To appear in the proceedings of ESOP '94, Edinburgh, April '94, Lecture Notes in Computer Science, Springer-Verlag.

The PCKS-Machine: an Abstract Machine for Sound Evaluation of Parallel Functional Programs with First-Class Continuations

Luc Moreau

Institut d'Electricité Montefiore, B28. Université de Liège, Sart-Tilman, 4000 Liège, Belgium. moreau@montefiore.ulg.ac.be

Abstract. The PCKS-machine is an abstract machine that evaluates parallel functional programs with first-class continuations. Parallelism is introduced by the construct pcall, which provides a fork-and-join type of parallelism. To the best of our knowledge, the PCKS-machine is the first implementation of such a language that is proved to have a transparent construct for parallelism: every program using such a construct returns the same result as in the absence of this construct. This machine is also characterised by the non-speculative invocation of continuations whose interest is illustrated in an application.

1 Introduction

The programming language Scheme [13] is often extended to parallelism by adding constructs like `future`, `fork`, or `pcall`, which explicitly indicate where evaluations can proceed in parallel [5]. These constructs are expected to be *transparent*, i.e. a program using such constructs is supposed to return the same result as in the absence of these constructs. Thus, these constructs can be regarded as annotations for parallel execution that do not change the meaning of programs. Consequently, parallel programs can be developed in two phases: first, a sequential program is written using the functional programming methodology; second, annotations for parallelism are added without changing the semantics. This approach also avoids the programmer to concentrate on parallelism-specific problems like deadlocks, race conditions, and non-determinism.

This approach has been studied at length. It began with the implementation of MultiLisp by Halstead [4] using the `future` construct; it was followed by Miller's MultiScheme [8], an extension of Scheme based on the same construct. The latter highlighted the difficulty of defining first-class continuations in a parallel setting. Katz and Weise [6] proposed a definition of the `future` construct that was suitable for first-class continuations, and they suggested a notion of *legitimacy* to guarantee the transparency of this construct; their propositions were successfully implemented by Feeley [1].

However, practice is ahead of theory in this field: two theoretical points have never been studied. First, there is no formalisation of the intuitive statement "a parallel program is expected to return the same result as in the absence of constructs for parallelism." Second, to the best of our knowledge, no implementation was proved to be correct, probably due to a lack of formal criteria with respect to which the correctness can be established.

In a previous paper [11], we answered the first question by designing a calculus that models sequential and parallel evaluations. In this framework, a parallel

and a sequential evaluation of the same expression yield *observationally equivalent* results. Intuitively, two expressions are observationally equivalent if we can replace one by the other without being able to distinguish which one is used (as far as termination is concerned).

The goal of this paper is to answer the second question about the correctness of an implementation. Therefore, we designed the PCKS-machine, a parallel abstract machine that implements this calculus; it models a MIMD machine with a shared memory similar to those used in the mentioned implementations. We proved that this machine is sound with respect to the above notion of observational equivalence.

This paper is organised as follows. In Section 2, we give an application that uses parallelism and first-class continuations. In Section 3, we present the calculus that we previously designed and the notion of observational equivalence used to prove the correctness of our implementation. In Sections 4 and 5, we describe the PCKS-machine, an implementation of this calculus. Some properties of the machine are stated in Section 6. The correctness of the implementation is a two-step proof. First, in Section 7, we prove that there is a translation of a PCSK-machine into a term of the calculus. Second, in Section 8 we prove that any transition of the PCSK-machine preserves the observational equivalence in the calculus. We compare our approach with related work in Section 9.

2 Example

Let us consider the problem of searching and displaying the leaves of a tree that satisfy a given predicate. For efficiency reasons, we expect the leaves to be searched in parallel, but we want them to be printed in the same order as in a sequential depth-first search. Leaves are searched and displayed by the functions `search` and `display-leaves` given below.

The expression `(call/cc exp)` packages up the current continuation as an "escape procedure", also called a reified continuation, and applies `exp` on it; this action is called *capturing* or *reifying* a continuation. Hence, in the function `search`, `exit` will be bound to the reification of the continuation that is current when entering `search`. Invoking a reified continuation on a value v, i.e. applying it as a regular function, resumes the computation where the continuation was captured with v the value of the `call/cc` expression. When a leaf satisfies the predicate `pred`, the continuation `exit` is invoked on a list containing this leaf and a reification of the current continuation; the immediate effect is the exit of the function `search` with this list as value. In the inductive case, the annotation `fork` initiates the searches in the left and right subtrees in parallel.

```
(define (search tree pred)
  (call/cc (lambda (exit)
    (letrec ((loop (lambda (tree)
                     (cond ((leaf? tree) (if (pred tree)
                                             (call/cc (lambda (next)
                                                (exit (list tree next))))
                                             '()))
                           (else (begin (fork (loop (tree->left tree)))
                                        (loop (tree->right tree))))))))
      (loop tree)))))
```

The function `display-leaves` begins the search with a call to the function `search`. After receiving and displaying a leaf, the function `display-leaves` invokes the received continuation to obtain the following leaf.

```
(define (display-leaves tree pred)
  (let ((a-leaf (search tree pred)))       ;  search for the first leaf
    (if (null? a-leaf)
        'end
        (begin (display (car a-leaf))      ;  display the leaf
               ((cadr a-leaf) '()))))))    ;  search for the next one
```

The semantics we propose guarantees that the function `search` displays the same results in the same order as the sequential version of `search` (obtained by removing the `fork` annotation). The search of leaves would be interleaved with their displaying in the sequential version, while it is performed *speculatively* with respect to their displaying in the parallel version. The search is said to be *speculative* because the function `search` recursively traverses the tree in parallel without knowing whether the results to be found are needed.

Results are displayed as in a sequential implementation because, in the PCSK-machine, a continuation is invoked only if its invocation preserves the sequential semantics. This mechanism of invocation is said to be *non-speculative*; it is explained in Section 3 and discussed in Section 9.

3 The CPP-Calculus

The CPP-calculus [11], [3] is an extension of Plotkin's call-by-value λ-calculus with the control operator callcc. The set of *terms* M of the CPP-calculus, called Λ_{cpp}, is defined by the following grammar.

$$M ::= V \mid (M\ M) \mid (\mathsf{callcc}\ M) \mid \#_\varphi(M) \qquad V ::= c \mid x \mid (\lambda x.M) \mid \langle \varphi, K[\,]_\varphi \rangle$$

An *application* is a juxtaposition of terms $(M\ M)$, a callcc-*application* is of the form $(\mathsf{callcc}\ M)$ (the term M is called a *receiver*), and a *prompt* construct is $\#_\varphi(M)$ (with φ a *name*). A *value* V can be a constant c, a *variable* x, an *abstraction* $(\lambda x.M)$, or a *continuation point* $\langle \varphi, K[\,]_\varphi \rangle$, representing a reified continuation. A captured context $K[\,]$ and a context $C[\,]$ are defined by:

$$K[\,] ::= [\,] \mid K[[\,]M] \mid K[V[\,]] \mid K[\mathsf{callcc}\lambda k.[\,]] \mid K[\#_\varphi([\,])] \mid K[(\lambda v.[\,])V]$$
$$C[\,] ::= [\,] \mid C[[\,]M] \mid C[M[\,]] \mid C[\mathsf{callcc}[\,]] \mid C[\#_\varphi([\,])] \mid C[\lambda v.[\,]] \mid C[\langle \varphi, [\,] \rangle]$$

In the CPP-calculus (standing for Continuation Point and Prompt), continuations have a semantics that makes them suitable for parallelism.

1. A continuation can be invoked only if one knows that it is invoked in a sequential implementation. Thus, the invocation of a continuation requires to wait for the values of all expressions that are evaluated before this invocation in a sequential implementation.
2. A continuation can be captured independently of the evaluation order.

We can observe that the invocation of a continuation seriously reduces parallelism (as opposed to the capture of a continuation). The first rule can be improved in the particular case of a *downward* continuation.

3 A continuation k is *downward* if k is always part of the continuation that is in effect when k is invoked. A downward continuation is always invoked in the extent of the callcc by which it was reified. In a sequential implementation, this usage of a continuation can be implemented by simply popping the stack to the desired control point. In a parallel implementation, the invocation of a downward continuation only requires to wait for the values of the expressions that are evaluated before this invocation, but are evaluated *in the extent* of the callcc by which the continuation was reified.

We can use the following strategy to detect the invocation of a downward continuation. When a continuation is reified, it is given a fresh name φ, and a mark with the same name is pushed on the stack; when a continuation with a name φ is invoked, it is said to be downward if there is a mark with the name φ in the stack. The same technique is used in the calculus; a continuation point $\langle \varphi, K[\,]_\varphi \rangle$ receives a fresh name φ at the time of its creation, and a prompt $\#_\varphi(M)$ has the role of a mark with a name φ in the stack.

$$
\begin{array}{rcll}
(\lambda x.M)V & \rightarrow & M\{V/x\} \text{ with } V \text{ a value} & (\mathsf{C1}) \\
(ab) & \rightarrow & \delta(a,b) \text{ if this is defined} & (\mathsf{C2}) \\
\multicolumn{4}{l}{\textbf{Capture of a continuation}} \\
\mathsf{callcc}\ M & \rightarrow & \mathsf{callcc}\ \lambda k.\#_\varphi(M\ \langle \varphi, k[\,]_\varphi \rangle) \text{ with a fresh } \varphi & (\mathsf{C3}) \\
M(\mathsf{callcc}\ N) & \rightarrow & \mathsf{callcc}\ \lambda k.(\lambda f.f\ (N\ \langle p, k(f\ [\,]_p)\rangle))M & (\mathsf{C4}) \\
(\mathsf{callcc}\ M)N & \rightarrow & \mathsf{callcc}\ \lambda k.(M\ \langle p, k([\,]_p\ N)\rangle)N & (\mathsf{C5}) \\
((\lambda f.f(\mathsf{callcc}\ N))M) & \rightarrow & \mathsf{callcc}\ \lambda k.((\lambda f.f(N\ \langle p, k(f[\,]_p)\rangle))M) & (\mathsf{C6}) \\
\#_\varphi(\mathsf{callcc}\ M) & \rightarrow & \mathsf{callcc}\ \lambda k.\#_\varphi(M\langle p, k(\#_\varphi([\,]_p))\rangle) & (\mathsf{C7}) \\
\langle \varphi, (\mathsf{callcc}\ M)\rangle & \rightarrow & \langle \varphi, (M\ \langle p, [\,]_p\rangle)\rangle & (\mathsf{C8}) \\
\mathsf{callcc}\ M & \rightarrow^T & M\ \langle p, [\,]_p\rangle & (\mathsf{C9}) \\
\multicolumn{4}{l}{\textbf{Invocation of a continuation}} \\
M(\langle \varphi, K[\,]_\varphi\rangle\ V) & \rightarrow & (\langle \varphi, K[\,]_\varphi\rangle\ V) \text{ with } M, V \text{ values} & (\mathsf{C10}) \\
(\langle \varphi, K[\,]_\varphi\rangle\ V)N & \rightarrow & (\langle \varphi, K[\,]_\varphi\rangle\ V) \text{ with } V \text{ a value} & (\mathsf{C11}) \\
\#_{\varphi_1}(\langle \varphi, K[\,]_\varphi\rangle\ V) & \rightarrow & (\langle \varphi, K[\,]_\varphi\rangle\ V) \text{ with } V \text{ a value} & (\mathsf{C12}) \\
\#_\varphi(\langle \varphi, K[\,]_\varphi\rangle\ V) & \rightarrow & V \text{ with } V \text{ a value} & (\mathsf{C13}) \\
\langle \varphi, \langle \varphi_1, K_1[\,]_{\varphi_1}\rangle\ (K_2[\,]_\varphi)\rangle & \rightarrow & \langle \varphi, K_1[K_2[\,]_\varphi]\rangle & (\mathsf{C14}) \\
(\langle \varphi, K[\,]_\varphi\rangle\ V) & \rightarrow^T & K[V] \text{ with } V \text{ a value} & (\mathsf{C15}) \\
\multicolumn{4}{l}{\textbf{Simplification of a prompt and of a callcc-application}} \\
\#_\varphi(V) & \rightarrow & V \text{ with } V \text{ a value} & (\mathsf{C16}) \\
\mathsf{callcc}\ \lambda k.M & \rightarrow & M \text{ with } k \notin FV(M) & (\mathsf{C17})
\end{array}
$$

Fig. 1. Reduction system with continuation points and prompts: $\rightarrow_{cpp}$

Figure 1 displays the transitions that can be performed in the CPP-calculus. Rule C1 is Plotkin's call-by-value β-reduction, and Rule C2 is the δ-reduction. The rules C3 to C9 concern the capture of continuations. Using Rule C3, a callcc-application (callcc M) can be transformed into the application of the receiver

M to the continuation point $\langle \varphi, k[\,]_\varphi \rangle$, which is the representation of a reified continuation. This continuation point is given a fresh name φ, which is also given to a prompt wrapping this application. Intuitively, the prompt represents a mark in the stack. The following rules are used to build, step-by-step, a representation of the continuation in the continuation point. The rules C4 to C7 have a left-hand side in which a callcc-application appears, and they have a right-hand side that *is* a callcc-application. Such rules are said to *bubble-up* a callcc-application from the inside of an expression towards its top level. When a callcc-application reaches the top level of an expression, Rule C9 applies the receiver on the initial continuation $\langle p, [\,]_p \rangle$. Since this rule can only be applied at the top level, we mark it by a superscript T, and we call it a *top level rule*. According to this set of rules, a continuation can be captured when it appears in operator or in operand position of an application, or inside a prompt. Hence, the capture of a continuation is not dependent of an evaluation order.

Rules C10 to C15 describe the invocation of a continuation $\langle \varphi, K[\,]_\varphi \rangle$ on a value V. The left-hand sides of these rules have an invocation of a continuation as a subexpression, and the right-hand sides of the first three equations are an invocation of the same continuation: the invocation of a continuation prunes its surrounding context. In Rule C10, the operator is required to be a value in order to preserve the left-to-right evaluation order. When a continuation with a name φ is invoked inside a prompt with the same name, this continuation is a downward continuation; by Rule C13, its invocation can be reduced to the value on which the continuation was invoked. When the invocation of the continuation reaches the top level, the top level rule C15 installs the captured context $K[\,]$ and fills it with the value on which the continuation was invoked.

A *reduction* $\rightarrow_{cpp}$ is defined as the compatible closure of the rules of Fig. 1: $M \equiv C[P] \rightarrow_{cpp} N \equiv C[Q]$, with $P \rightarrow Q$ (except C9, C15). A *computation*, $\rightarrow^{\triangleright}_{cpp}$, is either a reduction or an application of a top level rule:

$$M \rightarrow^{\triangleright}_{cpp} N = M \rightarrow_{cpp} N \;\cup\; M \overset{\mathsf{C9}}{\rightarrow} N \;\cup M \overset{\mathsf{C15}}{\rightarrow} N$$

and we note $\rightarrow^{\triangleright *}_{cpp}$ its reflexive, transitive closure. The evaluation process is abstracted by a relation $\mathsf{eval}_{\mathsf{cpp}}$: $\mathsf{eval}_{\mathsf{cpp}}(M) = V$ if $M \rightarrow^{\triangleright *}_{cpp} V$ with V a value.

From a programmer's point of view, two behaviours can be observed as far as termination is concerned: either a program terminates or it does not terminate. Consequently, we can say that two expressions M and N have indistinguishable behaviours, if for all contexts $C[\,]$, either $C[M]$ and $C[N]$ both terminate or both do not terminate. This leads to the formal definition of observational equivalence.

Definition 1 (Observational Equivalence). $M \cong_{cpp} N$ iff forall context $C[\,]$, such that $C[M]$ and $C[N]$ are programs, either both $\mathsf{eval}_{\mathsf{cpp}}(C[M])$ and $\mathsf{eval}_{\mathsf{cpp}}(C[N])$ are defined or both are undefined.

Parallel evaluation can be performed in the calculus using the following rule, which evaluates the subexpressions of an application in parallel.

$$M \rightarrow_{cpp} M', N \rightarrow_{cpp} N' \Rightarrow (M\ N) \rightarrow_{cpp} (M'\ N')$$

4 The PCKS-Machine: a Parallel Machine

Felleisen and Friedman [2] proposed the CEK-machine to evaluate functional programs with a control operator like callcc. We generalise the CEK-machine to

multiple processes and parallel evaluation: the PCKS-machine models a MIMD (Multiple Instruction Multiple Data) machine with a shared memory. The letters PCKS stand for Parallel machine with each process composed of a Control string and a continuation K (representing a program counter and a stack, respectively) and sharing a common Store.

A *configuration* of the PCKS-machine describes the complete state of a machine; by convention, a configuration is represented by curly letters $\mathcal{M}, \mathcal{M}_i, \ldots$. A configuration $\mathcal{M}$ is a pair $\langle P, \sigma \rangle$, composed of a set of processes P and a store σ. Each *process* is composed of a *control string* and a *continuation code*.

A *control string* is either an expression of Λ_{pcks} or the distinguished symbol ‡. The language accepted by the machine is noted Λ_{pcks} and is defined by the following grammar.

$$M ::= V \mid (M\ M) \mid (\mathsf{callcc}\ M) \mid (\mathsf{pcall}\ M\ M) \qquad V ::= c \mid x \mid (\lambda x.M) \mid \langle \varphi, \kappa \rangle$$

The term $(M\ M)$ is called a *sequential-application* as opposed to the term $(\mathsf{pcall}\ M\ M)$, called a *parallel-application*. For the former, the operator is evaluated before the operand, then the application is performed. For the latter, both the operator and the operand are evaluated in parallel, then the application is performed[1]. Continuation points $\langle \varphi, \kappa \rangle$ are pairs composed of a name φ and a *continuation code* κ that is a **p**-continuation to be described below.

A *continuation code* represents the rest of the computation to be performed by a process. We distinguish between two kinds of continuation codes: **p**-continuation and **d**-continuation.

A **p**-continuation κ is of the form $(\mathbf{init}) \mid (\kappa'\ \mathbf{fun}\ V) \mid (\kappa'\ \mathbf{arg}\ N) \mid (\kappa'\ \mathbf{cont}) \mid (\kappa'\ \mathbf{name}\ \varphi) \mid (\kappa'\ \mathbf{left}\ (\alpha_m, \alpha_n, N)) \mid (\kappa'\ \mathbf{right}\ (\alpha_m, \alpha_n))$, where κ' is also a **p**-continuation. The continuation code $(\mathbf{init})$ represents the initial continuation. The code $(\kappa'\ \mathbf{fun}\ V)$ means that the expression being evaluated is the operand of an application whose operator has the value V. The code $(\kappa'\ \mathbf{arg}\ N)$ means that the expression being evaluated is the operator of a sequential application whose operand N is still to be evaluated. The code $(\kappa'\ \mathbf{cont})$ means that the expression being evaluated is the receiver of a callcc-application. There is also a mark that delimitates the extent of a callcc-application, but this mark is represented by the code $(\kappa'\ \mathbf{name}\ \varphi)$ instead of a prompt as in the CPP-calculus. The codes $(\kappa'\ \mathbf{left}\ (\alpha_m, \alpha_n, N))$ and $(\kappa'\ \mathbf{right}\ (\alpha_m, \alpha_n))$ are used when evaluating the operator and the operand of a parallel application, respectively.

A **d**-continuation κ is of the form $(\kappa'\ \mathbf{forked}\ (\alpha_i, \alpha_j)) \mid (\kappa'\ \mathbf{stop}_l\ \alpha) \mid (\kappa'\ \mathbf{stop}_r\ \alpha) \mid (\kappa'\ \mathbf{stop})$, where κ' is also a **p**-continuation. Such continuation codes are used in *dead* processes. The first code appears after forking processes. The second and third code appear when a process is stopped because it requires the content of an empty location α. The last code appears when a process is stopped after returning the final value.

In the above definitions, we say that κ is *a one-step extension* of κ'; we write it $\kappa \sqsupset \kappa'$. The reflexive, transitive closure of $\sqsupset$, called *extension*, is written $\sqsupseteq$.

[1] The annotation `fork` is derived from the annotation pcall.

$$(\mathtt{begin}\ (\mathtt{fork}\ M)\ N) \equiv (\mathsf{callcc}\ \ (\lambda k.\ (\mathsf{pcall}(\mathtt{let}\ \ ((x\ M))\ (\lambda u.u))\quad (k\ N)\ \ )))$$

By convention, lower-case letters $p, p_i, \ldots$ designate processes, and capital letters $P, P_i, \ldots$ represent sets of processes. A process p is either *active* or *dead*.

- An *active* process, $\langle M, \kappa \rangle_p$, with M an expression of Λ_{pcks} and κ a **p**-continuation, is a process that evaluates M with the continuation κ.
- A *dead* process, $\langle \ddagger, \kappa \rangle_p$, with κ a **d**-continuation, is a process that has terminated its evaluation.

The second component of a configuration is a *store* σ that binds locations to their contents. *Locations*, usually represented by $\alpha_i, \alpha_j, \ldots$ letters, belong to a set of locations *Loc*; they model addresses in a real computer. The content of a location can be a value of Λ_{pcks} or a data structure $[c; v]$ that we present in the following section. As usual, $\sigma(\alpha_m)$ denotes the content of the store σ at location α_m; $\sigma(\alpha_m) \leftarrow V$ denotes the store σ after updating the location α_m with the value V. We write $\perp$ to designate the content of an empty location.

5 Evaluation with the PCKS-Machine

Transitions of the PCKS machine are described by a relation on the set of machine configurations. We write $\mathcal{M}_i \overset{pcks}{\mapsto} \mathcal{M}_j$ when the PCKS-configuration $\mathcal{M}_i$ reduces to the PCKS-configuration $\mathcal{M}_j$. Such a *global* transition relation can be expressed in term of a *local* relation $\rightarrow_p$ that associates one process p_k and a store σ_i to a set of processes P_{p_k} and a store σ_j: $\langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle$. The relation $\rightarrow_p$ is called the *process transition* relation. Note that the relation $\rightarrow_p$ associates a process p_k (and a store) to a set of processes P_{p_k} (and a store). There is a transition from the configuration $\mathcal{M}_i$ to the configuration $\mathcal{M}_j$ if there is a transition $\rightarrow_p$ of *one* process of $\mathcal{M}_i$

$$\mathcal{M}_i \equiv \langle P_i, \sigma_i \rangle \overset{pcks}{\mapsto} \mathcal{M}_j \equiv \langle P_j, \sigma_j \rangle \Leftrightarrow \exists p_k \in P_i, \langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle, P_j \equiv P_i \backslash \{p_k\} \cup P_{p_k}$$

We write $\overset{pcks}{\mapsto}{}^*$ for the reflexive, transitive closure of $\overset{pcks}{\mapsto}$.

The process transition relation $\rightarrow_p$ for the evaluation of sequential expressions is displayed in Fig. 2. In order to lighten the notation, a transition $\langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle$ is written $p_k \rightarrow_p p'_k, \sigma_i(\alpha) \leftarrow V$, when the resulting set of processes P_{p_k} contains a single process p'_k, and when the store σ_j results from an update of the store σ_i at location α.

Rules M1 to M5 concern the evaluation of sequential, purely functional expressions as in the CEK-machine (except for the substitution instead of an environment). A sequential application (MN) forces the evaluation of M before the evaluation of N by Rule M1. By Rule M5, when a value is returned to the initial continuation, the current process is stopped, and the returned value is stored in location 0, which is, by convention, the location aimed at receiving the final result of a computation.

The rules M6 to M8 concern the reification of continuations. According to Rule M6, a callcc-application begins the evaluation of its receiver with a continuation code (κ **cont**). When a value is returned to such a continuation code, this value is applied to the reification of the current continuation by Rule M7. As in the CPP-calculus (Rule C3), the continuation point is given a fresh name φ, and a continuation code (κ **name** φ) with the same name φ is left as a prompt in the CPP-calculus. Rule M8 corresponds to Rule C16.

$$\langle (MN), \kappa \rangle_{p_k} \rightarrow_p \langle M, (\kappa \ \mathbf{arg}\ N) \rangle_{p_k} \quad (\mathsf{M1})$$

$$\langle V, (\kappa \ \mathbf{arg}\ N) \rangle_{p_k} \rightarrow_p \langle N, (\kappa \ \mathbf{fun}\ V) \rangle_{p_k} \quad (\mathsf{M2})$$

$$\langle V, (\kappa \ \mathbf{fun}\ (\lambda x.M)) \rangle_{p_k} \rightarrow_p \langle M\{V/x\}, \kappa \rangle_{p_k} \quad (\mathsf{M3})$$

$$\langle b, (\kappa \ \mathbf{fun}\ a) \rangle_{p_k} \rightarrow_p \langle \delta(a,b), \kappa \rangle_{p_k} \ \text{ if } \delta(a,b) \text{ is defined} \quad (\mathsf{M4})$$

$$\langle V, (\mathbf{init}) \rangle_{p_k} \rightarrow_p \langle \ddagger, ((\mathbf{init})\ \mathbf{stop}) \rangle_{p_k}, \sigma(0) \leftarrow V \quad (\mathsf{M5})$$

$$\langle \mathsf{callcc}\ M, \kappa \rangle_{p_k} \rightarrow_p \langle M, (\kappa \ \mathbf{cont}) \rangle_{p_k} \quad (\mathsf{M6})$$

$$\langle V, (\kappa \ \mathbf{cont}) \rangle_{p_k} \rightarrow_p \langle \langle \varphi, \kappa \rangle, ((\kappa \ \mathbf{name}\ \varphi)\ \mathbf{fun}\ V) \rangle_{p_k} \text{ new } \varphi \quad (\mathsf{M7})$$

$$\langle V, (\kappa \ \mathbf{name}\ \varphi) \rangle_{p_k} \rightarrow_p \langle V, \kappa \rangle_{p_k} \quad (\mathsf{M8})$$

$$\langle V, ((\kappa \ \mathbf{fun}\ V')\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \quad (\mathsf{M9})$$

$$\langle V, ((\kappa \ \mathbf{cont})\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \quad (\mathsf{M10})$$

$$\langle V, ((\kappa \ \mathbf{arg}\ N)\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \quad (\mathsf{M11})$$

$$\langle V, ((\kappa \ \mathbf{name}\ \varphi_1)\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \quad (\mathsf{M12})$$

$$\langle V, ((\mathbf{init})\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, \kappa_0 \rangle_{p_k} \quad (\mathsf{M13})$$

$$\langle V, ((\kappa \ \mathbf{name}\ \varphi_0)\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, \kappa \rangle_{p_k} \quad (\mathsf{M14})$$

Fig. 2. Evaluation of sequential expressions in the PCKS-machine

Rules M9 to M14 concern the invocation of a continuation point $\langle \varphi_0, \kappa_0 \rangle$ on a value V. By rules M9 to M12 the current continuation code is unconditionally pruned until either an **init** or **name** continuation code is reached. On the one hand, if an **init** code is reached, Rule M13 invokes the continuation as Rule C15. On the other hand, if a **name** code with the name φ_0 is reached, we are invoking a downward continuation, and Rule M14 behaves as Rule C13.

Rule M15 introduces parallelism; the evaluation of a parallel application $(\mathsf{pcall}\ MN)$ creates two new processes p_i, p_j to evaluate M and N in parallel. The operator and the operand are given the continuations $(\kappa \ \mathbf{left}(\alpha_m, \alpha_n, N))$ and $(\kappa \ \mathbf{right}(\alpha_m, \alpha_n))$ respectively. A **left** continuation code indicates that the term being evaluated is an operator while a **right** continuation code indicates that the term is an operand. Both codes refer to two new empty locations α_m and α_n, which are, by construction, supposed to receive the values of M and N respectively. If α_m is empty (resp. α_n), it means that the value of the operator (resp. the operand) is not yet computed.

$$\langle \mathsf{pcall}\ MN, \kappa \rangle_{p_k} \rightarrow_p \{ \langle \ddagger, (\kappa \ \mathbf{forked}(\alpha_m, \alpha_n)) \rangle_{p_k}, \langle M, (\kappa \ \mathbf{left}\ (\alpha_m, \alpha_n, N)) \rangle_{p_i}, \quad (\mathsf{M15})$$
$$\langle N, (\kappa \ \mathbf{right}\ (\alpha_m, \alpha_n)) \rangle_{p_j} \} \ \text{ with fresh locations } \alpha_m, \alpha_n$$

Now, let us suppose that V is the value obtained by the process evaluating the operand N. In Rule M16, we consider two cases according to the content of location α_m.

- The location α_m is empty, i.e. the operator has not yet returned a value. After storing the value V in location α_n, the process evaluating the operand is stopped.
- The location α_m is not empty, i.e. both the operand and the operator have returned a value, the application can be performed with the content of α_m.

$$\langle V, (\kappa\ \mathbf{right}\ (\alpha_m, \alpha_n))\rangle_{p_k} \rightarrow_p \langle \ddagger, (\kappa\ \mathbf{stop}_r\ \alpha_n)\rangle_{p_k} \text{ if } \sigma(\alpha_m) = \bot, \sigma(\alpha_n) \leftarrow V \quad \text{(M16)}$$
$$\langle V, (\kappa\ \mathbf{right}\ (\alpha_m, \alpha_n))\rangle_{p_k} \rightarrow_p \langle V, (\kappa\ \mathbf{fun}\ V')\rangle_{p_k} \quad \text{if } \sigma(\alpha_m) = V', \sigma(\alpha_n) \leftarrow V$$

The symmetric case concerns the evaluation of the operator M yielding a value V. Let us suppose that the location α_m is empty. Rule M17 also distinguishes between two cases according to the content of α_n. Either α_n is empty and the application cannot be performed, or α_n contains the value of the operand on which V can be applied.

$$\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle_{p_k} \rightarrow_p \langle \ddagger, (\kappa\ \mathbf{stop}_l\ \alpha_m)\rangle_{p_k} \text{ if } \sigma(\alpha_n) = \bot, \sigma(\alpha_m) \leftarrow V \quad \text{(M17)}$$
$$\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle_{p_k} \rightarrow_p \langle V', (\kappa\ \mathbf{fun}\ V)\rangle_{p_k} \quad \text{if } \sigma(\alpha_n) = V', \sigma(\alpha_m) \leftarrow V$$

Now, let us consider the invocation of a continuation $\langle \varphi_0, \kappa_0 \rangle$ on a value V with the continuation codes **left** and **right**. According to Rule M18, a continuation code **left** can *always* be pruned regardless of the location α_n. After application of Rule M18, if the process evaluating the operand N has not obtained a value, it is said to become *speculative*, because its result is not known to be needed later. By Rule M19, the code **right** cannot be pruned if the location α_m is empty, i.e. the operator is not yet evaluated (as in Rule C10). Thus, we stop the process and store in α_n a data structure $[\langle \varphi_0, \kappa_0 \rangle; V]$, which represents the suspension of the invocation of a continuation $\langle \varphi_0, \kappa_0 \rangle$ on a value V.

$$\langle V, ((\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \rightarrow_p \langle V, (\kappa\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \quad \text{(M18)}$$
$$\langle V, ((\kappa\ \mathbf{right}\ (\alpha_m, \alpha_n))\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \rightarrow_p \langle \ddagger, (\kappa\ \mathbf{stop}_r\ \alpha_n)\rangle_{p_k} \quad \text{(M19)}$$
$$\text{if } \sigma(\alpha_m) = \bot, \sigma(\alpha_n) \leftarrow [\langle \varphi_0, \kappa_0 \rangle; V]$$
$$\langle V, ((\kappa\ \mathbf{right}\ (\alpha_m, \alpha_n))\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \rightarrow_p \langle V, (\kappa\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \quad \text{if } \sigma(\alpha_m) \neq \bot$$

As soon as the operator yields a value, Rule M20 helps in resuming the invocation of the continuation.

$$\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle_{p_k} \rightarrow_p \langle V', (\kappa\ \mathbf{fun}\ \langle \varphi_0, \kappa_0 \rangle)\rangle_{p_k} \text{ if } \sigma(\alpha_n) = [\langle \varphi_0, \kappa_0 \rangle; V'], \sigma(\alpha_m) \leftarrow V \quad \text{(M20)}$$

In the rules M17, M20, we supposed that the location α_m was empty, i.e. it was the first time a value was passed to the continuation code $(\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))$. Otherwise, if the location α_m is not empty, the continuation is said to be *multiply invoked*. The operand N must be reevaluated to preserve the sequential semantics. Hence, in Rule M21, we evaluate again the operand N.

$$\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle_{p_k} \rightarrow_p \langle N, (\kappa\ \mathbf{fun}\ V)\rangle_{p_k} \quad \text{if } \sigma(\alpha_m) \neq \bot \quad \text{(M21)}$$

By definition of the PCKS-machine, all transitions $\rightarrow_p$ are atomic, i.e. for each rule, the operations for verifying the side-conditions, for creating processes, and for updating the store are performed in a single step.

A computation with the PCKS-machine begins with an initial configuration $\mathcal{M}_{init}$ and terminates as soon as a final configuration $\mathcal{M}_f$ is reached. An *initial configuration* $\mathcal{M}_{init} \equiv \langle \{\langle M, (\mathbf{init})\rangle_0\}, \emptyset \rangle$ is composed of a single initial process and an empty store, where M is the program to evaluate. A *final configuration* $\mathcal{M}_f \equiv \langle P_f, \sigma_f \rangle$ is such that, the set of processes P_f contains the process $\langle \ddagger, ((\mathbf{init})\mathbf{stop})\rangle_{p_k}$, and the store σ_f contains a value in the location 0.

In the following sections, we show that the machine and the calculus compute the same results. We proceed in two steps to prove such a property. First, we define a translation of a machine configuration into a term of Λ_{cpp}. Second, we prove that, for any transition of the PCKS-machine from a configuration $\mathcal{M}_1$ to a configuration $\mathcal{M}_2$, the translation of $\mathcal{M}_1$ reduces to the translation of $\mathcal{M}_2$ in the CPP-calculus (up to observational equivalence). The translation is defined in Section 7, and the equivalence is proved in Section 8. But first, we state some properties of the machine.

6 Classes of Processes and Speculative Computation

Every time a pcall construct is reduced by Rule M15, the number of processes increases by 2, but the number of *active* processes only increases by 1. Since the number of processes with a continuation $(\kappa\ \mathbf{forked}(\alpha_m, \alpha_n))$ is equal to the number of applications of Rule M15, this number increased by one is an upper bound on the number of active processes in a configuration.

During evaluation of a parallel application (pcall M N), let us suppose that the evaluation of the operator M is not terminated. The process evaluating the operator performs the actions that would be performed in a sequential order, while the operand is evaluated in advance of the sequential order. A process begins a computation in advance of the sequential order if it has a continuation of the type $(\kappa\ \mathbf{right}(\alpha_m, \alpha_n))$ with an empty location α_m. It remains in advance of the sequential order until the operator gets evaluated. When the operator is evaluated, the location α_m receives a value, and the process evaluating the operand is now executing the actions that would be performed in the sequential order. (We just have to replace $(\kappa\ \mathbf{left}(\alpha_m, \alpha_n, N))$ and $(\kappa\ \mathbf{right}(\alpha_m, \alpha_n))$ by $(\kappa\ \mathbf{arg}\ N)$ and $(\kappa\ \mathbf{fun}\ V)$ respectively, where V is the content of α_m.)

Let us consider a process p_1 with a continuation κ_1 and a process $p_2 \equiv \langle M, \kappa_2 \rangle_{p_2}$ that is obtained by reducing p_1 (or its descendants). The process p_2 is not in advance of the sequential order with respect to κ_1, if p_2 performs the actions that would be performed by p_1 in a sequential evaluation. In such a case, κ_2 is said to be a *sequential extension* of κ_1, written $\kappa_2 \sqsupseteq_s \kappa_1$, satisfying

$$\kappa_2 \sqsupseteq_s \kappa_1 \Longleftrightarrow \kappa_2 \equiv \kappa_1 \vee \begin{cases} \kappa_2 \not\equiv (\kappa'\ \mathbf{right}\ (\alpha_m, \alpha_n)) \ \wedge\ \kappa_2 \sqsupset \kappa' \\ \kappa_2 \equiv (\kappa'\ \mathbf{right}\ (\alpha_m, \alpha_n)) \ \wedge\ \sigma(\alpha_m) \neq \bot \end{cases} \text{ and } \kappa' \sqsupseteq_s \kappa_1$$

Now, we introduce the concept of *class* to specify the processes that preserve the sequential order with respect to a given continuation. We represent a class $\mathcal{C}_i$ by a pair $\langle \alpha_{ui}, \kappa_{ui} \rangle$, where the location α_{ui} is expected to receive (or contains) a result, and the continuation κ_{ui} waits for the result to be stored in α_{ui}. In a configuration $\mathcal{M} \equiv \langle P, \sigma \rangle$, we define the following classes: the initial class $\mathcal{C}_1$ is $\langle 0, (\mathbf{init}) \rangle$; for each process $\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_m, \alpha_n)) \rangle_{p_k}$, such that $\sigma(\alpha_m) = \bot$, a new class $\mathcal{C}$ is defined by the location α_n and the continuation $(\kappa\ \mathbf{right}(\alpha_m, \alpha_n))$. A process $\langle M, \kappa \rangle_{p_k}$ *belongs to a class* $\mathcal{C}_i \equiv \langle \alpha_{ui}, \kappa_{ui} \rangle$ if its continuation is a sequential extension of κ_{ui}, $\kappa \sqsupseteq_s \kappa_{ui}$. This notion of class specifies the number of active processes in a configuration:

Lemma 2. *Let $\mathcal{C}_1, \ldots, \mathcal{C}_n$ be the set of classes of the configuration $\mathcal{M} \equiv \langle P, \sigma \rangle$ with each class defined by $\mathcal{C}_i \equiv \langle \alpha_{ui}, \kappa_{ui} \rangle$; let P_i be the set of processes belonging to class $\mathcal{C}_i$. The set of processes P_i form a partition of P: $P = P_1 \cup \ldots \cup P_n$, and $\forall i, j \in 1 \ldots n$, $P_i \cap P_j = \emptyset$.*

Moreover, a set of processes P_i contains a single active process if and only if the content of the store at location α_{ui} is $\bot$. A set of processes P_i does not contain any active process iff the content of the store at location α_{ui} is not $\bot$.

It is usual to distinguish between two kinds of computations. A *mandatory* computation is a computation whose result is needed to return the final result. A *speculative* computation is a computation whose result is not known to be needed for the final result (at the time this computation is initiated), but which is launched, hoping that it will be later mandatory. We also define such notions for the PCKS-machine. Intuitively, a process p_i is said to be speculative with respect to p_j if p_j has pruned (using Rule M18) the continuation waiting for the value of process p_i.

Definition 3 (Speculative Process or Result). Let $\mathcal{C}_i \equiv \langle \alpha_{ui}, \kappa_{ui} \rangle$ and $\mathcal{C}_j \equiv \langle \alpha_{uj}, \kappa_{uj} \rangle$ such that $\kappa_{ui} \equiv (\kappa\ \mathbf{right}(\alpha_{\ell i}, \alpha_{ui}))$ with $\kappa \sqsupseteq_s \kappa_{uj}$.
The active process p_i of class $\mathcal{C}_i$ (or the result $\sigma(\alpha_i) \neq \bot$) is *speculative* with respect to a class $\mathcal{C}_j$, written $p_i \$ \mathcal{C}_j$ (or $\sigma(\alpha_i) \$ \mathcal{C}_j$) if either

- the class $\mathcal{C}_j$ has no active process because location α_{uj} is not empty, or,
- the continuation of the active process of $\mathcal{C}_j$ is not an extension of κ: $\kappa_j \not\sqsupseteq \kappa$.

7 Mapping a PCKS-Machine to a Term of the Calculus

We translate a configuration of the PCKS-machine into a term of the CPP-calculus. This translation is composed of three phases: process rebuilding, process merging and continuation point simplifications.

The *process-rebuilding function* has the following signature: $[\![\ ]\!]_p : \mathit{process} \rightarrow \mathit{process}$. Intuitively, this function undoes transitions of the PCKS-machine that concerned sequential expressions:

$$[\![\langle M, \kappa\ \mathbf{cont} \rangle_{p_k}]\!]_p = \langle (\lambda x.\mathsf{ca}\|\mathsf{cc}\ x)\ M, \kappa \rangle_{p_k} \quad [\![\langle M, \kappa\ \mathbf{fun}\ F \rangle_{p_k}]\!]_p = \langle F M, \kappa \rangle_{p_k}$$
$$[\![\langle M, \kappa\ \mathbf{name}\ \varphi \rangle_{p_k}]\!]_p = \langle \#_\varphi(M), \kappa \rangle_{p_k} \quad [\![\langle M, \kappa\ \mathbf{arg}\ N \rangle_{p_k}]\!]_p = \langle M N, \kappa \rangle_{p_k}$$

The *process-merging function* makes a new process from processes that were forked while evaluating a $\mathsf{pca}\|$, and also returns a binding between a location and a value . Its signature is:

$$[\![., ., ., .]\!]_m . : \mathit{process} \times \mathit{process} \times \mathit{process} \times \mathit{store} \rightarrow (\mathit{process} \times \mathit{binding})$$

It takes three processes and a store; it returns a process and a binding. The binding associates a value to a location α_m allocated to receive the value of an operator by Rule M15. The process-merging function is defined by five equations:

$$[\![\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_i, \alpha_j)) \rangle_{p_k}, \langle M_i, (\kappa\ \mathbf{left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle M_j, (\kappa\ \mathbf{right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]\!]_m \quad (1)$$
$$= (\ \langle ((\lambda f_i.(f_i\ M_j))\ M_i), \kappa \rangle_{p_k},\ \ (\alpha_i, f_i)\ \ )$$
$$[\![\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_i, \alpha_j)) \rangle_{p_k}, \langle \ddagger, (\kappa\ \mathbf{stop}_l(\alpha_i)) \rangle_{p_i}, \langle M_j, (\kappa\ \mathbf{right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]\!]_m \quad (2)$$
$$= (\ \langle (\sigma(\alpha_i)\ M_j), \kappa \rangle_{p_k},\ \ (\alpha_i, \sigma(\alpha_i))\ \ )$$
$$[\![\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_i, \alpha_j)) \rangle_{p_k}, \langle M_i, (\kappa\ \mathbf{left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle \ddagger, (\kappa\ \mathbf{stop}_r\ \alpha_j) \rangle_{p_j}, \sigma]\!]_m \quad (3)$$
$$= \begin{cases} (\ \langle ((\lambda f_i.(f_i\ \sigma(\alpha_j)))\ M_i), \kappa \rangle_{p_k},\ \ (\alpha_i, f_i)\ \ ) & \text{if } \sigma(\alpha_i) = \bot \\ (\ \langle (M_i\ N), \kappa \rangle_{p_k},\ \ (\alpha_i, \sigma(\alpha_i))\ \ ) & \text{if } \sigma(\alpha_i) \neq \bot \end{cases}$$
$$[\![\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_i, \alpha_j)) \rangle_{p_k}, \langle \ddagger, (\kappa\ \mathbf{stop}_r(\alpha_j)) \rangle_{p_i}, \langle M_j, (\kappa\ \mathbf{right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]\!]_m \quad (4)$$
$$= (\ \langle (\sigma(\alpha_i)\ M_j), \kappa \rangle_{p_k},\ \ (\alpha_i, \sigma(\alpha_i))\ \ )$$
$$[\![\langle \ddagger, (\kappa\ \mathbf{forked}(\alpha_i, \alpha_j)) \rangle_{p_k}, \langle M_i, (\kappa\ \mathbf{left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle \ddagger, (\kappa\ \mathbf{stop}_l\ \alpha_i) \rangle_{p_j}, \sigma]\!]_m \quad (5)$$
$$= (\ \langle (M_i\ N), \kappa \rangle_{p_k},\ (\alpha_i, \sigma(\alpha_i))\ \ )$$

In (1), the processes created by Rule M15, with continuations (κ **forked** (α_i, α_j)), (κ **left**(α_i, α_j, N)), and (κ **right**(α_i, α_j)) are translated into a process with a continuation κ and a control string $((\lambda f_i.(f_i\ M_j))M_i)$, which is observationally equivalent to $(M_i M_j)$. If a continuation is captured in M_j, it references the value of M_i. Thus, we introduce a parameter f_i as in Rule C4 (where f is referenced by the operator and the continuation), and we return a binding (α_i, f_i) between the location α_i and the parameter f_i, respectively intended to receive or to be bound to the value of M_i.

In (2) and (3), we proceed similarly with the continuations **left** and **right** respectively replaced by $\mathbf{stop}_l$ and $\mathbf{stop}_r$. In (3), we take care to detect whether a value has already been passed to the **left** code in order to use the operand N in the result as in Rule M21. In (4) and (5), we consider the cases of multiple invocations.

The two phases "process rebuilding" and "process merging" are iteratively used according to the following algorithm. The translation algorithm requires a configuration of the PCKS-machine $\mathcal{M} \equiv \langle P, \sigma \rangle$, and an initially empty store σ_c. Two results are expected: *mandatory*, which is the mandatory computation, and *speculative*, which is a set of speculative computations. Initially *speculative* is an empty set and *mandatory* is undefined. Each invocation of the merging function, at step 6, extends the store σ_c with the new binding.

1. if $\langle \ddagger, ((\mathbf{init})\ \mathbf{stop}) \rangle_{p_k} \in P$ then $mandatory \leftarrow \sigma(0)$. Proceed with $P \leftarrow P \setminus \{p_k\}$.
2. if $\langle M, (\mathbf{init}) \rangle_{p_k} \in P$ then $mandatory \leftarrow M$. Proceed with $P \leftarrow P \setminus \{p_k\}$.
3. if $\exists p_i \equiv \langle M, (\kappa\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle_{p_i}$ and p_i is speculative, $speculative \leftarrow speculative \cup \{M\}$. Proceed with $P \leftarrow P \setminus \{p_i\}$. The expression M is said to be "associated" to the class defined by $\langle \alpha_n, (\kappa\ \mathbf{right}(\alpha_m, \alpha_n)) \rangle$.
4. if $\exists \alpha_{u_i}$ that is not marked and that contains a speculative result $(\sigma(\alpha_{u_i})\ \$\ \mathcal{C}_j)$, then proceed with $speculative \leftarrow speculative \cup \{\sigma(\alpha_{u_i})\}$, and mark α_{u_i} as visited. The content $\sigma(\alpha_{u_i})$ is said to be "associated" to the class defined by location α_{u_i}.
5. if $\exists p_i, [\![p_i]\!]_p = p_i'$, then proceed with $P \leftarrow P\{p_i'/p_i\}$.
6. if $\exists p_i, p_j, p_k \in P, [\![p_k,\ p_i,\ p_j,\ \sigma]\!]_m = (p_k',\ (\alpha, v))$, then proceed with $P \leftarrow P \setminus \{p_i, p_j, p_k\} \cup \{p_k'\}$ and $\sigma_c(\alpha) \leftarrow v$.
7. stop if there is no active process in P, or no unmarked location α_{u_i}.

Resulting terms may be composed of continuation points or suspensions of invocations $[\langle p, \kappa_0 \rangle; V]$ that remain to be translated into terms of Λ_{cpp}. The *translation of continuation points and suspensions of invocations* is performed by the function $\overline{\ }$. The translation is straightforward for most of the terms. Suspensions of invocations are translated into the invocation of the continuation on a value, and continuation points are translated by a specialised function $\mathcal{S}$ that maps a continuation code to a CPP context; it uses the content of the store σ_c at location α_i to translate a continuation code (κ **right** (α_i, α_j)).

$$
\begin{array}{rl}
\overline{\lambda x.M} = \lambda x.\overline{M} & \mathcal{S}((\kappa\ \mathbf{cont}), A[\ ]) = \mathcal{S}(\kappa, ((\lambda x.\mathsf{callcc}\ x)\ A[\ ])) \\
\overline{x} = x & \mathcal{S}((\kappa\ \mathbf{name}\ \varphi), A[\ ]) = \mathcal{S}(\kappa, \#_\varphi(A[\ ])) \\
\overline{M\,N} = (\overline{M}\ \overline{N}) & \mathcal{S}((\kappa\ \mathbf{fun}\ F), A[\ ]) = \mathcal{S}(\kappa, (\overline{F}\ A[\ ])) \\
\overline{\mathsf{callcc}\ M} = \mathsf{callcc}\ (\overline{M}) & \mathcal{S}((\kappa\ \mathbf{arg}\ N), A[\ ]) = \mathcal{S}(\kappa, (A[\ ]\ \overline{N})) \\
\overline{\#_\varphi(M)} = \#_\varphi(\overline{M}) & \mathcal{S}((\kappa\ \mathbf{left}\ (\alpha_i, \alpha_j, N)), A[\ ]) = \mathcal{S}(\kappa, (A[\ ]\ \overline{N})) \\
\overline{[\langle \varphi, \kappa_0 \rangle; V]} = (\overline{\langle \varphi, \kappa_0 \rangle}\ \overline{V}) & \mathcal{S}((\kappa\ \mathbf{right}\ (\alpha_i, \alpha_j)), A[\ ]) = \mathcal{S}(\kappa, (\overline{\sigma_c(\alpha_i)}\ A[\ ])) \\
\overline{\langle \varphi, \kappa \rangle} = \langle \varphi, \mathcal{S}(\kappa, [\ ]) \rangle & \mathcal{S}((\mathbf{init}), A[\ ]) = A[\ ]
\end{array}
$$

The result of the translation of a machine configuration, written $[\![\mathcal{M}]\!]$, is a set of expressions; one of them is called mandatory, while the others are called speculative. By convention, we write $mandatory([\![\mathcal{M}]\!])$ to denote the mandatory expression of the translation, and we write $speculative([\![\mathcal{M}]\!])$ to denote the set of speculative expressions of the translation.

It can be easily proved that the translation algorithm terminates. Moreover, there is one and only one translation of a machine configuration as long as this machine configuration can be reached from an initial configuration.

Lemma 4. *Let M be a program, and let $\mathcal{M}_{init} \equiv \langle\{\langle M, \mathbf{init}\rangle_0\}, \emptyset\rangle$ be an initial configuration. Let $\mathcal{M}$ be any configuration reachable from $\mathcal{M}_{init}$: $\mathcal{M}_{init} \overset{pcks}{\mapsto}{}^{*} \mathcal{M}$. There exists only one mandatory expression and only one set of speculative expression for the translation of configuration $\mathcal{M}$, i.e. the translation is a function for configurations accessible from the initial configuration.*

If a PCKS-configuration contains a process $\langle M, \kappa\rangle_{p_k}$, the term $\overline{M}$ appears as a subexpression of a term that results from the translation of this configuration.

Lemma 5. *Let $p_k \equiv \langle M, \kappa\rangle_{p_k}$ be a process of a PCKS-configuration $\mathcal{M}$. There exists an applicative context $A[\,]$, such that*

- *if process p_k is not speculative, then $A[\overline{M}] \equiv mandatory([\![\mathcal{M}]\!])$,*
- *if process p_k is speculative, $A[\overline{M}] \in speculative([\![\mathcal{M}]\!])$.*

where an applicative context $A[\,]$ is defined by:

$$A[\,] ::= [\,] \mid A[V[\,]] \mid A[[\,]\,M] \mid A[(\lambda f.f[\,])M] \mid A[\#_\varphi([\,])]$$

8 Equivalence of the PCKS-Machine and the CPP-Calculus

Now, we can prove that the PCKS-machine preserves the CPP-calculus.

Theorem 6. *Let M be an arbitrary program of Λ_{pcks} and N be the corresponding program of Λ_{cpp} (obtained by removing the* pcall *annotations). Let $\mathcal{M}_1$ be the machine configuration reached from the initial configuration after n transitions, and let $\mathcal{M}_2$ be the machine configuration reached after $n+1$ transitions.*

$$\mathcal{M}_{init} \equiv \langle\{\langle M, (\mathbf{init})\rangle_0\}, \emptyset\rangle \overset{pcks}{\mapsto}{}^{n} \mathcal{M}_1 \overset{pcks}{\mapsto} \mathcal{M}_2 \qquad n \geq 0$$

Then, there exist two terms $N', N'' \in \Lambda_{cpp}$, such that N reduces to the mandatory term of the translation of $\mathcal{M}_1$, which reduces to the mandatory term of the translation of $\mathcal{M}_2$ (up to observational equivalence)

$$N \rightarrow_{cpp}^{\triangleright *} N' \cong_{cpp} mandatory([\![\mathcal{M}_1]\!]) \rightarrow_{cpp}^{\triangleright *} N'' \cong_{cpp} mandatory([\![\mathcal{M}_2]\!])$$

Moreover, if there exist two speculative terms e_1 and e_2 of the translations of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, "associated" to the same class, then e_1 reduces to e_2 (up to observational equivalence).

Let $e_1 \in speculative([\![\mathcal{M}_1]\!]), e_2 \in speculative([\![\mathcal{M}_2]\!])$ be two terms "associated" to the same class, then there exists e_2' such that $e_1 \rightarrow_{cpp}^{*} e_2' \cong_{cpp} e_2$

It means that, for the mandatory term, any transition in the PCKS-machine corresponds to one (or more) transitions in the CPP-calculus (up to observational equivalence). For the speculative terms, a transition of the PCKS-machine preserves the observational equivalence in the CPP-calculus.

9 Related Work and Conclusion

The CEK-machine was proposed by Felleisen and Friedman [2] as a variant of Landin's SECD-machine [7]. The CEK-machine evaluates a language that is based on the control operator $\mathcal{C}$. When $\mathcal{C}$ reifies a continuation, it replaces the current continuation by the initial one. Unlike `call/cc`, $\mathcal{C}$ aborts the current computation and requires to synchronise processes to capture a continuation. Although both `call/cc` and $\mathcal{C}$ are as expressive, $\mathcal{C}$ is less suitable for parallel evaluation because it reduces parallelism.

Halstead [5, page 19] gives three criteria for the semantics of parallel constructs and continuations in a parallel Scheme. We briefly recall them here. (1) Programs using `call/cc` without constructs for parallelism should return the same results in a parallel implementation as in a sequential one. (2) Programs that use continuations exclusively in the single-use style should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary expressions. (3) Programs should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary subexpressions, with no restrictions on how continuations are used. Our implementation satisfy these three criteria for both the `pcall` and `fork` constructs.

We have based our language on the `pcall` construct. The `future` construct is different because it introduces a call-by-name parameter-passing technique. If we wish to prove the correctness of an implementation based on the `future` construct, another calculus and another notion of observational equivalence should probably be defined.

Katz and Weise [6], Feeley [1] proposed and implemented a definition of first-class continuations in a parallel Scheme with the `future` construct. Besides the construct chosen, their proposition differs from ours by the fact that continuations are invoked speculatively, i.e. without knowing whether they preserve the sequential semantics. In addition, they introduce a notion of *legitimacy* that specifies whether a result is correct. By definition, a process is said to be *legitimate* if the code it is executing would have been executed by a sequential implementation in the absence of `future`. When the evaluation begins, the initial process is given the legitimacy property. A process with the legitimacy property preserves it as long as it does not create processes. When a legitimate process p_1 forks a process p_2 (with the `future` construct), p_2 is given the legitimacy property, and p_1 loses its legitimacy. The process p_1 recovers its legitimacy when the placeholder it receives gets determined by a legitimate process.

In an implementation where continuations are invoked speculatively, one can expect more speed up, at least theoretically, although more unnecessary computations might be performed. But the example given in Section 2 is not guaranteed to return the results in the left-to-right order if continuations are invoked speculatively; the leaves are only displayed in a left-to-right order when continuations are invoked non-speculatively. In addition, Katz and Weise propose the concept of speculation barrier, which suspends *all* non-legitimate processes at a given point. This mechanism could be used to display leaves in the left-to-right order when continuations are invoked speculatively. However, the legitimacy and the speculation barrier do not appear to be able to model our continuations. Indeed, the legitimacy can be considered a global property since it requires to find a legitimacy link between the current process and the initial one. On the contrary, the non-speculative invocation of a downward continuation that we propose re-

quires to detect the legitimacy of the process invoking the continuation with respect to the process that created this continuation without knowing whether this latter process is legitimate.

To the best of our knowledge, it is the first time that an implementation of first-class continuations is proved to be correct in a parallel setting. The PCKS-machine reflects the computations that can be performed in the CPP-calculus. Consequently, this machine has the advantages of the calculus: continuations are captured independently of the evaluation order, and downward continuations are optimally invoked. But the machine has also its defaults: the machine is too cautious when invoking an upward continuation (a continuation that is not downward). However, in [10], we observed that many continuations have a limited region of effect. (Intuitively, the region of effect of a continuation is the part of the program where this continuation is accessible.) We proved that, when invoking an upward continuation, it is sufficient to wait for the values of expressions in its region of effect. Therefore, the non-speculative approach gives continuations a new role: first-class continuations can be considered a way to sequentialise operations in a parallel program; they avoid the introduction of new constructs able to sequentialise processes in programming a language.

Acknowledgements

The anonymous referees are acknowledged for their useful comments to this work.

References

1. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.
2. 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.
3. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theor. Comp. Sci.*, 52(3):205–237, 1987.
4. Robert H. Halstead, Jr. Implementation of Multilisp : Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 9–17, Augustus 1984.
5. 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. Japan.*, LNCS 441, pages 2–57. Springer-Verlag, 1990.
6. 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.
7. P. J. Landin. The mechanical evaluation of expressions. *Comp. J.*, 6:308–320, 1964.
8. James S. Miller. *MultiScheme : A parallel processing system based on MIT Scheme*. PhD thesis, MIT, 1987.
9. 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)*, LNCS 14, pages 415–430, Paris, June 1992. Springer-Verlag.
10. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, In preparation.
11. 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*, pages 125–135, Copenhagen, June 1993. ACM.
12. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.
13. Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

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