The semantics of `pcall` and `fork` in the presence of first-class continuations and side-effects

Luc Moreau[1] and Daniel Ribbens[2]

[1] Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, United Kingdom – Email: `L.Moreau@ecs.soton.ac.uk`
[2] Institut d'Electricité Montefiore, B28, University of Liège, 4000 Liège, Belgium – E-mail: `ribbens@montefiore.ulg.ac.be`

Abstract. We present the semantics of the annotations `pcall` and `fork` for parallel evaluation of Scheme. Annotated programs are proved to be behaviourly indistinguishable from their non-annotated counterparts, even in the presence of first-class continuations and side-effects. The semantics takes the form of an abstract machine, which can be regarded as a guideline for an implementation.

1 Introduction

Parallelism is usually considered as a promising way of speeding up the execution of programs. Nevertheless, the adjunction of several processors does not immediately guarantee a speedup of programs execution. Indeed, overheads due to synchronisations, process forking, and information copying and sharing can increase the execution time. In this perspective, the design of programming languages supporting parallelism and able to tame parallel architectures has become an important research subject, especially since parallel computers are more frequent and cheaper.

Functional languages allow the programmer to specify his programs at a rather high level of abstraction because they relieve him from the burden to consider the intricate details of the architecture running the program (for instance, by the first-class citizenship of objects, or by automatic memory management). In this paper, we consider mostly-functional programming languages like Scheme [26] and Standard ML of New Jersey [1], which provide side-effects and first-class continuations in addition to the purely functional style. These "effects" provide more expressiveness in the sense that programs using effects contain fewer programming patterns than equivalent programs in less expressive languages (conciseness conjecture [5]).

In addition, functional programming languages are good candidates to specify a parallel execution: indeed, the applicative style naturally allows function arguments to be evaluated in parallel before application.

The kind of parallelism that we study for these languages is based on annotations by which the programmer points out the parts of the program that can be evaluated in parallel. By definition, annotations must be *transparent*, that is, annotated terms must return the same[3] result as in the absence of annotations. This approach to parallelism is certainly compatible with the high-level abstraction of functional languages. Indeed, transparent annotations avoid the programmer to concentrate on parallelism-specific problems such as dead-locks, race conditions, and non-determinism.

In addition, a new programming methodology is not required to design annotation-based parallel programs since functional programs can be written traditionally

[3] The formulation "returns the same result" is intuitive, but it should preferably be replaced by "behaviourly indistinguishable" or "observationally equivalent".

and annotated afterwards only. The task of annotating programs, however, should not be underestimated because it controls the amount of parallelism that can be created at run-time and the speculative character of this parallelism.

Another advantage of this approach is the reusability of code: annotated programs can still run sequentially, and sequential programs can be "upgraded" to parallelism by annotating them.

It could be argued that such a high-level approach is not adapted to maximally utilise the parallel computing power of a machine. However, the ease of development compensates for this drawback. Furthermore, we consider that annotations are particularly suitable for coarse-grain parallelism.

The novelty of this paper is not to propose new annotations but to provide a semantics and a correctness proof for an annotated idealised version of Scheme with first-class continuations and side-effects. We consider the annotations `pcall` and `fork`, informally described as follows. An expression `(pcall M N)` evaluates `M` and `N` in parallel and then applies the value of `M` on the value of `N`. Such an expression is called a *parallel application* and is equivalent to a sequential application `(M N)`. An expression `(begin (fork M) N)` also evaluates the expressions `M` and `N` in parallel; it is behaviourly indistinguishable from `(begin M N)`.

Providing transparent annotations is not trivial, especially in the presence of effects (first-class continuations or side-effects). Indeed, it is possible to write programs using effects such that their values depend on the evaluation order. These programs become non deterministic if execution is parallel and if effects are uncontrolled. The following two examples illustrate the point:

```
(call/cc (lambda (k) (pcall (k 1) (k 2))))  [1]
(let ((x 1))
 (pcall list (begin (set! x (+ x 1)) x)     [2]
             (begin (set! x (+ x 1)) x)))
```

The expression `[1]` returns 1, 2, or even both of them [25], and the expression `[2]` returns `(2 3)`, `(3 2)`, or even `(2 2)`, `(3 3)`, if execution is interleaved. Even though these programs are simple, they reflect race-condition problems that are frequent in traditional parallel programming languages and that we want to avoid by annotation-based parallelism.

Research on annotation-based parallelism was pioneered by Baker, Hewitt [2], and Halstead's MultiLisp [11]. Most research on this subject dealt with the implementation of a language with annotations for parallelism, and practice has been ahead of theory in this field for a long time [11, 16, 31, 17, 4, 14, 15]. Recently only, Flanagan and Felleisen proposed a semantics of `future` in a purely functional language. On the other hand, the authors of the present paper [22, 19, 21] defined a semantic framework for the annotations `pcall` and `fork` in a functional language with first-class continuations. Here, we generalise the semantics proposed in [21] to side-effects.

It is rather obvious that preserving the sequential semantics of an annotated program with effects requires synchronisations between computations. The goal of this paper is to design a semantics that introduces as few synchronisations as possible so that parallelism can exist. It has been observed that mostly-functional programs use effects in a local way [24]: e.g. continuations are captured and invoked inside a function, or side-effects are performed locally in a function. The semantics that we propose offers parallelism when effects remain local: an expression e_2 is allowed to run speculatively in parallel with an expression e_1 as long as e_2 cannot observe effects performed by e_1 on data-structures (or continuations) accessible to both of them. Furthermore, we consider that our semantics is a suitable basis for further optimisations by program analysis (e.g. by detecting write/write or read/write conflicts). This aspect will be developed later in the paper.

The main contributions of this paper are:

1. We formalise the semantics of the annotations `pcall` and `fork` in the presence of both side-effects and first-class continuations. This is the first time that both effects are simultaneously considered in an approach to parallelism by annotations.
2. We define an abstract machine, called the PCKS-Machine, able to execute programs annotated by `pcall` and `fork` in parallel. Although preliminary versions of this machine were proposed in [19, 21], the version described here is reorganised so that sequential programs can be executed on this machine as efficiently as on a sequential machine.
3. By the proof of correctness of the PCKS-Machine, we can show that the PCKS-Machine is a possible parallel implementation of Felleisen, Sabry, and Hieb's λ_v-CS-calculus [9, 29], a calculus for state and continuations.

This paper is organised as follows. The PCKS-Machine is described in Section 2; its formal properties and proof of correctness are stated in Section 3. A discussion of the proposed semantics and a comparison with related work follow in Section 4. Appendix A contains a major Lemma used in the correctness proof.

2 The PCKS-Machine

Felleisen and Friedman [6] proposed the CK-Machine to evaluate sequential functional programs with first-class continuations; it is characterised by a *control string* (C) and a *continuation* (K). The control string is the term being evaluated and the continuation represents the operations that remain to be performed after evaluating the current control string.

The PCKS-Machine is an extension of the CK-Machine to parallelism; it models a MIMD architecture with a shared memory. Its computational state is described by a set of processes running in parallel (P); each process is a CK-configuration (CK) and has access to a shared memory (S). The PCKS-Machine is described below in three different sections according to the subset of the language to evaluate: purely functional, with first-class continuations, and with side-effects.

2.1 Purely Functional Subset

In this paper, we consider Λ_u, an idealised version of Scheme, of which the syntax is displayed in Figure 1. User terms are composed of values, *sequential applications* $(X\ X)$, and *parallel applications* $(\mathsf{pcall}\ X\ X)$. We consider the annotation pcall only because fork can be defined a follows:

$$(\texttt{begin}\ (\mathsf{fork}\ M)\ N) \equiv (\mathsf{pcall}\ (\texttt{begin}\ M\ (\lambda x.x))\ N),$$

with $(\texttt{begin}\ M\ N) \equiv (\lambda d.N)\ M$ with $d \notin FV(M)$. We adopt Barendregt's [3] Definitions and Conventions on free and bound variables. We define a *program* as a user term without free variable. Figure 2 contains the transition rules to evaluate the functional subset: the first set of rules deals with sequential evaluation, while the second set deals with parallel evaluation.

A PCKS-configuration describes the computational state of the PCKS-machine; it is composed of a set of processes P and a store σ. Processes can be *active* or *dead*. An active process is defined by a CK-configuration composed of a control string, i.e. a term M to evaluate, and a continuation κ representing what remains to be performed after obtaining the value of M. On the other hand, dead processes are represented by $\langle \ddagger, (\mathbf{stop})\rangle_\tau$: these are processes that have completed the evaluation of an expression and that have nothing left to do.

$$\begin{array}{lll}
M \in \Lambda_{pcks} & ::= V \mid (M\ M) \mid (\mathsf{pcall}\ M\ M) & \text{(Term)}\\
V & ::= c \mid x \mid (\lambda x.M) \mid \langle \mathsf{co}\ \kappa\rangle \mid b \mid (\mathsf{setref!}\ V) & \text{(Value)}\\
c & ::= a \mid d & \text{(Constant)}\\
b \in Box & ::= \langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle & \text{(Box)}\\
d \in \mathcal{D} & ::= \mathsf{callcc} \mid \mathsf{makeref} \mid \mathsf{deref} & \text{(Distinguished Constant)}\\
 & \quad \mid \mathsf{setref!} \mid \mathit{void} & \\
A & ::= c \mid \mathbf{Procedure} \mid \mathbf{Continuation} \mid \mathbf{Box} & \text{(Answer)}\\
s_\kappa & ::= () \mid \mathrm{cons}((\alpha_\mathrm{m}, \alpha_\mathrm{n}), \mathrm{s}_\kappa) & \text{(Spine of Continuation)}\\
X \in \Lambda_u & ::= V_x \mid (X\ X) \mid (\mathsf{pcall}\ X\ X) & \text{(User Term)}\\
V_x & ::= c \mid x \mid (\lambda x.X) & \text{(Syntactic Value)}
\end{array}$$

$$\begin{array}{lll}
\kappa & ::= \langle \gamma, \eta\rangle & \text{(Expression Continuation)}\\
\gamma & ::= (\mathbf{initp}) \mid (\gamma\ \mathbf{fun}\ V) \mid (\gamma\ \mathbf{arg}\ M) \mid (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, M)) & \text{(Local}\\
 & \quad \mid (\gamma\ \mathbf{unbox}(\kappa)) \mid (\gamma\ \mathbf{setbox}(\kappa, V)) & \text{Continuation)}\\
\eta & ::= (\mathbf{init}) \mid (\kappa\ \mathbf{right}(\alpha_m, \alpha_n)) & \text{(Synchronising Continuation)}
\end{array}$$

$$\begin{array}{llll}
\mathcal{M} & & ::= \langle P, \sigma\rangle & \text{(PCKS-configuration)}\\
p \in P & & ::= \langle M, \kappa\rangle_\tau \mid \langle \ddagger, (\mathbf{stop})\rangle_\tau & \text{(Process)}\\
\sigma & & :\ Loc \to Contents & \text{(Store)}\\
 & Contents & ::= V \mid \bot \mid \langle \mathsf{sco}\ \langle \mathsf{co}\ \kappa\rangle\ V\rangle & \text{(Store Content)}\\
 & & \quad \mid \langle \mathsf{subx}\ \kappa\ b\rangle & \\
 & & \quad \mid \langle \mathsf{ssbx}\ \kappa\ b\ V\rangle & \\
 & Loc & =\ Loc_l \cup Loc_r \cup Loc_b & \text{(Location)}
\end{array}$$

$a \in \mathcal{K}$	: Constant
$x \in Var$	: Variable
$\tau \in Pid$	: Process Identifier
$\alpha_m \in Loc_l$	: Left Location
$\alpha_n \in Loc_r$	: Right Location
$\alpha_b \in Loc_b$	: Box Location

$$\begin{array}{rl}
\mathrm{Unload} & :\ V^0 \to A\\
\mathrm{Unload}[c] & = c\\
\mathrm{Unload}[\lambda x.M] & = \mathbf{Procedure}\\
\mathrm{Unload}[\langle \mathsf{co}\ \kappa\rangle] & = \mathbf{Continuation}\\
\mathrm{Unload}[\langle \mathsf{bx}\ s_\kappa, \alpha\rangle] & = \mathbf{Box}
\end{array}$$

Fig. 1. State Space of PCKS-Machine

For the purpose of efficiency[4], the continuation κ of a process is structured in two components: $\langle \gamma, \eta\rangle$. The component γ, called the *local continuation*, represents what remains to be performed by the process before having to synchronise with other processes. The component η, called the *synchronising continuation*, represents what remains to be performed when a process has obtained a value. As a result the computation that is yet to be performed after evaluating an expression is given by the succession of actions denoted by γ and η. Continuations can be seen as record-like data-structures called *continuation codes*.

[4] In earlier versions of the PCKS-Machine [19, 21], continuations were not composed of two components.

$$\langle (M\ N), \langle \gamma, \eta \rangle \rangle_\tau \overset{cks}{\mapsto} \langle M, \langle (\gamma\ \mathbf{arg}\ N), \eta \rangle \rangle_\tau \qquad \text{(operator)}$$

$$\langle V, \langle (\gamma\ \mathbf{arg}\ N), \eta \rangle \rangle_\tau \overset{cks}{\mapsto} \langle N, \langle (\gamma\ \mathbf{fun}\ V), \eta \rangle \rangle_\tau \qquad \text{(operand)}$$

$$\langle V, \langle (\gamma\ \mathbf{fun}\ \lambda x.M), \eta \rangle \rangle_\tau \overset{cks}{\mapsto} \langle M\{V/x\}, \langle \gamma, \eta \rangle \rangle_\tau \qquad (\beta_v)$$

$$\langle a', \langle (\gamma\ \mathbf{fun}\ a), \eta \rangle \rangle_\tau \overset{cks}{\mapsto} \langle \delta(a, a'), \langle \gamma, \eta \rangle \rangle_\tau \qquad (\delta)$$

$$\begin{array}{ll}
\langle (\mathsf{pcall}\ M\ N), \langle \gamma, \eta \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \{\ \langle M, \langle (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N)), \eta \rangle \rangle_\tau, & \text{(pcall)} \\
\qquad \langle N, \langle (\mathbf{initp}), (\langle \gamma, \eta \rangle\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle \rangle_{\tau_i}\ \ \} & \\
\qquad \text{with } \alpha_m, \alpha_n \notin DOM(\sigma),\ \text{a new } \tau_i \in\ Tid & \\
\langle V, \langle (\mathbf{initp}), (\langle \gamma, \eta \rangle\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle \dagger, (\mathbf{stop}) \rangle_\tau ; \sigma[\alpha_n \leftarrow V] \quad \text{if } \sigma(\alpha_m) = \bot & (\mathsf{stop}_r) \\
\langle V, \langle (\mathbf{initp}), (\langle \gamma, \eta \rangle\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle V, \langle (\gamma\ \mathbf{fun}\ \sigma(\alpha_m)), \eta \rangle \rangle_\tau ; \sigma[\alpha_n \leftarrow V] \quad \text{if } \sigma(\alpha_m) \neq \bot & (\mathsf{ret}_r) \\
\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N)), \eta \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle \dagger, (\mathbf{stop}) \rangle_\tau ; \sigma[\alpha_m \leftarrow V] \quad \text{if } \sigma(\alpha_m) = \bot \wedge \sigma(\alpha_n) = \bot & (\mathsf{stop}_l) \\
\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N)), \eta \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle \sigma(\alpha_n), \langle (\gamma\ \mathbf{fun}\ V), \eta \rangle \rangle_\tau ; \sigma[\alpha_m \leftarrow V] \quad \text{if } \sigma(\alpha_m) = \bot \wedge \mathit{value?}(\sigma(\alpha_n)) & (\mathsf{ret\text{-}1}_l) \\
\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N)), \eta \rangle \rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle N, \langle (\gamma\ \mathbf{fun}\ V), \eta \rangle \rangle_\tau \quad \text{if } \sigma(\alpha_m) \neq \bot & (\mathsf{ret\text{-}n}_l)
\end{array}$$

Fig. 2. PCKS-Machine: Purely Functional Subset

In order to specify the legal transitions between configurations of the PCKS-machine, we first define the *CKS-transition* as a relation on processes (represented by CK-configurations) and stores (S).

Definition 1 (CKS-transition) A CKS-transition is a relation $\langle p, \sigma_1 \rangle \overset{cks}{\mapsto} \langle P, \sigma_2 \rangle$, which associates a process p and a store σ_1 with a set of processes P and a store σ_2. $\square$

The applicability of a CKS-transition can depend on the content of the store, and a CKS-transition can update the store (hence the returned store σ_2). Furthermore, a CKS-transition produces a *set* of processes, because new processes can be created (by the pcall-construct). We assume that CKS-transitions are performed *atomically*. We can now define a transition and the evaluation relation of the PCKS-Machine.

Definition 2 (PCKS-transition) There is a transition between two PCKS configurations $\mathcal{M}_1 \equiv \langle P_1, \sigma_1 \rangle$ and $\mathcal{M}_2 \equiv \langle P_2, \sigma_2 \rangle$, written $\mathcal{M}_1 \mapsto_{pcks} \mathcal{M}_2$, if there exists a process p and a set of processes P such that $\langle p, \sigma_1 \rangle \overset{cks}{\mapsto} \langle P, \sigma_2 \rangle$ with $p \in P_1$ and $P_2 \equiv P_1 \setminus \{p\} \cup P$. □

The evaluation relation of the PCKS-Machine can now be formalised.

Definition 3 ($\mathsf{eval}_{\mathsf{pcks}}$) The evaluation relation is defined for a program M and the value is A, written $\mathsf{eval}_{\mathsf{pcks}}(M) = A$, if there exists a final configuration $\mathcal{M}_f$ that contains a process of the form $\langle V, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle_\tau$, and such that, for the initial configuration $\mathcal{M}_i \equiv \langle \{ \langle M, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle_{\tau_0} \}, \emptyset \rangle$, we have $\mathcal{M}_i \mapsto^*_{pcks} \mathcal{M}_f$, with $\mathrm{Unload}[V] = A$. □

First let us describe the transitions for evaluating sequential terms. Rules operator and operand adapted from the CK-Machine force a left-to-right evaluation order of application subexpressions. If a sequential application $(M\ N)$ is the control string of a process, rule operator yields a configuration whose control string is the operator M, and whose local continuation γ is extended by a code **arg** N, indicating that the operand N remains to be evaluated. When a value for the operator is obtained, rule operand transforms the process into a configuration whose control string is the operand N, and whose local continuation is extended by the code **fun** V, after having removed the code **arg** N. The code **fun** V means that the value V has still to be applied on the value of the current control string.

Applications can be performed by rules (β_v) or (δ). The former implements the call-by-value β-reduction [23], where $M\{V/x\}$ denotes the substitution of V for x in M. Rule (δ) performs the δ-reduction; that is, it applies the functional constant a on the constant a'. If a program contains sequential applications only, the component η of the continuation constantly keeps the value $(\mathbf{init})$.

If the control string of a process is a parallel application $(\mathsf{pcall}\ M\ N)$, a new process is created, with a fresh process identifier τ_i. The control string of the new process is the operand N, the η component of its continuation is a code **right** indicating that the expression being evaluated is the operand of a parallel application, and the γ component is $(\mathbf{initp})$ meaning that after evaluating N there remains nothing to do locally. The control string of the process τ becomes the operator M, and its local continuation **left** indicates that it is the operator of a parallel application. The reader should notice that a code **left** is in the grammar of local continuations, while a code **right** is in the grammar of synchronising continuations.

In addition, both codes **left** and **right** contain two freshly allocated locations α_m and α_n. Initially, the contents of these locations is empty, which is represented by $\perp$. The locations α_m and α_n are intended to receive the values of M and N, respectively. Not only do they serve as temporary "buffers" to receive the values, but also are they used for synchronising the processes evaluating the operator and the operand as explained below.

After a transition pcall, the evaluations of M and N can proceed in parallel, which is modelled by an interleaving of their transitions. Either the operator or the operand terminates first. Let us examine the latter case, a symmetric explanation follows for the former.

A process has evaluated the operand of a parallel application, if its control string is a value and its synchronising continuation is a code **right**. It has access to the content of the location α_m that appears in its continuation. If location α_m is empty (rule stop_r), it means that the operator is not evaluated yet, and the application cannot be performed: so, the process evaluating the operand must be stopped, which is represented by the dead process $\langle \ddagger, (\mathbf{stop}) \rangle_\tau$. On the contrary, if the location α_m contains a value, this value is the value of the operator, and it can be applied on the value of the operand as indicated by ret_r. In both cases, the location α_n is updated with the value of the operand. Rules stop_l and $\mathsf{ret}\text{-}\mathsf{1}_l$ deal with the symmetric case.

Let us note that there is an occurrence of the operand N of a parallel application in the local continuation of the operator, which is of the form $(\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N))$. Its role as well as rule $\mathsf{ret}\text{-}\mathsf{n}_l$ will be explained in the sequel.

2.2 First-Class Continuations

Figure 3 displays how the PCKS-Machine can be extended to support first-class continuations. The set of terms is extended by a distinguished constant callcc, which is the primitive used to *capture* (or *reify*) a first-class continuation (it corresponds to `call-with-current-continuation` in Scheme [26]). In addition, an object representing a first-class continuation is added to the set of values. It is of the form $\langle \mathsf{co}\ \kappa \rangle$ where κ is an expression continuation, and co a tag distinguishing first-class continuations from any other object; such object is usually called a *continuation point* [6]. Furthermore, locations, which could either contain values or be empty, can now accept a data-structure whose role is described below, while commenting the transition rules.

$$
\begin{array}{ll}
\langle V, \langle (\gamma\ \mathbf{fun}\ \mathsf{callcc}\ ), \eta \rangle\rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle, \langle (\gamma\ \mathbf{fun}\ V), \eta\rangle\rangle_\tau & (\mathsf{capture}) \\
\langle V, \langle (\gamma'\ \mathbf{fun}\ \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle), \eta'\rangle\rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle V, \langle \gamma, \eta\rangle\rangle_\tau \quad \mathsf{if}\ \langle \gamma, \eta\rangle \sqsupseteq \langle (\mathbf{initp}), \eta'\rangle & (\mathsf{invoke}) \\
\langle V, \langle (\gamma'\ \mathbf{fun}\ \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle), (\langle \gamma'', \eta'\rangle\ \mathbf{right}(\alpha_m, \alpha_n))\rangle\rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle V, \langle (\gamma''\ \mathbf{fun}\ \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle), \eta'\rangle\rangle_\tau & (\mathsf{prune\text{-}invoke}) \\
\qquad \mathsf{if}\ \sigma(\alpha_m) \neq \bot \wedge \langle \gamma, \eta\rangle \not\sqsupseteq \langle (\mathbf{initp}), (\langle \gamma'', \eta'\rangle\ \mathbf{right}(\alpha_m, \alpha_n))\rangle & \\
\langle V, \langle (\gamma'\ \mathbf{fun}\ \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle), (\langle \gamma'', \eta'\rangle\ \mathbf{right}(\alpha_m, \alpha_n))\rangle\rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle \dagger, (\mathbf{stop})\rangle_\tau; \sigma[\alpha_n \leftarrow \langle \mathsf{sco}\ \langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle\ V\rangle] & (\mathsf{suspend\text{-}invoke}) \\
\qquad \mathsf{if}\ \sigma(\alpha_m) = \bot \wedge \langle \gamma, \eta\rangle \not\sqsupseteq \langle (\mathbf{initp}), (\langle \gamma'', \eta'\rangle\ \mathbf{right}(\alpha_m, \alpha_n))\rangle & \\
\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m, \alpha_n, N)), \eta\rangle\rangle_\tau & \\
\quad \overset{cks}{\mapsto} \langle V_1, \langle (\gamma\ \mathbf{fun}\ \langle \mathsf{co}\ \langle \gamma', \eta'\rangle\rangle), \eta\rangle\rangle_\tau; \sigma[\alpha_m \leftarrow V] & (\mathsf{resume\text{-}invoke}) \\
\qquad \mathsf{if}\ \sigma(\alpha_n) = \langle \mathsf{sco}\ \langle \mathsf{co}\ \langle \gamma', \eta'\rangle\rangle\ V_1\rangle \wedge \sigma(\alpha_m) = \bot &
\end{array}
$$

Fig. 3. PCKS-Machine: First-Class Continuations

Rule $\mathsf{capture}$ defines the behaviour of the primitive callcc. When callcc is ready to be applied on a value as in the process $\langle V, \langle (\gamma\ \mathbf{fun}\ \mathsf{callcc}\ ), \eta\rangle\rangle_\tau$, rule $\mathsf{capture}$ packages up the local continuation γ and the synchronising continuation η as a first-class object $\langle \mathsf{co}\ \langle \gamma, \eta\rangle\rangle$. This continuation point has the same status as a numerical value; that is, it can be passed to or returned by functions, and stored in data-structures. Furthermore, a first-class continuation can be applied on a value, which is called *invoking* the continuation on a value. Rule invoke specifies the behaviour of continuation invocation. Assuming that the side-condition is satisfied (we will describe it later), the effect of invoking a continuation $\langle \mathsf{co}\ \langle \gamma, \eta \rangle \rangle$ on a value V is to replace the continuation of the call, i.e. $\langle \gamma', \eta' \rangle$, by $\langle \gamma, \eta \rangle$.

Invoking a continuation in a parallel environment is a delicate operation [18, 22, 19]. Indeed, let us again consider the first example presented in Section 1. An unrestricted use of rule invoke[5] can produce the answers 1 and 2 for the expression $(\mathsf{callcc}\ \lambda k.(\mathsf{pcall}\ (k\ 1)\ (k\ 2)))$, where the first-class continuation k is invoked on 1 and 2 in parallel. Rule invoke without side-condition is unsound because pcall is no longer an annotation for parallelism.

As in previous papers [22, 19, 21], our approach for invoking continuations is the following: a continuation can be invoked only if it can be proved that its invocation does not infringe the sequential semantics. This approach is usually called *non-speculative* invocation; it essentially consists in waiting for the values of some expressions that should be evaluated before the invocation in a left-to-right evaluation order. The challenge is to wait for the values of a *minimal* number of expressions so that execution can indeed proceed in parallel, while still preserving the sequential semantics.

Generally, the current continuation of a process and the invoked continuation have a common prefix. Hence, invoking a continuation is equivalent to removing a suffix of the current continuation, and to replacing it by a suffix of the invoked continuation. This notion of prefix can be formalised by the relation "is an extension" based on the relations[6] γ-extension and η-equality.

Definition 4 (γ-Extension) A continuation γ_1 is a *"one-step" γ-extension* of a continuation γ_2, written $\gamma_1 \sqsupset^\gamma \gamma_2$, if one of the following equations holds.

$$\gamma_1 \equiv (\gamma_2\ \mathbf{fun}\ V),\ \gamma_1 \equiv (\gamma_2\ \mathbf{unbox}\ (\kappa)),$$
$$\gamma_1 \equiv (\gamma_2\ \mathbf{arg}\ N),\ \gamma_1 \equiv (\gamma_2\ \mathbf{setbox}(\kappa, V))$$

The relation *γ-extension* is the reflexive, transitive closure of *"one-step" γ-extension* and is written $\gamma_1 \sqsupseteq^\gamma \gamma_2$. □

Definition 5 (η-equality) A continuation η_1 is equal to a continuation η_2, $\eta_1 =^\eta \eta_2$ if $\eta_1 \equiv (\mathbf{init})$ and $\eta_2 \equiv (\mathbf{init})$, □
or if $\eta_1 \equiv (\kappa\ \mathbf{right}(\alpha_{m1}, \alpha_{n1}))$, $\eta_2 \equiv (\kappa'\ \mathbf{right}(\alpha_{m2}, \alpha_{n2}))$, and $\alpha_{m1} = \alpha_{m2}$.

Note that equality on η-continuations requires to compare locations (not their contents in shared memory!) or two initial continuations.

Definition 6 (Extension) Let σ be the store of a given PCKS-configuration. A continuation $\langle \gamma_1, \eta_1 \rangle$ is an *extension* of a continuation $\langle \gamma_2, \eta_2 \rangle$, written $\langle \gamma_1, \eta_1 \rangle \sqsupseteq \langle \gamma_2, \eta_2 \rangle$, if $\gamma_1 \sqsupseteq^\gamma \gamma_2$ and $\eta_1 =^\eta \eta_2$, or if $\eta_1 \equiv (\kappa\ \mathbf{right}(\alpha_m, \alpha_n))$, and $\kappa \sqsupseteq \langle \gamma_2, \eta_2 \rangle$. □

In a parallel setting, the dangerous operation is to delete a suffix of the current continuation because this can discard an uncompleted computation that can potentially escape and that should have been evaluated before the invocation in a left-to-right evaluation order. Rule prune-invoke handles the operation of deleting the suffix of a continuation. In the left-hand side of rule prune-invoke, the continuation of the call of $\langle \mathsf{co}\ \langle \gamma, \eta \rangle \rangle$ has a local continuation γ' and a synchronising continuation $(\langle \gamma'', \eta' \rangle\ \mathbf{right}(\alpha_m, \alpha_n))$. The local continuation γ' can be discarded

[5] Rule invoke without side-condition is the rule used in the CK-Machine [6], modulo the reorganisation of the continuation in two components.

[6] The Definition 4 already contains continuation codes related to side-effects.

because it was produced by a sequential (left-to-right order) evaluation. On the contrary, the code $\mathbf{right}(\alpha_m, \alpha_n)$ can be deleted only if the corresponding operator of the parallel application has completed its execution. Indeed, if the operator has returned a value, it will not be able to escape. The operator is already evaluated if its associated location α_m is not empty. If this condition is not satisfied, the invocation must be temporarily suspended until the operator yields a value. Rule suspend-invoke stops a process invoking a continuation in the operand of a parallel application when the operator is not yet evaluated. In order to remember that we tried to invoke a continuation, a data-structure containing the continuation and the value on which it was invoked is stored in location α_n. The invocation is resumed as soon as the operator yields a value, which is detected by rule resume-invoke. Let us notice the efficiency gained by the organisation of a continuation in two components: rule prune-invoke can delete a local continuation γ' in a single step, without considering each code composing γ'.

A succession of prune-invoke, suspend-invoke, and resume-invoke safely deletes a suffix of the current continuation such that the current continuation becomes a prefix of the invoked continuation. More precisely, the synchronising component of the current continuation becomes a prefix of the invoked continuation. This situation, detected by the side-condition of invoke, is the state in which we would be if evaluation had been from left to right. So, rule invoke can replace the current continuation by the invoked one.

First-class continuations have an unlimited extent; that is, they can be invoked as long as they remain accessible. So a same first-class continuation can be invoked several times. In the sequential semantics, the continuation of an operator is of the form $\mathbf{arg}\ N$. If it is captured and invoked several times, rule operand forces the reevaluation of N for every invocation. The parallel semantics must behave similarly. Hence, rule ret-n_l detects that a value has already been returned to the continuation of the operator and forces the reevaluation of the operand N, which appears in code $\mathbf{left}(\alpha_m, \alpha_n, N)$.

2.3 Side-Effects

As an illustration of the programming style that can be adopted with first-class continuations as defined in Section 2.2, let us extend our language with side-effects while still preserving the transparency property of annotations for parallelism. We assume that three new primitives are provided: `(new)` returns a fresh uninitialised location; `(unsafe-deref loc)` reads the content of location `loc`; `(unsafe-setref! loc V)` sets the content of location `loc` to the value `V`.

The last two primitives are "unsafe" because they can break the transparency of annotations for parallelism. Figure 4 displays three functions concerned with side-effects. The function `makeref` returns a `box` object, which contains a continuation and a freshly allocated location. The continuation applies a thunk and passes its value to the continuation of the call of `makeref`. The function `deref` returns the content of the box received in argument. The function `setref!` requires a box and a value V in argument, and sets the content of the location contained in the box with the value V.

Unlike `unsafe-deref` and `unsafe-setref!`, `deref` and `setref!` are safe because they preserve the transparency property of annotations for parallelism. Indeed, dereference and assignment are only performed after invoking the continuation that existed when the box was created. This ensures that dereference (or assignment) of a box is only performed by the leftmost expression that has access to this box; that is, the one that would be evaluated first in a left-to-right evaluation order.

```
(define-structure (box cont loc))

(define (makeref V)
  ((call/cc (lambda (k)
              (let ((loc (new)))
                (unsafe-setref! loc V)
                (lambda () (make-box k loc)))))))

(define (deref box)
  (call/cc (lambda (k)
             ((box-cont box) (lambda ()
                               (k (unsafe-deref (box-loc box))))))))

(define (setref! box V)
  (call/cc (lambda (k)
             ((box-cont box) (lambda ()
                               (unsafe-setref! (box-loc box) V)
                               (k void))))))
```

Fig. 4. First-Class Locations

Figure 4 illustrates the power of first-class continuations in the parallel framework: they are used to synchronise processes according to the sequential evaluation order. However, this implementation of boxes is quite expensive because it requires to capture a continuation for every created box, and it requires to capture a continuation and to invoke two continuations for every access to a box. In Figure 5, we present a new set of rules for the PCKS-Machines, which deal with side-effects in a more efficient way (although based on the same general principle).

In Figure 5, the set of terms is extended with a new value, which is a box object $\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle$, where bx is a tag identifying all the boxes, α is a location containing the value given to the box, and s_κ is the *spine of a continuation*. The spine of a continuation is a list of location pairs (α_m, α_n) collected from the synchronising components of a continuation as follows.

Definition 7 (Spine) The spine s_κ of a continuation κ is defined by the function $\mathcal{S}_p$:

$$\mathcal{S}_p(\langle\gamma, (\mathbf{init})\rangle) = ()$$
$$\mathcal{S}_p(\langle\gamma, (\kappa\ \mathbf{right}(\alpha_m, \alpha_n))\rangle) = \mathrm{cons}((\alpha_\mathrm{m}, \alpha_\mathrm{n}), \mathcal{S}_\mathrm{p}(\kappa))$$

□

The spine of a continuation contains the locations needed to determine whether this continuation is an extension of another one. The length of a spine is given by the number of nested parallel calls in operand position.

Three new distinguished constants represent the operators offered to the programmer to manipulate boxes. Since setref! is a binary function, and all functions are curried, we consider (setref! V) as a value as well. The constant *void* is the value returned after an assignment.

Rule makeref is the counterpart of rule capture: both rules create a first-class object. Roughly speaking, rule makeref implements the function makeref of Figure 4. It passes the continuation of the call of makeref a new box, with a freshly allocated

$\langle V, \langle (\gamma\ \mathbf{fun}\ \mathsf{makeref}), \eta \rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ \mathcal{S}_p(\langle\gamma,\eta\rangle), \alpha_b\rangle, \langle\gamma,\eta\rangle\rangle_\tau; \sigma[\alpha_b \leftarrow V]$ with $\alpha_b \notin DOM(\sigma)$ (makeref)

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{fun}\ \mathsf{deref}), \eta\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{unbox}(\langle\gamma,\eta\rangle)), \eta\rangle\rangle_\tau$ (deref)

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma'\ \mathbf{unbox}(\langle\gamma,\eta\rangle)), \eta'\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \sigma(\alpha_b), \langle\gamma,\eta\rangle\rangle_\tau \quad \text{if } s_\kappa \sqsupseteq \eta'$ (unbox)

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma'\ \mathbf{unbox}(\langle\gamma,\eta\rangle)), (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma''\ \mathbf{unbox}(\langle\gamma,\eta\rangle)), \eta'\rangle\rangle_\tau$ (prune-unbox)

if $\sigma(\alpha_m) \neq \bot \wedge s_\kappa \not\sqsupseteq (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma'\ \mathbf{unbox}(\langle\gamma,\eta\rangle)), (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \dagger, (\mathbf{stop})\rangle_\tau; \sigma[\alpha_n \leftarrow \langle \mathsf{subx}\ \langle\gamma,\eta\rangle\ \langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle\rangle]$ (susp-unbox)

if $\sigma(\alpha_m) = \bot \wedge s_\kappa \not\sqsupseteq (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$

$\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m,\alpha_n,N)), \eta\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{unbox}(\kappa)), \eta\rangle\rangle_\tau; \sigma[\alpha_m \leftarrow V]$ (resume-unbox)

if $\sigma(\alpha_n) = \langle \mathsf{subx}\ \kappa\ \langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle\rangle \wedge \sigma(\alpha_m) = \bot$

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{fun}\ \mathsf{setref!}), \eta\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \mathsf{setref!}'\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle\gamma,\eta\rangle\rangle_\tau$ (setref)

$\langle V, \langle (\gamma\ \mathbf{fun}\ \mathsf{setref!}'\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle), \eta\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{setbox}(\langle\gamma,\eta\rangle, V)), \eta\rangle\rangle_\tau$ (setref$'$)

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma'\ \mathbf{setbox}(\langle\gamma,\eta\rangle, V)), \eta'\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \mathit{void}, \langle\gamma,\eta\rangle\rangle_\tau; \sigma[\alpha_b \leftarrow V] \quad \text{if } s_\kappa \sqsupseteq \eta'$ (setbox)

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{setbox}(\kappa, V)), (\langle\gamma',\eta\rangle\ \mathbf{right}(\alpha_m,\alpha_n))\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma'\ \mathbf{setbox}(\kappa, V)), \eta\rangle\rangle_\tau$ (prune-setbox)

if $\sigma(\alpha_m) \neq \bot \wedge s_\kappa \not\sqsupseteq (\langle\gamma',\eta\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$

$\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{setbox}(\kappa, V)), (\kappa'\ \mathbf{right}(\alpha_m,\alpha_n))\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \dagger, (\mathbf{stop})\rangle_\tau; \sigma[\alpha_n \leftarrow \langle \mathsf{ssbx}\ \kappa\ \langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle\ V\rangle]$ (susp-setbox)

if $\sigma(\alpha_m) = \bot \wedge s_\kappa \not\sqsupseteq (\kappa'\ \mathbf{right}(\alpha_m,\alpha_n))$

$\langle V, \langle (\gamma\ \mathbf{left}(\alpha_m,\alpha_n,N)), \eta\rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle (\gamma\ \mathbf{setbox}(\kappa, V_1)), \eta\rangle\rangle_\tau; \sigma[\alpha_m \leftarrow V]$ (resume-setbox)

if $\sigma(\alpha_n) = \langle \mathsf{ssbx}\ \kappa\ \langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle\ V_1\rangle \wedge \sigma(\alpha_m) = \bot$

Fig. 5. PCKS-Machine: First-Class Locations

location α initialised to the value received in argument by makeref. Furthermore, the box is also given the spine of the continuation of the call of makeref, instead of the continuation itself as in Figure 4.

The analogy between the sets of rules for continuations, dereferences, and assignments is striking. Rules unbox to resume-unbox and rules setbox to resume-setbox match rules invoke to resume-invoke. Let us explain the rules for assignment; similar comments also apply to rules for dereference. After using rules setref and setref′, both arguments of setref! are collected so as to reach a configuration of the form $\langle\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle, \langle(\gamma\ \mathbf{setbox}(\kappa', V)), \eta\rangle\rangle_\tau$, (with κ' initially equal to $\langle\gamma, \eta\rangle$). Its meaning is that the box $\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle$ is ready to be assigned a value V, and that the continuation of the assignment is κ'.

In Figure 4, an assignment requires to invoke the continuation held in the box. According to rule capture, a continuation can be invoked if it is an extension of the synchronising component of the current continuation. Rule setbox combines both ideas: an assignment can be performed if the continuation that existed when the box was created is an extension of the synchronising component of the current continuation. Such a property is written $s_\kappa \sqsupseteq \eta'$ where the operator $\sqsupseteq$ is overloaded for spines as follows.

Definition 8 (Spine Extension)
$s_\kappa \sqsupseteq (\mathbf{init})$ and $\mathrm{cons}((\alpha_{\mathrm{m}_1}, \alpha_{\mathrm{n}_1}), \mathrm{s}_\kappa) \sqsupseteq (\kappa'\ \mathbf{right}(\alpha_{\mathrm{m}_2}, \alpha_{\mathrm{n}_2}))$ if $\alpha_{m_1} = \alpha_{m_2}$ or $s_\kappa \sqsupseteq (\kappa' \mathbf{right}(\alpha_{m_2}, \alpha_{n_2}))$. $\square$

Let us notice that the spine of a continuation is sufficient to determine whether a continuation is an extension of another one. Hence, by extending the relation $\sqsupseteq$ to spines, we do not need to capture the entire continuation in rule makeref (as opposed to what we did in Figure 4).

In order to perform an assignment on a box, we have to come down to a situation where the spine contained in the box is an extension of the synchronising part of the current continuation. Like continuation invocation, the suffix of the current continuation can be deleted by rules prune-setbox, susp-setbox, and resume-setbox. A new data-structure ssbx is used when the operand of a parallel application tries to assign a box, whereas the operator is not evaluated yet.

3 Correctness of the PCKS-machine

3.1 The P-machine

In order to establish the correctness of the PCKS-machine, we consider the P-machine, a variant of the PCKS-machine. The major difference between the P-machine and the PCKS-machine is that boxes encapsulate the continuation of the call of makeref instead of its spine. For the sake of uniformity, we also add an extra rule, called (init), which stores the final result into a distinguished location α_0. Now, a P-configuration is said to be final when α_0 contains a value.

Figure 6 shows rule (init) and a few updated rules concerning boxes; other rules can be derived easily. Lemma 1 states that the PCKS-machine and the P-machine are equivalent.

Lemma 1 Let $X \in \Lambda_u^0$, $\mathsf{eval}_{\mathsf{pcks}}(X) = \mathsf{eval}_{\mathsf{P}}(X)$. $\square$

Proof Sketch. The proof involves a translation mapping each box $\langle \mathsf{bx}\ \kappa, \alpha_b\rangle$ of the P-machine to a box $\langle \mathsf{bx}\ s_\kappa, \alpha_b\rangle$ of the PCKS-machine, where the continuation κ was replaced by its spine. Then, one can show that each transition of the P-machine can also be performed by the PCKS-machine, and vice-versa. Hence, their evaluation relations are the same.

$$b \in Box ::= \langle \mathsf{bx}\ \kappa, \alpha \rangle \qquad \text{(Box)}$$
$$Loc \;=\; \{\alpha_0\} \cup Loc_l \cup Loc_r \cup Loc_b$$

$\langle V, \langle (\textbf{initp}), (\textbf{init}) \rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \ddagger, \textbf{stop} \rangle_\tau; \sigma[\alpha_0 \leftarrow V]$ (init)

$\langle V, \langle (\gamma\ \textbf{fun}\ \mathsf{makeref}), \eta \rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \langle \mathsf{bx}\ \langle \gamma, \eta \rangle, \alpha_b \rangle, \langle \gamma, \eta \rangle\rangle_\tau; \sigma[\alpha_b \leftarrow V]$ with $\alpha_b \notin DOM(\sigma)$ (makeref)

$\langle \langle \mathsf{bx}\ \kappa, \alpha_b \rangle, \langle (\gamma\ \textbf{fun}\ \mathsf{deref}), \eta \rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \langle \mathsf{bx}\ \kappa, \alpha_b \rangle, \langle (\gamma\ \textbf{unbox}(\langle \gamma, \eta \rangle)), \eta \rangle\rangle_\tau$ (deref)

$\langle \langle \mathsf{bx}\ \kappa, \alpha_b \rangle, \langle (\gamma'\ \textbf{unbox}(\langle \gamma, \eta \rangle)), \eta' \rangle\rangle_\tau$

$\overset{cks}{\mapsto} \langle \sigma(\alpha_b), \langle \gamma, \eta \rangle\rangle_\tau$ if $\kappa \sqsupseteq \langle (\textbf{initp}), \eta' \rangle$ (unbox)

$\vdots$

and similar rules for setref!

Fig. 6. P-Machine: differences with the PCKS-machine

3.2 Translation of the P-machine

Now let us translate a configuration of the P-machine into a term of a variant of Felleisen, Hieb, and Sabry's λ_v-CS-calculi[7] [9, 29], whose syntax follows.

Definition 9 (Λ_v-CS)

M	$::= V \mid (M\ M) \mid (\mathsf{letref}\ \ \theta\ M)$	(Terms)
V	$::= c \mid x \mid (\lambda x.M) \mid (\mathsf{setref!}\ x_\sigma)$	(Values)
c	$::= a \mid d$	(Constants)
d	$::= \mathcal{A} \mid \mathsf{callcc} \mid \mathsf{makeref} \mid \mathsf{deref} \mid \mathsf{setref!} \mid void$	(Constants)
θ	$::= (\ldots (x_\sigma\ V) \ldots)$	(Local Store)
$E[\,]$	$::= [\,] \mid (E[\,]\ M) \mid (V\ E[\,])$	(Evaluation Context)

$x \in Vars, \quad x_\sigma \in AssignableVars, \quad a \in Csts$

□

The letref construct corresponds to Felleisen and Hieb's [9] ρ-notation where the store θ, a finite function represented as a set, maps assignable variables to values. The axioms of the λ_v-CS theory can be found in [30] pages 91–92.

Each (pcall M N) creates a speculative process to evaluate N. As soon as the value of M is obtained, the speculative process becomes mandatory. In order to know the number of speculative processes or speculative results in a configuration, we just need to count the number of pairs (α_m, α_n) allocated by pcall, such that α_m is empty. As a location α_n is aimed at receiving the result of a speculative process,

[7] We shall use the control axioms of Sabry and Felleisen's λ_v-CS-calculus and the state axioms of Felleisen and Hieb's λ_v-CS-calculus.

each result being computed (or already computed but unused) ends up in a location α_n, and will be passed to a continuation, thanks to ($\mathsf{ret}\text{-}1_l$). This idea is embodied in the concept of *target* [20, 21].

Definition 10 (Target) Let $\mathcal{M} \equiv \langle P, \sigma \rangle$ be a configuration of the P-Machine. A target, denoted by g, is a pair $\langle \alpha, \kappa \rangle$ containing a location α and a continuation κ. The set of targets of a configuration $\mathcal{M}$ is defined as follows.

- The pair $\langle \alpha_0, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle$ is a target of $\mathcal{M}$.
- If there are two locations α_m and α_n that were allocated by a transition pcall, such that $\sigma(\alpha_m) = \bot$, then the pair $\langle \alpha_n, \langle (\mathbf{initp}), (\kappa\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle \rangle$ is a target of $\mathcal{M}$.

Any continuation κ that appears in a configuration is related to a target:

$$\mathcal{G}_\sigma[\kappa] = g_i \quad \text{with} \quad g_i \equiv \langle \alpha_i, \kappa_i \rangle, \quad \text{such that} \quad \kappa \sqsupseteq^s_\sigma \kappa_i \text{ and } \sigma(\alpha_i) = \bot$$

By extension, first-class continuations, boxes, and processes are related to targets according to the same principle:

$$\mathcal{G}_\sigma[\langle \mathsf{co}\ \kappa \rangle] = \mathcal{G}_\sigma[\kappa], \quad \mathcal{G}_\sigma[\langle \mathsf{bx}\ \kappa, \alpha \rangle] = \mathcal{G}_\sigma[\kappa], \quad \mathcal{G}_\sigma[\langle M, \kappa \rangle_\tau] = \mathcal{G}_\sigma[\kappa]$$

□

We are now ready to define the translation of a configuration of the P-machine. The essence of the translation is to associate each active process (and each unused speculative result) with a term of the λ_v-CS-calculus, composed of a local store (within a letref) and an initial continuation. Definition 11 relies on three mutually recursive translation functions appearing in Definitions 12, 13, and 14.

Definition 11 (Machine Translation) Let $\mathcal{M} \equiv \langle P, \sigma \rangle$ be a P-configuration. Let $\{g_1, \ldots, g_n\}$ be the targets of the machine, and let $\{k_1, \ldots, k_n\}$ be their associated fresh variables such that ρ_g maps each target g_i to a variable k_i. Let $\{b_1, \ldots, b_m\}$ be the boxes allocated in configuration $\mathcal{M}$, and let $\{x_{b_1}, \ldots, x_{b_m}\}$ be their associated fresh assignable variables, such that ρ_b maps each box b_i to an assignable variable x_{b_i}. The translation of the machine $\mathcal{M}$ is a set of terms, obtained as follows for each target g_i.

1. For any active process $\langle M, \kappa \rangle_\tau$ associated with target $g_i = \mathcal{G}_\sigma[\kappa]$, there exists a term $\mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ \mathcal{P}^{\rho_g,\rho_b}_\sigma[\![\langle M, \kappa \rangle]\!])$, with $\theta = (\ldots(x_{b_j}\ \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![\sigma(\alpha_{b_j})]\!])\ldots)$, for each box $b_j = \langle \mathsf{bx}\ \kappa_j, \alpha_{b_j} \rangle$ such that $\rho_b(b_j) = x_{b_j}$ and $\mathcal{G}_\sigma[b_j] = g_i$.
2. For any non empty location α_i associated with target g_i, there exists a term $\mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![\sigma(\alpha_i)]\!])$, with $\theta = (\ldots(x_{b_j}\ \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![\sigma(\alpha_{b_j})]\!])\ldots)$, for each box $b_j = \langle \mathsf{bx}\ \kappa_j, \alpha_{b_j} \rangle$ such that $\rho_b(b_j) = x_{b_j}$ and $\mathcal{G}_\sigma[b_j] = g_i$.

□

Definition 12 (Process Translation) $\mathcal{P}^{\rho_g,\rho_b}_\sigma[\![\langle M, \kappa \rangle_\tau]\!] = \mathcal{C}^{\rho_g,\rho_b}_\sigma[\![\kappa, \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![M]\!]]\!]$ □

Definition 13 (Term Translation)

$$\begin{aligned}
\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![(\mathsf{pcall}\ M\ N)]\!] &= (\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![M]\!]\ \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![N]\!]) \\
\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![(M\ N)]\!] &= (\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![M]\!]\ \mathcal{T}^{\rho_g,\rho_b}_\sigma[\![N]\!]) \\
\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![\lambda x.M]\!] &= \lambda x.\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![M]\!] \\
\mathcal{T}^{\rho_g,\rho_b}_\sigma[\![c]\!] &= c
\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![x]\!] &= x \\
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{co}\ \kappa\rangle]\!] &= \lambda v.\mathcal{A}\ (k_i\ \mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\kappa, v]\!]) \text{ with } k_i = \rho_g(g_i) \\
&\quad \text{if } g_i \equiv \mathcal{G}[\langle \mathsf{co}\ \kappa\rangle] \text{ and with } v \notin FV(\mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\kappa, [\ ]]\!]) \\
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{bx}\ \kappa_b, \alpha_b\rangle]\!] &= x_b \quad \text{if } \mathcal{G}[\langle \mathsf{bx}\ \kappa_b, \alpha_b\rangle] = \{x_b, g_b\} \\
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{sco}\ \langle \mathsf{co}\ \kappa\rangle\ V\rangle]\!] &= (\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{co}\ \kappa\rangle]\!]\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![V]\!]) \\
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{subx}\ \kappa\ b\rangle]\!] &= (\lambda v.\ \mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\mathit{suffix}_\sigma(\kappa), v]\!])\ (\mathsf{deref}\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![b]\!]) \\
\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\langle \mathsf{ssbx}\ \kappa\ b\ V\rangle]\!] &= (\mathsf{begin}\ \ (\mathsf{setref!}\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![b]\!]\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![V]\!]) \\
&\qquad\qquad \mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\mathit{suffix}_\sigma(\kappa), \mathit{void}]\!]\ \ )
\end{aligned}
$$

□

Definition 14 (Continuation Translation)

$$\mathcal{C}_\sigma^{\rho}[\![\langle\gamma,\eta\rangle, M]\!] = \mathcal{N}_\sigma^{\rho_g,\rho_b}[\![\eta, \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,M]\!]]\!]$$

$$
\begin{aligned}
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\gamma\ \mathbf{arg}\ N),\eta,M]\!] &= \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,(M\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![N]\!])]\!] \\
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\gamma\ \mathbf{fun}\ V),\eta,M]\!] &= \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,(\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![V]\!]\ M)]\!] \\
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\gamma\ \mathbf{left}(\alpha_m,\alpha_n,N)),\eta,M]\!] &= \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,(M\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![N]\!])]\!] \\
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\gamma\ \mathbf{unbox}(\kappa)),\eta,M]\!] &= \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,((\lambda v.\mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\kappa_d,v]\!])\ (\mathsf{deref}\ M))]\!] \\
&\quad \text{with } \kappa_d = \{\kappa \setminus \langle\gamma,\eta\rangle\} \\
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\gamma\ \mathbf{setbox}(\kappa,V)),\eta,M]\!] &= \Gamma_\sigma^{\rho_g,\rho_b}[\![\gamma,\eta,(\mathsf{begin}\ \ (\mathsf{setref!}\ \mathcal{T}_\sigma^{\rho_g,\rho_b}[\![V]\!]\ M)\ ]\!] \\
&\qquad\qquad \mathcal{C}_\sigma^{\rho_g,\rho_b}[\![\kappa_d, \mathit{void}]\!]) \\
&\quad \text{with } \kappa_d = \{\kappa \setminus \langle\gamma,\eta\rangle\} \\
\Gamma_\sigma^{\rho_g,\rho_b}[\![(\mathbf{initp}),\eta,M]\!] &= M
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N}_\sigma^{\rho_g,\rho_b}[\![(\mathbf{init}),M]\!] &= M \\
\mathcal{N}_\sigma^{\rho_g,\rho_b}[\![(\kappa\ \mathbf{right}(\alpha_m,\alpha_n)),M]\!] &= \text{ if } \sigma(\alpha_m)\neq\bot \text{ then } \mathcal{C}_\sigma^{\rho}[\![\kappa,(\mathcal{T}_\sigma^{\rho_g,\rho_b}[\![\sigma(\alpha_m)]\!]\ M)]\!] \text{ else } M
\end{aligned}
$$

□

The difference of two continuations κ_1 and κ_2 is defined if κ_2 is a prefix of κ_1; the difference is the part of the continuation κ_1 that should be "appended" to κ_2 to obtain κ_1.

Definition 15 (Continuation Difference)

$$\{\langle\gamma_1,\eta_1\rangle \setminus \langle\gamma_2,\eta_2\rangle\} = \begin{cases} \langle\{\gamma_1 \setminus_\gamma \gamma_2\}, (\mathbf{init})\rangle & \text{if } \eta_1 \equiv \eta_2 \\ \langle\gamma_1, (\{\kappa_1 \setminus \langle\gamma_2,\eta_2\rangle\}\ \mathbf{right}(\alpha_m,\alpha_n))\rangle & \text{if } \eta_1 \equiv (\kappa_1\ \mathbf{right}(\alpha_m,\alpha_n)) \end{cases}$$

$$
\begin{aligned}
\{\gamma_1 \setminus_\gamma \gamma_2\} &= (\mathbf{initp}) \quad \text{if } \gamma_1 \equiv \gamma_2 \\
\{(\gamma\ \mathbf{left}(\alpha_m,\alpha_n,N)) \setminus_\gamma \gamma'\} &= (\{\gamma \setminus_\gamma \gamma\}\ \mathbf{left}(\alpha_m,\alpha_n,N)) \\
\{(\gamma\ \mathbf{fun}\ V) \setminus_\gamma \gamma'\} &= (\{\gamma \setminus_\gamma \gamma\}\ \mathbf{fun}\ V) \\
\{(\gamma\ \mathbf{arg}\ N) \setminus_\gamma \gamma'\} &= (\{\gamma \setminus_\gamma \gamma\}\ \mathbf{arg}\ N) \\
\{(\gamma\ \mathbf{unbox}(\kappa)) \setminus_\gamma \gamma'\} &= (\{\gamma \setminus_\gamma \gamma\}\ \mathbf{unbox}(\kappa)) \\
\{(\gamma\ \mathbf{setbox}(\kappa,V)) \setminus_\gamma \gamma'\} &= (\{\gamma \setminus_\gamma \gamma\}\ \mathbf{setbox}(\kappa,V))
\end{aligned}
$$

□

The suffix of a continuation κ is defined as the difference of κ and the continuation of its target, formally defined as follows.

Definition 16 (Continuation Suffix) $\text{suffix}_\sigma(\kappa) = \{\ \kappa\ \setminus\ \kappa_i\ \}$ if $g_i \equiv \langle \alpha_i, \kappa_i \rangle = \mathcal{G}_\sigma[\kappa]$. □

We now have the tools to prove the equivalence of the P-machine and the λ_v-CS-calculus. The correctness of the PCKS-machine will follow by Lemma 1.

3.3 Equivalence of the P-machine and the λ_v-CS-calculus

The goal of this section is to prove that the evaluation functions of the P-machine and the λ_v-CS-calculus are the same. We proceed in two steps (i) we establish that if the P-machine gives an answer A, so does the λ_v-CS-calculus, (ii) we prove that when the λ_v-CS-calculus gives an answer, the P-machine returns the same answer.

As in [21], we define a translation function $\mathcal{S}[\![X]\!]$, which returns the sequential version of a program X by removing all annotations pcall. Such a function is essentially the identity function, except for (pcall M N), which is translated in $(M\ N)$.

The first proposition is established by proving that each transition of the P-machine corresponds to a provable equality in the λ_v-CS-theory. Intuitively, Lemma 2 states that two terms corresponding to the same target are equal in the theory, up to the substitution of free continuation variables.

Lemma 2 Let $\mathcal{M}_t \equiv \langle P_t, \sigma_t \rangle$ be a P-configuration obtained after t transitions. Let $\{g_1, \ldots, g_{n_t}\}$ be the targets of the machine. Let e_{i_t} be the term that appears in the translation of $\mathcal{M}$ for target g_{i_t}.

Let $\mathcal{M}_s \equiv \langle P_s, \sigma_s \rangle$ be a configuration obtained after s transitions, with $s \leq t$. Let e_{i_s} be the term associated with target g_{i_s} in the translation of $\mathcal{M}_s$, such that $g_{i_s} \equiv g_{i_t}$. We have that

$$\lambda_v\text{-}CS \vdash e_{i_s}\{c_u/k_u\}^* \ = \ e_{i_t}$$

for all variables k_u, such that $\rho_s(g_u) = k_u$,$\rho_t(g_u) = \perp$,$\rho_s(g_v) = \rho_t(g_v) = k_v$, and $c_u = \mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\langle \mathsf{co}\ \kappa_u \rangle]\!]$. □

Proof. See Appendix A

From Lemma 2, we can deduce that if the evaluation relation of the P-machine returns an answer, the λ_v-CS-calculus will also return the same answer.

Lemma 3 Let $X \in \Lambda_u^0$, if $\mathsf{eval}_\mathsf{P}(X) = A$, then $\mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!]) = A$. □

Proof. If $\mathsf{eval}_\mathsf{P}(X) = A$, then there exists an initial configuration $\mathcal{M}_i \equiv \langle \{\langle X, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle_\tau\}, \sigma_i \rangle$, and a final configuration $\mathcal{M}_f \equiv \langle P_f, \sigma_f \rangle$, such that $\text{Unload}[\sigma_f(\alpha_0)] = A$, and such that $\mathcal{M}_i \mapsto_P^* \mathcal{M}_f$. By Lemma 2, with $\mathcal{M}_s$ the initial configuration, and $\mathcal{M}_t$ the final one, we have that

$$\lambda_v\text{-}CS \vdash \mathcal{S}[\![X]\!] = (\mathsf{letref}\ \theta\ \mathcal{T}_{\sigma_f}^{\rho_{gf},\rho_{bf}}[\![\sigma_f(\alpha_0)]\!])$$

if we consider the terms associated to target $\langle \alpha_0, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle$ in $\mathcal{M}_s$ and $\mathcal{M}_t$. So $\mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!]) = A$.

In order to establish the opposite proposition, we rely on the fact that the λ_v-CS-calculus and the CKS-machine [6, 8] have the same evaluation functions. Let us now prove that the P-machine is able to simulate the CKS-machine using a mandatory reduction strategy.

We define a subrelation of the transition relation of the P-machine, which only allows the mandatory process to perform transitions. This relation defines a sequential evaluation function of the P-machine and is formally stated as follows.

Definition 17 ($\mapsto_P^S$) Let $\mathcal{M}_i$ and $\mathcal{M}_j$ be two configurations of the P-machine,

$$\mathcal{M}_i \equiv \langle P_i, \sigma_i \rangle, \quad \mathcal{M}_j \equiv \langle P_j, \sigma_j \rangle.$$

There exists a mandatory transition between $\mathcal{M}_i$ and $\mathcal{M}_j$, written $\mathcal{M}_i \mapsto_P^S \mathcal{M}_j$, if there exists a process p_k of the form $\langle M, \kappa \rangle_{\tau_k}$ in P_i with $\kappa \sqsupseteq_{\sigma_i}^s \langle (\mathbf{initp}), (\mathbf{init}) \rangle$, and a set of processes P_{p_k}, such that $\langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle$, and $P_j \equiv P_i \setminus \{p_k\} \cup P_{p_k}$. We write $\mapsto_P^{S\,*}$ to denote the reflexive, transitive closure of $\mapsto_P^S$. □

The evaluation function that schedules the mandatory process is defined as follows.

Definition 18 ($\mathsf{eval}_\mathsf{P}^\mathsf{S}$) Let X and A be a program and an answer. We have that $\mathsf{eval}_\mathsf{P}^\mathsf{S}(X) = A$ if there exists a final configuration $\mathcal{M}_f \equiv \langle P_f, \sigma_f \rangle$ such that $\sigma_f(\alpha_0)$ is a value with $\text{Unload}[\sigma_f(\alpha_0)] = A$, and such that, for the initial configuration $\mathcal{M}_{init} \equiv \langle \{ \langle M, \langle (\mathbf{initp}), (\mathbf{init}) \rangle \rangle_{\tau_0} \}, \sigma_i \rangle$,

$$\mathcal{M}_{init} \mapsto_P^{S\,*} \mathcal{M}_f.$$

□

Now we can establish that the P-machine is able to simulate the CKS-machine.

Lemma 4 Let $X \in \Lambda_u^0$, $\mathsf{eval}_\mathsf{P}^\mathsf{S}(X) = \mathsf{eval}_\mathsf{cks}(\mathcal{S}[\![X]\!])$. □

Proof Sketch. The proof [20] involves a translation that maps every state of the P-machine (evaluated according to the sequential evaluation function) into a state of the CKS-machine.

We are now ready to prove the soundness of the P-machine with respect to the λ_v-CS-calculus.

Theorem 1 *Let* $X \in \Lambda_u^0$, $\mathsf{eval}_\mathsf{P}(X) = \mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!])$. □

Proof. We have established that if $\mathsf{eval}_\mathsf{P}(X) = A$ then $\mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!]) = A$ in Lemma 3. Let us now prove that if $\mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!]) = A$, then $\mathsf{eval}_\mathsf{P}(X) = A$.

If $\mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!]) = A$, then using Felleisen and Friedman's Theorem 3 [8], Corollary 5.8 and Theorem 5.1 [6], we conclude that $\mathsf{eval}_\mathsf{cks}(\mathcal{S}[\![X]\!]) = A$. Using Lemma 4, we deduce that $\mathsf{eval}_\mathsf{P}^\mathsf{S}(X) = A$. Since the relation $\mapsto_P^S$ is a subset of $\mapsto_P$, we conclude that $\mathsf{eval}_\mathsf{P}(X) = A$.

Hence, we derive the soundness of the PCKS-machine.

Theorem 2 *Let* $X \in \Lambda_u^0$, $\mathsf{eval}_\mathsf{pcks}(X) = \mathsf{eval}_{v\text{-}CS}(\mathcal{S}[\![X]\!])$. □

Proof. By transitivity from Lemma 1 and Theorem 1.

In addition, we establish that the PCKS-machine is faithful to the CKS-machine.

Theorem 3 *Let* $X \in \Lambda_u^0$, $\mathsf{eval}_\mathsf{pcks}(X) = \mathsf{eval}_\mathsf{cks}(\mathcal{S}[\![X]\!])$. □

Proof. From Theorem 2, and Felleisen and Friedman's Theorem 3 [8], Corollary 5.8 and Theorem 5.1 [6].

4 Discussion and Related Work

The first contribution of this paper is a reorganisation scheme of the continuation of a process in two components[8]. A major consequence of this reorganisation is that sequential programs can be executed on the PCKS-Machine as efficiently[9] as on a sequential CK(S)-Machine [6, 7]. In addition, the invocation of a continuation is also fast in parallel programs for two reasons. On the one hand, the side-condition of rule invoke can be implemented efficiently because the test $\kappa_1 \sqsupseteq \kappa_2$ only requires to compare locations (and not their contents in shared memory). Furthermore, this test is bounded by the length of the continuation κ_1, which is the number of nested parallel calls that were performed when the continuation was captured. In the case of coarse grain parallelism, this number is relatively small. On the other hand, rule prune-invoke is able to delete a local continuation in a single step, without considering the codes that it contains.

The second contribution is the extension of the PCKS-Machine to side-effects. Again here, we see the benefits of the reorganisation of the continuation since similar patterns can be observed for the rules for continuations, dereferences, and assignments.

The reader might wonder whether parallelism can really exist for an annotated program with side-effects and continuations. The condition to perform an effect (continuation invocation or side-effect) is that the current continuation must be a prefix of the "effect continuation" (i.e. the continuation to invoke, or the continuation that existed when a box was created). If this condition is not satisfied, a pruning rule should be used, and possibly a suspend rule which sequentialises the program.

Sequentialisations of expressions can be avoided if effects remain "local"; by this, we mean that if an expression creates a continuation (or box) and uses it internally, it can be evaluated in parallel with other expressions without synchronising. Such a property is nothing else but the abstraction principle: outside a function, we cannot observe synchronisations stemming from effects performed on objects created locally by the function.

Now let us examine some details of the PCKS-Machine which could be improved to provide more parallelism. A strong assumption in the machine is that CKS-transitions (Definition 1) must be performed atomically. This is a high-level of atomicity: for instance, rule suspend-invoke performs two accesses to shared memory locations and a test $\sqsupseteq$. Even though the semantics proposed in [18] does not deal with side-effects and sequentialises computations when continuations are invoked in an upward way, some ideas can hold our attention. In particular, each pair of locations (α_m, α_n) allocated by rule pcall is considered as a critical section. Therefore, several transitions can be performed together if they concern different critical sections.

There is a source of centralisation in rules pcall and makeref, where a unique location is allocated. If several processors run in parallel, they will have to consult each other when such rules are executed. However, allocation of fresh locations can

[8] In [18], two continuations were also used but for another reason. There, continuations were represented as regular abstractions. Since abstractions hide their bodies, introspective predicate like, "is a prefix of", are difficult to implement. A second continuation uncovering some internal information was used to compute such a predicate.

[9] In a purely sequential program, the invoked continuation is necessarily reified by the current process, which is the single one. Hence, the test $\langle \gamma, \eta \rangle \sqsupseteq \langle (\mathbf{initp}), \eta' \rangle$ is quickly performed because η and η' are the same and any γ is always an extension of $(\mathbf{initp})$.

be decentralised if we assume that each processor has a unique name and that each processor is responsible of a pool of shared memory. We just have to define a location as a pair, processor name – local address, which uniquely names a location in the addressable space of shared memory. With such a scheme, locations can be allocated locally on each processor.

We could easily add an environment [6], which would give more a realistic implementation than substitution. We have not done it here so as to reduce the rules length.

We have proposed a semantics of Scheme programs annotated for parallelism by `fork` and `pcall`. A static analysis could certainly improve the parallel execution of programs. Flanagan and Felleisen [10] use the set-based program analysis [13] to perform a "touch optimisation", that is, to remove the touch operation, when it can be predicted at compile-time that the touch argument will be different from a placeholder. Two similar optimisations could be performed here. If we can predict that an application site only invokes continuations that are extensions of the current synchronising continuation, we can avoid to perform the test that appears in rule invoke. Similarly, if we can predict that the argument of a deref (or the first argument of a setref!) is a box whose spine is an extension of the current synchronising continuation, deref and setref! could be replaced by their unsafe versions. Furthermore, two processes reading a location do not need to synchronise; expressions should be sequentialised only when read/write or write/write conflicts can occur. A possible output of a static analysis could be Jade's access declarations. Rinard and Lam's Jade [28, 27] is a declarative language designed to express access information about data shared by tasks in a coarse-grain parallel program. The semantics guarantees that the parallel and sequential executions of Jade[10] programs compute the same results.

Now, let us compare our approach with other parallel implementations of Scheme or Lisp. MultiLisp [11] certainly is the implementation that philosophically is the closest to our. It is also based on annotations that specify which expressions can be evaluated in parallel. The annotation is called `future` and its semantics in a purely functional language was stated by Flanagan and Felleisen [10]. Preserving the transparency of `future` in the presence of first-class continuations has been problematic [17, 11]: Katz and Weise [16] proposed an evaluation schema that was implemented by Feeley [4]. The case of `future` and side-effects was also studied by Tinker and Katz [31] in their Paratran system. But the combination of first-class continuations and side-effects in a parallel language based on the annotation `future` has never been investigated before.

Tinker, Katz, and Weise's [31, 16] definitions of `future` have in common that they perform effects *optimistically* or *speculatively*. In Paratran, side-effects are performed optimistically: a run-time system detects data dependency violations and is able to correct them by restoring a previous state by a roll-back mechanism. Similarly, continuations in [16] are speculatively invoked; a complementary mechanism, called legitimacy link, is able to detect which result is compatible with the sequential semantics. This approach has some runtime cost (due to unrolling and dependencies detections); moreover, it can generate too many speculative computations. Our approach is totally opposite to theirs because it consists in performing effects *non-speculatively*, that is, only when these effects are guaranteed not to infringe the sequential semantics.

[10] Rinard and Lam deal with an imperative language without closures or control operators; their approach should first be extended to a Scheme-like language.

Halstead [11, page 19] proposed three criteria for the semantics of parallel constructs and continuations in parallel Scheme. Not only does our semantics satisfy them, but also can these criteria be extended to side-effects.

Queinnec [24] proposed a coherency protocol for shared mutable variables in a distributed dialect of Scheme, called ICSLAS [25]. The protocol ensures that a side-effect perceived by a thread cannot be ignored by the threads of its continuations. However, Queinnec's goal differs from ours because parallelism is not provided by annotations for parallelism, and the sequential result of a program is not expected to be preserved.

Ito, Matsui, and Seino [14, 15] give the semantics of PaiLisp, a parallel dialect of Lisp with first-class continuations. Their goal is also different from ours because their primitives for parallelism add expressiveness to the sequential core, like for instance *parallel-or*.

5 Conclusion

We have presented the PCKS-Machine, a parallel abstract machine that evaluates Scheme programs annotated by `pcall` and `fork`. The PCKS-Machine is proved to be correct since annotated expressions are observationally equivalent to their non-annotated counterparts.

The PCKS-Machine has both theoretical and practical impacts: it is a semantics of annotation-based parallel Scheme, and it is a guideline to implement a parallel version of Scheme with such annotations. As far as parallel programs are concerned, the PCKS-Machine supports both first-class continuations and side-effects. Parallelism is provided in the presence of such effect when they remain local.

In future work, we plan to study the annotation `future` using the `pcall` and `set!` constructs as Queinnec [25, 11]. Several static analysis of parallel programs have also been suggested in Section 4 in order to improve the efficiency of effects.

6 Acknowledgement

Most of the results of this paper were obtained while the first author was at the University of Liège; now he is supported by the Engineering and Physical Sciences Research Council, grant GR/K30773. We are grateful to the referees for their comments.

A Proof of Lemma 2

We proceed by induction on the length of the reduction and by case on the last transition $\mathcal{M}_s \mapsto_P \mathcal{M}_t$. We also prove the following invariants. For any process of the form $\langle V, \langle(\gamma' \; \mathbf{unbox}(\langle\gamma,\eta\rangle)), \eta'\rangle\rangle_\tau$, the following properties hold:

$$\langle\gamma,\eta\rangle \sqsupseteq^s_\sigma \langle\gamma',\eta'\rangle \qquad (1)$$

$$\mathcal{C}^{\rho_g,\rho_b}_\sigma[\![\langle\gamma,\eta\rangle,[\;]]\!] = \mathcal{N}^{\rho_g,\rho_b}_\sigma[\![\eta', \Gamma^{\rho_g,\rho_b}_\sigma[\![\gamma',\eta',\mathcal{C}^{\rho_g,\rho_b}_\sigma[\![\{\langle\gamma,\eta\rangle \setminus \langle\gamma',\eta'\rangle\},[\;]]\!]]\!]]\!] \qquad (2)$$

Similar invariants also hold for the continuation code **setbox**.

We only consider typical transitions. (capture), (invoke), (prune-invoke) deal with continuations; (makeref), (unbox), (prune-unbox), and (susp-unbox) concern boxes; (stop_l) uses the induction on the length of the reduction. The other cases are either similar to the ones presented or can be found in [20].

(capture) We have that $\rho_{g_s} = \rho_{g_t}$ and $\sigma_s = \sigma_t$. Let g_i be the target of process τ, with $\rho_{g_s}(g_i) = \rho_{g_t}(g_i) = k_i$. There exist a store θ and an evaluation context

$E[\,]$, such that

$$e_{is} \equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E[\mathsf{callcc}\ \ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket])$$
$$e_{it} \equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E[\mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket V\rrbracket\ \ \mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket\langle \mathsf{co}\ \langle\gamma,\eta\rangle\rangle\rrbracket])$$

with $E[\,] = \mathcal{N}_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\eta, \Gamma_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\gamma,\eta,[\,]\rrbracket\rrbracket$. Furthermore, $\mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket\langle \mathsf{co}\ \langle\gamma,\eta\rangle\rangle\rrbracket \equiv \lambda v.\mathcal{A}(k_i\ \mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket\langle\gamma,\eta\rangle, v\rrbracket) \equiv \lambda v.\mathcal{A}(k_i\ E[v])$ because the evaluation context $E[\,]$ was defined as $E[\,] \equiv \mathcal{C}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket\langle\gamma,\eta\rangle,[\,]\rrbracket \equiv \mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket\langle\gamma,\eta\rangle,[\,]\rrbracket$.
Therefore $\lambda_v\text{-}CS \vdash e_{is} = e_{it}$ by (C_{lift}), (C_{idem}), and $(\rho\mathcal{C})$ [9].

(invoke) We have that $\rho_{g_s} = \rho_{g_t}$ and $\sigma_s = \sigma_t$. Let $g_i \equiv \langle\alpha_i,\kappa_i\rangle$ be the target of process τ, with $\rho_{g_s}(g_i) = \rho_{g_t}(g_i) = k_i$. As $\langle \mathsf{co}\ \langle\gamma,\eta\rangle\rangle$ is accessible by the process τ, the target $g_j \equiv \langle\kappa_j,\alpha_j\rangle = \mathcal{G}_\sigma[\langle \mathsf{co}\ \langle\gamma,\eta\rangle\rangle]$ is such that $\kappa_i \sqsupseteq \kappa_j$. By hypothesis, we have $\langle\gamma,\eta\rangle \sqsupseteq \kappa_i$. Therefore, we have $\langle\gamma,\eta\rangle \sqsupseteq \kappa_i \sqsupseteq \kappa_j$ and $\langle\gamma,\eta\rangle \sqsupseteq_{\sigma_s}^{s} \kappa_j$, which implies $\langle\gamma,\eta\rangle \sqsupseteq_{\sigma_s}^{s} \kappa_i \sqsupseteq_{\sigma_s}^{s} \kappa_j$. Since κ_i and κ_j are both continuations of targets then $\kappa_i \equiv \kappa_j$. Hence, there exist two evaluation contexts $E_s^1[\,]$ and $E_s^2[\,]$, such that:

$$\begin{aligned}
e_{is} &\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E_s^1[(\lambda v.\mathcal{A}k_i(E_s^2[v]))\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket])\\
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E_s^1[\mathcal{A}k_i(E_s^2[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket])]) \text{ by } (\beta_v)\\
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E_s^2[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket]) \text{ by } (C_{cur})\\
e_{it} &\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ \mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket\langle\gamma,\eta\rangle, \mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket V\rrbracket\rrbracket)
\end{aligned}$$

$$\begin{aligned}
\text{with}\quad E_s^1[\,] &= \mathcal{N}_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\eta', \Gamma_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\gamma',\eta',[\,]\rrbracket\rrbracket\\
E_s^2[\,] &= \mathcal{N}_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\eta, \Gamma_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\gamma,\eta,[\,]\rrbracket\rrbracket
\end{aligned}$$

So we have $\lambda_v\text{-}CS \vdash e_{is} = e_{it}$.

(prune-invoke) We have that $\rho_{g_s} = \rho_{g_t}$ and $\sigma_s = \sigma_t$. Let g_i be the target of process τ with $\rho(g_i) = k_i$. Let $g_j \equiv \langle\alpha_j,\kappa_j\rangle = \mathcal{G}_{\sigma_s}[\langle\gamma,\eta\rangle]$ be the target of the first-class continuation, with $\rho_{g_s}(g_j) = k_j$. We have that $k_j \not\equiv k_i$ because $\langle\gamma,\eta\rangle \not\sqsupseteq \langle(\mathbf{initp}),\eta_i\rangle$, with $\eta_i = (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$. There exist a store θ and three evaluation contexts $E_s^1[\,]$, $E_s^2[\,]$, and $E_s^3[\,]$ such that:

$$\begin{aligned}
e_{is} \equiv\ &\mathsf{callcc}\lambda k_i.\,(\mathsf{letref}\ \theta\\
&\qquad E_s^1[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket\sigma_s(\alpha_m)\rrbracket\ \ E_s^2[(\lambda v.\mathcal{A}(k_j\ E_s^3[v]))\ \ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket]])\\
e_{it} \equiv\ &\mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E_t^1[(\lambda v.\mathcal{A}(k_j\ E_t^3[v]))\ \ \mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}\llbracket V\rrbracket])
\end{aligned}$$

$$\begin{aligned}
\text{with}\quad E_s^1[\,] &\equiv \mathcal{C}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket\langle\gamma'',\eta'\rangle,[\,]\rrbracket \equiv E_t^1[\,]\\
E_s^2[\,] &\equiv \Gamma_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\gamma',\eta_i,[\,]\rrbracket\\
E_s^3[\,] &\equiv \Gamma_{\sigma}^{\rho_{g_s},\rho_{b_s}}\llbracket\gamma,\eta,[\,]\rrbracket \equiv E_t^3
\end{aligned}$$

We have that $\lambda_v\text{-}CS \vdash e_{is} = e_{it}$ by (C_{cur}), (β_v), $(abort)$. Note that we use the fact that the continuation is represented by an abortive function. If we had used Sabry and Felleisen's representation $(\lambda v.k_jE_s^3[v])$, it would have been difficult to detect the use of a continuation because k_j can be free in e_i.

(makeref) We have that $\rho_{g_s}{=}\rho_{g_t}$, and $\rho_{bt}{=}\rho_{bs}[b \to x_b]$ with $b \notin Dom(\rho_{bs})$, and $b \equiv \langle \mathsf{bx}\ \langle\gamma,\eta\rangle,\alpha\rangle$. Let g_i be the target of process τ, with $\rho_{g_s}(g_i) = \rho_{g_t}(g_i) = k_i$. Therefore, the target of the box b defined as $\mathcal{G}_{\sigma_s}[b]$ is also g_i. There exists an evaluation context $E[\,]$ such that:

$$\begin{aligned}
e_{is} &\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_s\ E[(\mathsf{makeref}\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket)])\\
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_s\ E[(\mathsf{letref}\ ((x_b\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket))x_b)]) \text{ by } (\mathsf{makeref})\\
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_t\ E[x_b]) \text{ by } (\mathsf{letref}_{lift}), \text{ with } \theta_t = \theta_s \cup \{(x_b\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}\llbracket V\rrbracket)\}\\
&\equiv e_{it}
\end{aligned}$$

(deref) We have ${\rho_g}_s={\rho_g}_t$, ${\rho_b}_s={\rho_b}_t$, and $\sigma_s = \sigma_t$. Let b be the box $\langle \mathsf{bx}\ \kappa, \alpha\rangle$, with ${\rho_g}_s(b) = {\rho_g}_t(b) = x_b$. Let g_i be the target of process τ, with ${\rho_g}_s(g_i) = {\rho_g}_t(g_i) = k_i$. There exists an evaluation context $E[\,]$ such that

$$\begin{aligned} e_{i\,s} &\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E[(\mathsf{deref}\ x_b)]) \\ e_{i\,t} &\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E[(\lambda v.v)\ (\mathsf{deref}\ x_b)]) \end{aligned}$$

So we have $\lambda_v\text{-}CS \vdash e_{i\,s} = e_{i\,t}$ by (β_Ω).
The invariants 1 and 2 are satisfied because $\gamma = \gamma'$ and $\eta = \eta'$.

(unbox) We have ${\rho_b}_s={\rho_b}_t$, ${\rho_g}_s={\rho_g}_t$, and $\sigma_s = \sigma_t$. Let b be the box $\langle \mathsf{bx}\ \kappa, \alpha\rangle$, with ${\rho_g}_s(b) = {\rho_g}_t(b) = x_b$. Let g_i be the target of the process τ with ${\rho_g}_s(g_i) = k_i$. The target of b is $\mathcal{G}_{\sigma_s}[b] = \mathcal{G}_{\sigma_s}[\kappa] = \mathcal{G}_{\sigma_s}[\langle(\mathbf{initp}), \eta'\rangle] = g_i$
There exist a store θ and two evaluation contexts $E^1_s[\,]$ and $E^2_s[\,]$ such that:

$$e_{i\,s} \equiv (\mathsf{callcc}\lambda k_i.\ (\mathsf{letref}\ \theta\ E^1_s[(\lambda v E^2_s[v])\ (\mathsf{deref}\ x_b)]))$$

with $\theta = (\ldots(x_b\ \mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\sigma_s(\alpha)]\!])\ldots)$, and where

$$\begin{aligned} E^1_s[\,] &= \mathcal{N}^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\eta', \Gamma^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\gamma', \eta', [\,]]\!]]\!] \\ E^2_s[\,] &= \mathcal{C}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\{\langle\gamma,\eta\rangle \setminus \langle\gamma',\eta'\rangle\}, [\,]]\!] \end{aligned}$$

There exists an evaluation context $E^1_t[\,]$ such that:

$$e_{i\,t} \equiv (\mathsf{callcc}\lambda k_i.\ (\mathsf{letref}\ \theta\ E^1_t[\mathcal{T}^{\rho_{gt},\rho_{bt}}_{\sigma_t}[\![\sigma_t(\alpha)]\!]]))$$

where $E^1_t[\,] = \mathcal{N}^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\eta, \Gamma^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\gamma, \eta, [\,]]\!]]\!]$. So we have that $\lambda_v\text{-}CS \vdash e_{i\,s} = e_{i\,t}$ because the invariants 1, 2 guarantee that $E^1_s[E^2_s[\,]] = \mathcal{C}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\langle\gamma,\eta\rangle, [\,]]\!] = E^1_t[\,]$.

(prune-unbox) We have ${\rho_b}_s={\rho_b}_t$, ${\rho_g}_s={\rho_g}_t$, and $\sigma_s = \sigma_t$. Let b be the box $\langle \mathsf{bx}\ \kappa, \alpha\rangle$, with ${\rho_g}_s(b) = {\rho_g}_t(b) = x_b$. Let g_i be the target of the process τ with ${\rho_g}_s(g_i) = k_i$. There exist three evaluation contexts $E^1_s[\,]$, $E^2_s[\,]$, and $E^3_s[\,]$ such that:

$$\begin{aligned} e_{i\,s} &\equiv (\mathsf{callcc}\lambda k_i.\ (\mathsf{letref}\ \theta\ E^1_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\sigma_s(\alpha_m)]\!]\ E^2_s[(\lambda r.E^3_s[r])\ (\mathsf{deref}\ x_b)]])) \\ \text{where}\ \ E^1_s[\,] &= \mathcal{C}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\langle\gamma'',\eta'\rangle, [\,]]\!] = \mathcal{N}^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\eta', \Gamma^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\gamma'', \eta', [\,]]\!]]\!] \\ E^2_s[\,] &= \Gamma^{{\rho_g}_s,{\rho_b}_s}_\sigma[\![\gamma', \eta_1, [\,]]\!] \\ E^3_s[\,] &= \mathcal{C}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\{\langle\gamma,\eta\rangle \setminus \langle\gamma',\eta_1\rangle\}, [\,]]\!] \end{aligned}$$

with $\eta_1 = (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$. Similarly, there exist three evaluation contexts $E^1_t[\,]$, $E^2_t[\,]$, and $E^3_t[\,]$

$$\begin{aligned} e_{i\,t} &\equiv (\mathsf{callcc}\lambda k_i.\ (\mathsf{letref}\ \theta\ E^1_t[E^2_t[(\lambda r.E^3_t[r])\ (\mathsf{deref}\ x_b)]])) \\ \text{where}\ \ E^1_t[\,] &= \mathcal{N}^{{\rho_g}_t,{\rho_b}_t}_\sigma[\![\eta', [\,]]\!] \\ E^2_t[\,] &= \Gamma^{{\rho_g}_t,{\rho_b}_t}_\sigma[\![\gamma'', \eta', [\,]]\!] \\ E^3_t[\,] &= \mathcal{C}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\{\langle\gamma,\eta\rangle \setminus \langle\gamma'',\eta'\rangle\}, [\,]]\!] \end{aligned}$$

So, we have $E^1_s[\,] = E^1_t[E^2_t]$ and $E^3_t[\,] = \mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![\sigma_s(\alpha_m)]\!]\ E^2_s[E^3_s[\,]]$. It results that $\lambda_v\text{-}CS \vdash e_{i\,s} = e_{i\,t}$ because $\lambda_v\text{-}CS \vdash E_1[(\lambda r.E_2[r])\ M] = ((\lambda r.E_1[E_2[r]])\ M)$ by applying (β'_Ω) to both sides. The invariants 1, 2 are also preserved.

(susp-unbox) We have ${\rho_b}_s={\rho_b}_t$, ${\rho_g}_s={\rho_g}_t$, and $\sigma_t = \sigma_s[\alpha_n \leftarrow \langle \mathsf{subx}\ \kappa_1\ \langle \mathsf{bx}\ \kappa, \alpha\rangle\rangle V]$. Let g_i be the target of the process τ, with $g_i = \langle \alpha_n, \langle(\mathbf{initp}), \eta_i\rangle\rangle$, with ${\rho_g}_s(g_i) = k_i$, and with $\eta_i = (\langle\gamma'',\eta'\rangle\ \mathbf{right}(\alpha_m,\alpha_n))$. Let b_j be the box $\langle \mathsf{bx}\ \kappa, \alpha\rangle$, with $\rho_b(b_j) = x_{b_j}$, and $\mathcal{G}_{\sigma_s}[b_j] = g_{b_j}$, $g_{b_j} \not\equiv g_i$.

There exist two evaluation contexts E_s^1 and E_s^2 such that:

$$e_{is} \equiv (\mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ E_s^1[((\lambda r.E_s^2[r])\ (\mathsf{deref}\ x_{b_j}))]))$$

with $E_s^1[\,] = \mathcal{I}_\sigma^{\rho_{gs},\rho_{bs}}[\![\gamma', \eta_i, [\,]]\!]$ and $E_s^2[\,] \equiv \mathcal{C}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![\{\langle\gamma,\eta\rangle \setminus \langle\gamma',\eta_i\rangle\}, [\,]]\!]$. On the other hand, we have

$$e_{it} \equiv (\mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta\ \mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\langle \mathsf{subx}\ \langle\gamma,\eta\rangle\ \langle \mathsf{bx}\ \kappa, \alpha\rangle\rangle]\!]))$$

with $\mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\langle \mathsf{subx}\ \langle\gamma,\eta\rangle\ b\rangle]\!]$
$\equiv ((\lambda r.\mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\mathit{suffix}_{\sigma_t}(\langle\gamma,\eta\rangle), r]\!])\ \ (\mathsf{deref}\ x_b))$
$\equiv ((\lambda r.\mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\{\langle\gamma,\eta\rangle \setminus \langle(\mathbf{initp}), \eta_i\rangle\}, r]\!])\ \ (\mathsf{deref}\ x_b))$
because $\langle\gamma,\eta\rangle \sqsupseteq_{\sigma_t}^s \langle\gamma',\eta_i\rangle \sqsupseteq_{\sigma_t}^s \langle(\mathbf{initp}),\eta_i\rangle$ using the invariants.
$\equiv ((\lambda v.E_s^1[E_s^2[v]])\ \ (\mathsf{deref}\ x_b))$

Therefore $\lambda_v\text{-}CS \vdash e_{is} = e_{it}$ by (β_Ω).

(stop_l) Let $g_i \equiv \langle\alpha_i,\kappa_i\rangle$ be the target of process τ with $\rho_{g_s}(g_i) = k_i$. Let g_j be the target with continuation $\kappa_j \equiv \langle(\mathbf{initp}), (\langle\gamma,\eta\rangle\ \mathbf{right}(\alpha_m,\alpha_n))\rangle$ in configuration $\mathcal{M}_s$. After transition, target g_j disappears in $\mathcal{M}_t$. So we have, $\forall g \neq g_j, \rho_{g_t}(g) = \rho_{g_s}(g)$, $\rho_{g_s}(g_j) = k_j$, and $\rho_{g_t}(g_j)$ is not defined; $\sigma_t = \sigma_s[\alpha_m \leftarrow V]$.
In $\mathcal{M}_s$ there is a process $\langle M',\kappa'\rangle$ with $\kappa' \sqsupseteq_{\sigma_s}^s \kappa_j$. There exist two evaluation contexts $E_s^1[\,]$ and $E_s^2[\,]$, such that:

$$e_{is} \equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{is}\ E_s^1[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![V]\!]\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![N]\!]])$$
$$e_{js} \equiv \mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \theta_{js}\ E_s^2[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![M']\!]])$$

with $E_s^1[\,] \equiv \mathcal{N}_\sigma^{\rho_{gs},\rho_{bs}}[\![\eta, \mathcal{I}_\sigma^{\rho_{gs},\rho_{bs}}[\![\gamma,\eta,[\,]]\!]]\!]$
$E_s^2[\,] \equiv \mathcal{C}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![\kappa', [\,]]\!]$

The process $\langle M',\kappa'\rangle$ that was active in $\mathcal{M}_s$ remains active in $\mathcal{M}_t$, but now, $\kappa' \sqsupseteq_{\sigma_t}^s \kappa_i$. There exists an evaluation context $E_t^1[\,]$, such that

$$e_{it} \equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{it}\ E_t^1[\mathcal{T}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![M']\!]])$$

with $E_t^1 = \mathcal{C}_{\sigma_t}^{\rho_{gt},\rho_{bt}}[\![\kappa', [\,]]\!]$. Let s' be the transition pcall that allocated locations α_m and α_n. By inductive hypothesis, we have that

$\mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \emptyset\ \mathcal{T}_{\sigma_s}^{\rho_{gs'},\rho_{bs'}}[\![N]\!])\{c_u/k_v\}^*$
$= \mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \emptyset\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![N]\!])$
$= \mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \theta_{js} E_s^2[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![M']\!]])$

with $k_j \notin FV(\mathcal{T}_{\sigma_s}^{\rho_{gs'},\rho_{bs'}}[\![N]\!])$, with k_u, k_v such that $\exists g_u, g_v, \rho_{g_{s'}}(g_u) = \rho_{g_s}(g_u) = k_u, \rho_{g_{s'}}(g_v) = k_v, \rho_{g_s}(g_v) = \perp$, with $\kappa_v \sqsupseteq_{\sigma_s}^s \kappa_u$, and $c_u \equiv \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![\langle \mathsf{co}\ \kappa_v\rangle]\!]$.
Therefore, we can deduce that

e_{is}
$\equiv \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{is}\ E_s^1[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![V]\!]\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![N]\!]])$
$= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{is}\ E_s^1[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![V]\!]\ \mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \emptyset\ \mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![N]\!])])$
by (C_{elim}) and (letref_{elim})
$= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{is}\ E_s^1[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![V]\!]\ \mathsf{callcc}\lambda k_j.(\mathsf{letref}\ \theta_{js}\ E_s^2[\mathcal{T}_{\sigma_s}^{\rho_{gs},\rho_{bs}}[\![M']\!]])])$ by IH

$$
\begin{aligned}
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{i\,s}\ E^1_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![V]\!]\ (\mathsf{letref}\ \theta_{j\,s}\ E^2_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![M']\!]])\{c_j/k_j\})] \\
&\qquad \text{by } (C_{lift}),\ (\beta_v), \text{ and } (C_{idem}) \text{ with } c_j = (\lambda v.\mathcal{A}(k_i E^1_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![V]\!]\ v])) \\
&= \mathsf{callcc}\lambda k_i.(\mathsf{letref}\ \theta_{i\,s} \cup \theta_{j\,s}'\ E^1_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![V]\!]\ \ E^2_s[\mathcal{T}^{\rho_{gs},\rho_{bs}}_{\sigma_s}[\![M']\!]]\{c_j/k_j\}]) \\
&\qquad \text{by } (\mathsf{letref}_{lift}) \text{ with } \theta_{j\,s}' = \theta_{j\,s}\{c_j/k_j\} \\
&= e_{i\,t}
\end{aligned}
$$

References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.
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. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, second edition, 1984.
4. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, 1993.
5. Matthias Felleisen. On the Expressive Power of Programming Languages. In *Proc. European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 134–151. Springer-Verlag, 1990.
6. Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).
7. Matthias Felleisen and Daniel P. Friedman. A Calculus for Assignments In Higher-Order Languages. In *Proceedings of the Fourtheen Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 314–345, Munich, W. Germany, January 1987.
8. Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *Proc. Conf. on Parallel Architecture and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 206–223. Springer-Verlag, 1987.
9. Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992. Technical Report 100, Rice University, June 1989.
10. 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.
11. Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.
12. Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.
13. 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.
14. Takayasu Ito and Manabu Matsui. A Parallel Lisp Language Pailisp and its Kernel Specification. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 58–100. Springer-Verlag, 1990.

15. 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.
16. 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.
17. James S. Miller. *MultiScheme : A Parallel Processing System Based on MIT Scheme.* PhD thesis, MIT, 1987.
18. Luc Moreau. An Operational Semantics for a Parallel Language with Continuations. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE'92)*, number 605 in Lecture Notes in Computer Science, pages 415–430, Paris, June 1992. Springer-Verlag.
19. Luc Moreau. The PCKS-machine. An Abstract Machine for Sound Evaluation of Parallel Functional Programs with First-Class Continuations. In *European Symposium on Programming (ESOP'94)*, number 788 in Lecture Notes in Computer Science, pages 424–438, Edinburgh, Scotland, April 1994. Springer-Verlag.
20. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations.* PhD thesis, University of Liège, Service d'Informatique, Institut Montefiore B28, 4000 Liège, Belgium, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.
21. Luc Moreau. Non-speculative and Upward Invocation of Continuations in a Parallel Language. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'95)*, number 915 in Lecture Notes in Computer Science, pages 726–740, Aarhus, Denmark, May 1995.
22. Luc Moreau and Daniel Ribbens. Sound Rules for Parallel Evaluation of a Functional Language with callcc. In *ACM conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 125–135, Copenhagen, June 1993.
23. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.
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 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.
26. Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.
27. Martin C. Rinard. *The Design, Implementation and Evaluation of Jade: A Portable, Implicitly Parallel Programming Language.* PhD thesis, Stanford University, August 1994.
28. Martin C. Rinard and Monica S. Lam. Semantic Foundations of Jade. In *Proceedings of the Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 105–118, January 1992.
29. Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic and Computation, Special Issue on Continuations*, 6(3/4):289–360, November 1993.
30. Amr A. Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: a Synthesis of Two Paradigms.* PhD thesis, Rice University, Houston, Texas, August 1994.
31. 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.

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