.
3. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, 1993.
4. M. Felleisen and D. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1986. Elsevier Pub.
5. Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *PARLE'87*, LNCS 259, pages 206–223. Springer-Verlag, 1987.
6. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *POPL'95*. Also in Technical Reports 238, 239, Rice University, 1994.
7. I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Approach. *Scientific Programming*, 1(1):51–66, 1992.
8. Message Passing Interface Forum. A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, June 1995.
9. I. Foster, C. Kesselman, and S. Tuecke. The Nexus Apporach to Integrating Multithreading and Communications. Math. and Comp. Sci. Division, Argonne National Laboratory, 1995.
10. Al. Geist and al. PVM 3 User's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Knoxville, Tennessee, May 1993.
11. Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In *Parallel Lisp : Languages and Systems*, LNCS 441, pages 2–57, 1990.
12. 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.
13. Takayasu Ito and Manabu Matsui. A Parallel Lisp Language Pailisp and its Kernel Specification. In *Parallel Lisp : Languages and Systems*, LNCS 441, pages 58–100, 1990.
14. Takayasu Ito and Tomohiro Seino. On Pailisp Continuation and its Implementation. In *Proceedings of the ACM SIGPLAN workshop on Continuations CW92*, pages 73–90, 1992.
15. Morry Katz and Daniel Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *LFP'90*, pages 176–184, June 1990.
16. Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *POPL'92*, pages 39–50, Albuquerque, New Mexico, 1992.
17. James S. Miller. *MultiScheme : A Parallel Processing System Based on MIT Scheme.* PhD thesis, MIT, 1987.
18. Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation : a Technique for Increasing the Granularity of Parallel Programs. In *LFP'90*, pages 185–197, June 1990.
19. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* PhD thesis, University of Liège, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.
20. Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, Philadelphia, May 1996.
21. Luc Moreau. Correctness of a Distributed-Memory Model for Scheme. Technical report M96/3, University of Southampton, 1996.
22. Luc Moreau and Daniel Ribbens. The Semantics of pcall and fork. In *PSLS 95 – Parallel Symbolic Langages and Systems*, LNCS 1068, Beaune, France, October 1995.
23. Julian Padget. Controlling (Virtual) Multicomputers. In *Massively Parallel Computer Systems (MPCS'94)*, pages 102–112. IEEE Computer Society Press, 1994.
24. Christian Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.
25. Christian Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. In *TPPP'94*, LNCS 700, pages 70–93, Sendai (Japan), 1994.
26. Christian Queinnec. DMEROON: a Distributed Class-based Causally-coherent Data Model: Preliminary Report. In *Parallel Symbolic Languages and Systems.*, LNCS 1068, 1995.
27. C. Queinnec and D. De Roure. Design of a Concurrent and Distributed Language. In *Parallel Symbolic Computing: Languages, Systems and Applications*, LNCS'748, p. 234–259, 1992.
28. Pete Tinker and Morry Katz. Parallel Execution of Sequential Scheme with ParaTran. In *LFP'88*, pages 28–39, 1988.