To appear in the proceedings of TAPSOFT'95, Aarhus, May 95, Lecture Notes in Computer Science, Springer-Verlag.

Non-speculative and Upward Invocation of Continuations in a Parallel Language

Luc Moreau*

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

Abstract. A method of preserving the sequential semantics in parallel programs with first-class continuations is to invoke continuations non-speculatively. This method, which prevents a continuation from being invoked as long as its invocation can infringe the sequential semantics, reduces parallelism by the severe conditions that it imposes, especially on upward uses. In this paper, we present new conditions for invoking continuations in an upward way and both preserving the sequential semantics and providing parallelism. This new approach is formalised in the PCKS-machine, which is proved to be correct by showing that it has the same observational equivalence theory as the sequential semantics.

1 Introduction

The continuation of an expression is the computation that remains to be performed after evaluating this expression [16]. Some programming languages like Scheme [14], or SML of New Jersey [1] provide the user with two facilities to act on the interpreter continuation: the capture and the invocation. The capture of a continuation consists in packaging up the current continuation as a first-class object so that it can be passed to or returned by functions like any other object. The invocation of a continuation discards the current continuation and resumes the computation with the invoked continuation.

Parallelism can be added to a language by some annotations that specify which expressions should be evaluated in parallel [7]. These annotations are required to be *transparent*; that is, parallel programs must return the same results as in the absence of annotations.

Parallelism and first-class continuations can prevent the annotations for parallelism from being transparent. Indeed, as continuations explicitly encode the evaluation order, it is possible to write continuation-based programs that depend on this order. Since parallelism changes the evaluation order, combining both parallelism and first-class continuations can result in non-deterministic programs, which is in opposition to the definition of transparent annotations.

Previously [12, 10, 11], we proposed to invoke continuations *non-speculatively* in order to preserve the transparency property. A continuation is invoked non-speculatively if its invocation can be performed only when it is proved not to infringe the sequential semantics. The non-speculative approach essentially consists in waiting for some expressions to be evaluated before actually invoking the continuation; these expressions are the ones that are evaluated before the invocation in the

* This work was supported in part by the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Office - Science Policy Office. The scientific responsibility is assumed by its author.

sequential order. This method of invocation preserves the transparency property of annotations, but it imposes such drastic conditions on continuation invocations that it can seriously reduce parallelism in programs.

One usually distinguishes two usages of continuations [8]. If the invoked continuation is a prefix of the current continuation, the invocation is *downward*; otherwise, it is *upward*. A downward invocation simply consists in discarding a suffix of the current continuation, i.e. it corresponds to an escape. In order to provide more parallelism, we devised [12, 10, 11] a mechanism able to reduce the number of expressions for which a value had to be waited before a downward invocation of a continuation. However, this mechanism [12, 10, 11] still imposes so severe conditions on upward uses that it can reduce parallelism.

In this paper, we propose new conditions for invoking continuations in an upward way without losing parallelism, but still preserving the transparency property. The essence of our new approach relies on the observation that many uses of continuations remain local to a part of a program; for instance, when the use of a continuation (creation, invocation, storage) remains limited to a function. In such circumstances, there is no need to coordinate the invocation of the continuation with expressions that are evaluated in parallel in a part of the program that is unreachable by the continuation.

The original contributions of this paper are the following:

- We propose a new version of the PCKS-machine, an abstract machine that evaluates parallel functional programs with first-class continuations. The machine recognises upward uses of continuations and provides parallelism in such cases, while retaining the non-speculative approach for invocation. This abstract machine formalises the semantics of continuations is an annotation-based parallel language and can be regarded as a guideline for an implementation.
- We prove the correctness of the machine: the PCKS-machine returns the same result for a program as a sequential machine would do for the same program without annotations. Put differently, the semantics implemented by the PCKS-machine guarantees the transparency of annotations for parallelism.
  The proof essentially consists in proving that the observational equivalence theories of the PCKS-machine and of the sequential machine are the same. The proofs differs from the one in [10] and is much simpler.

This paper is organised as follows. We present Felleisen and Friedman's CK-machine, an abstract machine that evaluates sequential functional programs with first-class continuations. After giving the intuition of the annotations for parallelism fork and pcall, we present the PCKS-machine, and its non-speculative approach to continuations invocation. The basic approach is then modified to recognise the upward use. In Section 5, we state some properties of the machine and prove its correctness. A comparison with related work and a conclusion end this paper.

2 The CK-Machine

The set of terms accepted by the CK-machine [4, 6] is denoted by Λ_{ck} and is defined as follows, where x is taken from a set of variables $Vars$ and a from a set of constants $Csts$.

$$M ::= V \mid (M\ M) \qquad \text{(Terms)}$$
$$V ::= c \mid x \mid (\lambda x.M) \mid \langle p, \kappa \rangle \qquad \text{(Values)}$$
$$c ::= \mathsf{callcc} \mid a \qquad \text{(Constants)}$$
$$\kappa ::= (\mathbf{init}) \mid (\kappa\ \mathbf{fun}\ V) \mid (\kappa\ \mathbf{arg}\ M) \qquad \text{(Continuation Code)}$$

Terms can be *values* or *applications* $(M_1\ M_2)$ composed of an operator M_1 and an operand M_2. Values can be *constants*, *variables*, *abstractions* $(\lambda x.M)$, or *continuation points* $\langle p, \kappa\rangle$. A continuation point represents a first-class continuation; p is a tag that identifies all continuation points and κ is a *continuation code*. Constants either belong to a set of constants $Csts$ or are the distinguished constant callcc. A continuation code is an abstract data type that represents the rest of the computation in the CK-machine; its meaning will be described below. We adopt Barendregt's convention and terminology [2]. In an abstraction $(\lambda x.M)$, a variable x in M occurs *bound*; variables that are not bound by an abstraction are *free*. We define a *program* as a term without free variables.

Felleisen and Friedman [4, 6] introduced the CK-machine, an abstract machine that is characterised by two components: a control string C and a continuation K. A *configuration* of the CK-machine is a pair $\langle M, \kappa\rangle$, where the term M of Λ_{ck} is the *control string*, and the *continuation code* κ represents the rest of the computation, i.e. what remains to be performed after evaluating the control string M. An example of program annotated for parallelism can be found in [10].

In order to evaluate a term M with the CK-machine, we begin the computation with the *initial configuration* $\langle M, (\mathbf{init})\rangle$, and we end the computation when a *terminal configuration* is reached; such a terminal configuration is of the form $\langle V, (\mathbf{init})\rangle$. Transitions between configurations follow Definition 1.

Definition 1 (CK-machine)

$$\langle (M\ N), \kappa\rangle \overset{ck}{\mapsto} \langle M, (\kappa\ \mathbf{arg}\ N)\rangle \qquad \text{(operator)}$$

$$\langle V, (\kappa\ \mathbf{arg}\ N)\rangle \overset{ck}{\mapsto} \langle N, (\kappa\ \mathbf{fun}\ V)\rangle \qquad \text{(operand)}$$

$$\langle V, (\kappa\ \mathbf{fun}\ (\lambda x.M))\rangle \overset{ck}{\mapsto} \langle M\{V/x\}, \kappa\rangle \qquad (\beta_v)$$

$$\langle b, (\kappa\ \mathbf{fun}\ a)\rangle \overset{ck}{\mapsto} \langle \delta(a,b), \kappa\rangle \quad \text{if } \delta(a,b) \text{ is defined, and } a, b \in Csts. \qquad (\delta)$$

$$\langle V, (\kappa\ \mathbf{fun}\ \mathsf{callcc})\rangle \overset{ck}{\mapsto} \langle \langle p, \kappa\rangle, (\kappa\ \mathbf{fun}\ V)\rangle \qquad \text{(capture)}$$

$$\langle V, (\kappa\ \mathbf{fun}\ \langle p, \kappa_0\rangle)\rangle \overset{ck}{\mapsto} \langle V, \kappa_0\rangle \qquad \text{(invoke)}$$

□

Rules operator and operand force a left-to-right evaluation order of components of applications, using the continuation codes **fun** and **arg** which explicitly indicate the part of an application that is already evaluated or remains to be evaluated. The four last rules deal with similar configurations, whose continuation code is of the form $(\kappa\ \mathbf{fun}\ V)$, denoting that the value of the operator is V, and whose control string is a value, which is the value of the operand. We say that the value of the operator is ready to be applied on the value of the operand. Rules (β_v) and (δ) perform the β_v and δ-reductions as in Plotkin's λ_v-calculus [13]. When the value of the operator is the constant callcc, rule capture packages up the current continuation κ as a continuation point $\langle p, \kappa\rangle$ and generates a configuration where the value of the operand is ready to be applied on the continuation point. Continuation points are first-class values that can be used like regular abstractions: rule invoke describes the behaviour of the CK-machine when a continuation point is applied on a value, which is usually called *invoking a continuation*. We see that κ, the continuation of the call of $\langle p, \kappa_0\rangle$ on V, is replaced by the invoked continuation κ_0.

We can abstract the evaluation process of the CK-machine by a function.

Definition 2 ($\mathsf{eval}_{\mathsf{ck}}$) Let M and V be a term and a value of Λ_{ck}. The evaluation function $\mathsf{eval}_{\mathsf{ck}}$ is defined for M, written $\mathsf{eval}_{\mathsf{ck}}(M) = V$, if there are some transitions from the initial configuration to a final configuration of the CK-machine: $\langle M, (\mathbf{init})\rangle \overset{ck}{\mapsto}{}^{*} \langle V, (\mathbf{init})\rangle$, where $\overset{ck}{\mapsto}{}^{*}$ denotes the reflexive, transitive closure of $\overset{ck}{\mapsto}$. □

3 Parallelism: pcall and fork

We accept two annotations for parallelism: pcall and fork. An expression (pcall M N) means that the terms M and N should be evaluated in parallel; afterwards, the value of M should be applied to the value of N. Hence, the pcall annotation provides a fork-and-join type of parallelism.

The second annotation fork takes one argument and must appear in a sequence. A sequence is a construct of the form (begin M N), which is an abbreviation for $((\lambda d.N)\ M)$ with d not free in N ($d \notin FV(N)$). The expression (begin (fork M) N) means that the terms M and N should be evaluated in parallel, and the value of the sequence is the value of N.

Both pcall and fork must be transparent: the expressions (pcall M N) and (begin (fork M) N) must be indistinguishable from (M N) and (begin M N), respectively. Furthermore, we define the expression (begin (fork M) N) as (pcall (begin M $(\lambda u.u)$) N). In the sequel, we shall only consider the annotation pcall.

4 The PCKS-Machine

The PCKS-machine [10, 11] is an abstract machine that evaluates parallel functional programs with first-class continuations. This machine consists of a set of processes running in parallel (P), where each process is a represented by a CK configuration, and of a store (S) which specifies the coordination between the different processes.

The set of terms accepted by the PCKS-machine is called Λ_{pcks} and is defined by extending the grammar of Λ_{ck} as follows, with α ranging over a set of locations Loc.

$$M ::= \ldots \mid (\mathsf{pcall}\ M\ M)$$
$$\kappa ::= \ldots \mid (\kappa\ \mathbf{left}(\alpha_m, \alpha_n, M)) \mid (\kappa\ \mathbf{right}(\alpha_m, \alpha_n)) \mid (\mathbf{stop})$$

The set Λ_{pcks} extends the set Λ_{ck} with a *parallel application* (pcall M_1 M_2) composed of an operator M_1 and an operand M_2. The behaviour of the three new continuation codes is explained below in the set of transitions of the PCKS-machine.

A *configuration* of the PCKS-machine consists of a set of *processes* and a *store*. We distinguish two kinds of processes.

1. An *active process* is represented by a named CK-configuration $\langle M, \kappa\rangle_n$, where M is a control string, i.e. a term of Λ_{pcks}, κ a continuation code, and n a *process name* taken from a set of process identifiers Pid.
2. A *dead process* is represented by a special CK-configuration $\langle \ddagger, (\mathbf{stop})\rangle_n$, where the control string is the distinguished symbol $\ddagger$ and n is a process name of the set Pid.

A store binds *locations* to their *contents*. Locations model addresses in a real computer and are taken from a set Loc. Their content can be empty, can contain a value, or can contain a special data structure, whose role will be explained in the sequel. We shall use the letter p to range over processes, P over sets of processes, n over names of processes ($n \in Pid$), and α over locations ($\alpha \in Loc$).

A configuration $\mathcal{M}$ of PCKS-machine consists of a set of processes P and a store σ, and is written $\langle P, \sigma \rangle$. In order to evaluate a term M with the PCKS-machine, we begin the computation with an *initial configuration*, which is composed of a single process $\langle M, (\mathbf{init}) \rangle_{n_0}$ and an empty store. We end the computation when a *final configuration* is reached, i.e. when a process is of the form $\langle V, (\mathbf{init}) \rangle_n$. We can observe that an initial or a final configuration of the PCKS-machine contains a process that is an initial or a final configuration of the CK-machine, respectively.

In order to specify the legal transitions between configurations of the PCKS-machine, we first define a relation, called the *CKS-transition*, which can be applied to a process (represented by a CK-configuration) and a store (S).

Definition 3 (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. A CKS-transition is assumed to be performed *atomically*. □

Unlike a CK-transition, 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, instead of a single process, because new processes can be created (by the pcall-construct).

In Definition 4, parallelism in the PCKS-machine is modelled by an interleaving semantics. Then, the evaluation relation of the machine is formalised.

Definition 4 (PCKS-transition) There is a transition between a PCKS configuration $\mathcal{M}_1 \equiv \langle P_1, \sigma_1 \rangle$ and a PCKS-configuration $\mathcal{M}_2 \equiv \langle P_2, \sigma_2 \rangle$, written $\mathcal{M}_1 \overset{PCKS}{\mapsto} \mathcal{M}_2$ if there exists a process p and a set of process P such that $\langle p, \sigma_1 \rangle \overset{cks}{\mapsto} \langle P, \sigma_2 \rangle$ with $p \in P_1$ and $P_2 \equiv P_1 \setminus \{p\} \cup P$. Furthermore, transitions performed by processes are supposed to be *atomic*. □

Definition 5 ($\mathsf{eval}_{\mathsf{pcks}}$) Let M and V be a program and a value of Λ_{pcks}. The evaluation function is defined for M, written $\mathsf{eval}_{\mathsf{pcks}}(M) = V$, if there exists a final configuration $\mathcal{M}_f$ that contains a process $\langle V, (\mathbf{init}) \rangle_n$, and such that, for the initial configuration $\mathcal{M}_i \equiv \langle \{\langle M, (\mathbf{init}) \rangle_{n_0}\}, \emptyset \rangle$, we have $\mathcal{M}_i \overset{PCKS^*}{\mapsto} \mathcal{M}_f$, where $\overset{PCKS^*}{\mapsto}$ denotes the reflexive, transitive closure of $\overset{PCKS}{\mapsto}$. □

It remains to define the CKS-transitions. The first four transitions of the CK-machine (Definition 1) remains valid in the PCKS-machine. Now, let us see how a process that is evaluating a parallel application is transformed. As indicated by rule pcall in Definition 6, a new process, with a name n_i, is created to evaluate the operand N, while the process that was evaluating the parallel application has to evaluate the operator M. The continuation κ of the process evaluating the parallel application is extended with a new continuation code for each process: **left** for the process evaluating the operator and **right** for the one evaluating the operand. Furthermore, two locations α_m and α_n are allocated; these locations are intended to receive the values of the operator and of the operand, respectively. Since they explicitly appear in the continuation codes **left** and **right**, they can be accessed by the processes evaluating the operator and the operand.

Definition 6

$$\langle(\mathsf{pcall}\ M\ N),\kappa\rangle_n \overset{cks}{\mapsto} \langle M,(\kappa\ \mathbf{left}(\alpha_m,\alpha_n,N))\rangle_n,\langle N,(\kappa\ \mathbf{right}(\alpha_m,\alpha_n))\rangle_{n_i} \quad (\mathsf{pcall})$$
$$\text{with a fresh } \alpha_m \in Loc, \text{ a fresh } \alpha_n \in Loc, \text{ a new } n_i \in Pid$$

$$\langle V,(\kappa\ \mathbf{right}(\alpha_m,\alpha_n))\rangle_n \overset{cks}{\mapsto} \langle\ddagger,(\mathbf{stop})\rangle_n;\sigma(\alpha_n)\leftarrow V \quad \text{if } \sigma(\alpha_m)=\bot \quad (\mathsf{stop}_r)$$

$$\langle V,(\kappa\ \mathbf{right}(\alpha_m,\alpha_n))\rangle_n \overset{cks}{\mapsto} \langle V,(\kappa\ \mathbf{fun}\ \sigma(\alpha_m))\rangle_n;\sigma(\alpha_n)\leftarrow V \quad \text{if } \sigma(\alpha_m)\neq\bot \quad (\mathsf{ret}_r)$$

$$\langle V,(\kappa\ \mathbf{left}(\alpha_m,\alpha_n,N))\rangle_n \overset{cks}{\mapsto} \langle\ddagger,(\mathbf{stop})\rangle_n;\sigma(\alpha_m)\leftarrow V \quad \text{if } \sigma(\alpha_m)=\bot\wedge\sigma(\alpha_n)=\bot \quad (\mathsf{stop}_l)$$

$$\langle V,(\kappa\ \mathbf{left}(\alpha_m,\alpha_n,N))\rangle_n \overset{cks}{\mapsto} \langle\sigma(\alpha_n),(\kappa\ \mathbf{fun}\ V)\rangle_n;\sigma(\alpha_m)\leftarrow V \quad (\mathsf{ret}\text{-}1_l)$$
$$\text{if } \sigma(\alpha_m)=\bot\wedge value?(\sigma(\alpha_n))$$

$$\langle V,(\kappa\ \mathbf{left}(\alpha_m,\alpha_n,N))\rangle_n \overset{cks}{\mapsto} \langle N,(\kappa\ \mathbf{fun}\ V)\rangle_n \quad \text{if } \sigma(\alpha_m)\neq\bot \quad (\mathsf{ret}\text{-}\mathsf{n}_l)$$

□

A process knows that it has evaluated the operand of a parallel application, because its control string is a value and its continuation is a code **right**. It can access 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 $\langle\ddagger,(\mathbf{stop})\rangle_n$. On the contrary, if the location α_m contains a value, this value is the value of the operator, and it can be applied to 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.

Symmetrically, a process knows that it has evaluated the operator of a parallel application, because its control string is a value and its continuation is a code **left**. It must be stopped if location α_n is empty. If location α_n contains a value, two cases must be distinguished.

1. The operator is evaluated for the first time, which can be observed by the fact that location α_m is empty (cfr. rule $\mathsf{ret}\text{-}1_l$). Then, the value of the operator can be applied on the value of the operand, after updating the location α_m with the value of the operator.
2. The operator has already been evaluated, which can be observed by the fact that location α_m is not empty (cfr. rule $\mathsf{ret}\text{-}\mathsf{n}_l$). In order to preserve the sequential semantics, the operand must be re-evaluated. Such a case corresponds to rule operand which forces the operand of a sequential application to be re-evaluated. The operand N of the parallel application can be retrieved from the code **left** of the continuation where it explicitly appears.

So, locations α_m and α_n are used to coordinate the processes evaluating the operator and the operand. The location α_m is aimed at receiving the first value of the operator, while the location α_n is intended to receive the value of the operand. Definition 7 displays the CKS-transitions related to continuations.

Definition 7

$$\langle V,(\kappa\ \mathbf{fun}\ \mathsf{callcc})\rangle_n \overset{cks}{\mapsto} \langle\langle p,\kappa\rangle,(\kappa\ \mathbf{fun}\ V)\rangle_n \quad (\mathsf{capture})$$

$$\langle V,((\mathbf{init})\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V,\kappa\rangle_n \quad (\mathsf{invoke}_{init})$$

$$\langle V,((\kappa_1\ \mathbf{fun}\ V')\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V,(\kappa_1\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \quad (\mathsf{prune}_f)$$

$$\langle V,((\kappa_1\ \mathbf{arg}\ N)\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V,(\kappa_1\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \quad (\mathsf{prune}_a)$$

$$\langle V,((\kappa_1\ \mathbf{left}(\alpha_m,\alpha_n,N))\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V,(\kappa_1\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \qquad (\mathsf{prune}_l)$$

$$\langle V,((\kappa_1\ \mathbf{right}(\alpha_m,\alpha_n))\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V,(\kappa_1\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \quad \text{if } \sigma(\alpha_m)\neq\perp \qquad (\mathsf{prune}_r)$$

$$\langle V,((\kappa_1\ \mathbf{right}(\alpha_m,\alpha_n))\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle \ddagger,(\mathbf{stop})\rangle_n;\sigma(\alpha_n)\leftarrow\mathbf{suspend}(\langle p,\kappa\rangle,V)(\mathsf{suspend}_r)$$
$$\text{if } \sigma(\alpha_m)=\perp$$

$$\langle V,(\kappa\ \mathbf{left}(\alpha_m,\alpha_n,N))\rangle_n \overset{cks}{\mapsto} \langle V',(\kappa\ \mathbf{fun}\ \langle p,\kappa_0\rangle)\rangle_n;\sigma(\alpha_m)\leftarrow V \qquad (\mathsf{resume}_r)$$
$$\text{if } \sigma(\alpha_m)=\perp \wedge \text{ if } \sigma(\alpha_n)=\mathbf{suspend}(\langle p,\kappa_0\rangle,V')$$

□

Rule capture of the CK-machine is still valid in the PCKS-machine, but rule invoke becomes unsound. Indeed, rule invoke, as designed in the CK-machine, can replace the current continuation by an invoked continuation in a *single* transition; used in the PCKS-machine, invoke would be able to replace a current continuation with a code **right** by the invoked continuation even though the operator corresponding to the code **right** has not returned a value. Since the PCKS-machine must compute the same results as the CK-machine, we replace rule invoke by three rules invoke_{init}, prune_f, and prune_a. A single transition invoke of the CK-machine will be simulated by a sequence of prune_f and prune_a followed by invoke_{init} in the PCKS-machine[2]. A rule like prune_f (and similarly for prune_a) is said to prune the continuation of a process; indeed, the process continuation before transition $((\kappa_1\ \mathbf{fun}\ V')\ \mathbf{fun}\ \langle p,\kappa\rangle)$ is shortened to $(\kappa_1\ \mathbf{fun}\ \langle p,\kappa\rangle)$. A succession of prune_f and prune_a forms the *abortive phase* where the continuation code of the process is pruned until invoke_{init} can be used. We can observe that invoke_{init} is an instance of invoke with κ replaced by (**init**).

The mode of invocation of continuations in the PCKS-machine is said to be *non-speculative* because a continuation is invoked only if its invocation does not infringe the sequential semantics. Let us examine how such a mode of invocation behaves in the presence of **left** and **right**. According to rule prune_l, a continuation code **left** can always be pruned. Indeed, a process $\langle V,((\kappa_1\ \mathbf{left}(\alpha_m,\alpha_n,N))\ \mathbf{fun}\ \langle p,\kappa\rangle)\rangle_n$ evaluates the operator of a parallel application; since the operator is evaluated before the operand in a CK-machine, a continuation can always be invoked in the operator.

Symmetrically, if the operator of a parallel application is already evaluated and the operand invokes a continuation, the code **right** can be pruned as specified by rule prune_r, because the execution of the PCKS-machine follows the CK-execution. On the contrary, if the operator of a parallel application is not yet evaluated when the operand invokes a continuation, the code **right** cannot be pruned if the sequential semantics must be preserved; in such a case, rule $\mathsf{suspend}_r$ *suspends* the invocation of the continuation, by storing in α_n a data-structure containing the continuation and the value, and by stopping the process. The invocation of the continuation can be resumed as soon as the process that was evaluating the operator yields its value, as specified by rule resume_r.

The rules of Definitions 1 (four first rules), 6, 7 specify a machine that evaluates parallel programs with first-class continuations, while preserving the sequential semantics. Unfortunately, in order to preserve the sequential semantics, rule $\mathsf{suspend}_r$ imposes so drastic conditions on the invocation of continuations that it can seriously reduce parallelism in a parallel program with first-class continuations.

[2] By a simple reorganisation of the continuation of a process in two components, several continuation codes **arg** and **fun** can be pruned in a single step as in the CK-machine [11].

In the CK-machine, invoking a continuation replaces the current continuation by the invoked continuation. In many usages of continuations, the invoked continuation and the current continuation have a common prefix (in the worst case, the common prefix is simply the initial continuation **init**). Hence, invoking a continuation is equivalent to replacing a suffix of the current continuation by a suffix of the invoked continuation. We can say that the invocation of a continuation has a *local* effect on the computation because it only changes a suffix of the current continuation and leaves the prefix unchanged.

In the PCKS-machine, instead of suspending the invocation of a continuation in the operand of a parallel application when the operator has not yielded a value, we can immediately reinstate the invoked continuation if the current continuation is a prefix of it. (This corresponds to a local use of the continuation.) Let us define the relation "is prefix of" on continuations.

Definition 8 (Extension and Prefix) A continuation κ_1 is a *"one-step" extension* of a continuation κ_2, written $\kappa_1 \sqsupset \kappa_2$, if one of the following equality[3] holds.

$$\begin{array}{ll} \kappa_1 \equiv (\kappa_2\ \mathbf{fun}\ V) & \kappa_1 \equiv (\kappa_2\ \mathbf{arg}\ N) \\ \kappa_1 \equiv (\kappa_2\ \mathbf{left}\ (\alpha_m, \alpha_n, N)) & \kappa_1 \equiv (\kappa_2\ \mathbf{right}\ (\alpha_m, \alpha_n)) \end{array}$$

The relation *extension* is the reflexive, transitive closure of *"one-step" extension* and is written $\kappa_1 \sqsupseteq \kappa_2$. We also say that k_2 is a *prefix* of k_1 if $\kappa_1 \sqsupseteq \kappa_2$. □

Rule $\mathsf{suspend}_r$ of Definition 7 should be replaced by the rules of Definition 9. Now, if the invoked continuation is not an extension of the current continuation, it must be suspended by rule $\mathsf{suspend}_r$. Otherwise, rule invoke_{up} reinstates the invoked continuation, while preserving the sequential semantics.

Definition 9

$$\langle V, ((\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))\ \mathbf{fun}\ \langle p, \kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle \ddagger, (\mathbf{stop})\rangle_n; \sigma(\alpha_n) \leftarrow \mathbf{suspend}(\langle p, \kappa\rangle, V) \quad (\mathsf{suspend}_r)$$
$$\text{if } \sigma(\alpha_m) = \bot \wedge \kappa \not\sqsupseteq (\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))$$
$$\langle V, ((\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))\ \mathbf{fun}\ \langle p, \kappa\rangle)\rangle_n \overset{cks}{\mapsto} \langle V, \kappa\rangle_n \quad (\mathsf{invoke}_{up})$$
$$\text{if } \sigma(\alpha_m) = \bot \wedge \kappa \sqsupseteq (\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))$$

□

Thanks to the rules of Definition 9, the PCKS-machine not only evaluates parallel programs using first-class continuations, but also preserves parallelism in the program, by avoiding to suspend local upward invocations. Invocations can be suspended only in the presence of race conditions that might not preserve the sequential result.

5 Properties

Our goal is to prove the soundness of the PCKS-machine with respect to the sequential semantics, which is implemented by the CK-machine. First, we define a translation that removes the annotations for parallelism in a parallel program, i.e. which returns the sequential version of a program.

[3] The relation $\sqsupseteq$ uses the syntactic equality on terms and continuation codes. In practice, the reorganisation of the continuation in two components, as suggested in footnote 2, can be used for improving the efficiency of $\sqsupseteq$.

Definition 10 (Sequential Version of a Term) The sequential version of a term M of Λ_{pcks} is a term $\mathcal{S}[\![M]\!]$ of Λ_{ck}, defined as follows[4].

$$\begin{array}{rlrl}
\mathcal{S}[\![(M\ N)]\!] &= (\mathcal{S}[\![M]\!]\ \mathcal{S}[\![N]\!]) & \mathcal{S}[\![(\mathbf{init})]\!]^c &= (\mathbf{init}) \\
\mathcal{S}[\![(\mathsf{pcall}\ M\ N)]\!] &= (\mathcal{S}[\![M]\!]\ \mathcal{S}[\![N]\!]) & \mathcal{S}[\![(\kappa\ \mathbf{fun}\ V)]\!]^c &= (\mathcal{S}[\![\kappa]\!]^c\ \mathbf{fun}\ \mathcal{S}[\![V]\!]) \\
\mathcal{S}[\![\lambda x.M]\!] &= \lambda x.\mathcal{S}[\![M]\!] & \mathcal{S}[\![(\kappa\ \mathbf{arg}\ N)]\!]^c &= (\mathcal{S}[\![\kappa]\!]^c\ \mathbf{arg}\ \mathcal{S}[\![N]\!]) \\
\mathcal{S}[\![\langle p, \kappa\rangle]\!] &= \langle p, \mathcal{S}[\![\kappa]\!]^c\rangle & & \\
\mathcal{S}[\![x]\!] &= x, \text{ for } x \in Csts \text{ or } \in Vars & &
\end{array}$$

□

The major result of this section is the following theorem, which states that the observational equivalence theories of the CK-machine and PCKS-machine are the same.

Theorem 11 *Let M be a term of Λ_{pcks} and $\mathcal{S}[\![M]\!]$ its sequential version.*

$$\mathsf{eval}_{\mathsf{ck}}(\mathcal{S}[\![M]\!]) = V \quad \textit{iff} \quad \mathsf{eval}_{\mathsf{pcks}}(M) = V'.$$

□

In the sequel, we present the intuition of the proof. The transition pcall creates a new active process to evaluate the operand of a parallel application. Such an operand is evaluated *in advance of* the sequential order. The evaluation of the operand remains in advance of the sequential order as long as the operator is being evaluated. In order to identify the computations that are in advance of the sequential order, we simply have to detect all the pcall transitions that were executed for which the location α_m is empty, i.e. the operator is not evaluated yet. On the other hand, there is a single computation that is *not* in advance of the sequential order: it is the process that follows the left-to-right evaluation order. Let us call this process the *mandatory* process and all other processes *speculative*.

In order to uniformly characterise the different kinds of computations (speculative or mandatory), we introduce the concept of *target*. Each computation evaluating an expression is characterised by the continuation of this expression and the location where to store the value of this expression. Intuitively, a *target* is a pair (location, continuation) for a computation. First, let us slightly change the definition of a final configuration of the PCKS-machine. We assume that the special location 0 is allocated to receive the final value of the whole computation. We add an extra rule to the PCKS-machine, called init, which stores the final result into location 0. A PCKS-configuration will be said to be final if it contains a value in location 0.

$$\langle V, (\mathbf{init})\rangle_n \rightarrow \langle \ddagger, (\mathbf{stop})\rangle_n; \sigma(0) \leftarrow V \qquad (\mathsf{init})$$

Now, we can define the notion of target.

Definition 12 (Target) Let $\mathcal{M} \equiv \langle P, \sigma\rangle$ be a configuration of the PCKS-Machine. A target is a pair $\langle \alpha, \kappa\rangle$ containing a location α and a continuation κ, characterising a computation evaluating an expression with a continuation κ, and whose value is intended to be stored in α. The set of targets of a configuration $\mathcal{M}$ is defined as follows.

- The pair $\langle 0, (\mathbf{init})\rangle$ 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 the location α_m is empty, $\sigma(\alpha_m) = \bot$, then the pair $\langle \alpha_n, (\kappa\ \mathbf{right}(\alpha_m, \alpha_n))\rangle$ is a target of $\mathcal{M}$.

□

[4] The translation is not defined for continuation codes **left** and **right** because it is meaningless to consider such codes independently of a store of the PCKS-machine.

The target $\langle 0, (\mathbf{init})\rangle$ is said to be *mandatory*, while the others are *speculative*. A target $\langle \alpha, \kappa\rangle$ is *active* if $\sigma(\alpha) = \bot$. Each active process can be uniquely associated with an active target. For this purpose, we define a new relation, called *sequential extension*, which is a subset of the relation *extension*.

Definition 13 (Sequential Extension) Let σ be the store of a given PCKS-configuration. A continuation κ_1 is a *"one-step" sequential extension* of a continuation κ_2 (with respect to σ), written $\kappa_1 \sqsupset^s_\sigma \kappa_2$, if one of the following equation holds.

$$\begin{array}{ll} \kappa_1 \equiv (\kappa_2\ \mathbf{fun}\ V) & \kappa_1 \equiv (\kappa_2\ \mathbf{arg}\ N) \\ \kappa_1 \equiv (\kappa_2\ \mathbf{left}\ (\alpha_m, \alpha_n, N)) & \kappa_1 \equiv (\kappa_2\ \mathbf{right}\ (\alpha_m, \alpha_n)) \quad \text{if } \sigma(\alpha_m) \neq \bot \end{array}$$

The relation *sequential extension* is the reflexive, transitive closure of *"one-step" sequential extension* and is written $\kappa_1 \sqsupseteq^s_\sigma \kappa_2$. □

The active target associated with an active process can be obtained by the following Definition, and it is easy to prove that there is a unique active process that is associated with each active target.

Definition 14 Let $\mathcal{M} \equiv \langle P, \sigma\rangle$ be a configuration of the PCKS-machine. Let $g \equiv \langle \alpha, \kappa\rangle$ be a target of $\mathcal{M}$. The active process $\langle M, \kappa'\rangle_n$ of P is *associated* with target g if $\kappa' \sqsupseteq^s_\sigma \kappa$. We also say that the target of a continuation κ' is $g \equiv \langle \alpha, \kappa\rangle$ if $\kappa' \sqsupseteq^s_\sigma \kappa$. □

In order to prove the soundness of the PCKS-machine, we proceed in two steps. First, we define a translation of a PCKS-machine configuration into a term of Sabry and Felleisen's λ_v-C-calculus [15]. Second, we prove that for any transition of the PCKS-machine between two configurations $\mathcal{M}_1$ and $\mathcal{M}_2$, the translations of $\mathcal{M}_1$ and $\mathcal{M}_2$ are provably equal in the λ_v-C-calculus.

First, we define the translation of a configuration of the PCKS-machine. A configuration will be translated into a *set* of terms, one term for each target of the machine. The translation of a configuration uses a process translation function $\mathcal{P}^\rho_\sigma[\![\ ]\!]$, a term translation function $\mathcal{T}^\rho_\sigma[\![\ ]\!]$, and a continuation translation function $\mathcal{C}^\rho_\sigma[\![\ ,]\!]$, specified in Definitions 16, 17. By convention, we shall use the letter x to range over user variables, k to range over variables associated with targets, and v to range over variables for continuations. The sets of these variables are supposed to be disjoined.

Definition 15 (Machine Translation) Let $\mathcal{M} \equiv \langle P, \sigma\rangle$ be a PCKS-configuration. Let $\{g_1, \ldots, g_n\}$ be the targets of the machine, and let be $\{k_1, \ldots, k_n\}$ be fresh variables. Let ρ be an environment mapping each target g_i to variable k_i. The translation of the machine $\mathcal{M}$ is a set of terms, obtained as follows for each target g_i.

1. If target g_i is active, then let $\langle M, \kappa\rangle$ be its associated active process. The translation of $\mathcal{M}$ contains the term $\mathsf{callcc}\lambda k_i.\mathcal{P}^\rho_\sigma[\![\langle M, \kappa\rangle]\!]$.
2. If target g_i is not active, then let M be the content of its non empty location α_i. The translation of $\mathcal{M}$ contains the term $\mathsf{callcc}\lambda k_i.\mathcal{T}^\rho_\sigma[\![M]\!]$.

□

Let us define the function that translates a process. It uses the term and the continuation translation functions, which are mutually recursive.

Definition 16 (Process Translation) Let $\mathcal{M} \equiv \langle P, \sigma\rangle$ be a configuration of the PCKS machine, with ρ a function mapping each target g_i of $\mathcal{M}$ to a fresh variable k_i. The translation of a process $\langle M, \kappa\rangle_n$, written $\mathcal{P}^\rho_\sigma[\![\langle M, \kappa\rangle_n]\!]$, is defined as $\mathcal{C}^\rho_\sigma[\![\kappa, \mathcal{T}^\rho_\sigma[\![M]\!]]\!]$, where $\mathcal{T}^\rho_\sigma[\![M]\!]$ and $\mathcal{C}^\rho_\sigma[\![\kappa,\ ]\!]$ are the translations of M and κ. □

Definition 17 (Term and Continuation Translation) Let $\mathcal{M} \equiv \langle P, \sigma \rangle$ be a configuration of the PCKS machine, with ρ a function mapping each target g_i of $\mathcal{M}$ to a fresh variable k_i. The translation of a term M, written $\mathcal{T}^{\rho}_{\sigma}[\![M]\!]$, is defined as follows.

$$\begin{array}{rl}
\mathcal{T}^{\rho}_{\sigma}[\![(\mathsf{pcall}\ M\ N)]\!] = (\mathcal{T}^{\rho}_{\sigma}[\![M]\!]\ \mathcal{T}^{\rho}_{\sigma}[\![N]\!]) & \mathcal{T}^{\rho}_{\sigma}[\![\lambda x.M]\!] = \lambda x.\mathcal{T}^{\rho}_{\sigma}[\![M]\!] \\
\mathcal{T}^{\rho}_{\sigma}[\![(M\ N)]\!] = (\mathcal{T}^{\rho}_{\sigma}[\![M]\!]\ \mathcal{T}^{\rho}_{\sigma}[\![N]\!]) & \mathcal{T}^{\rho}_{\sigma}[\![x]\!] = x \ \text{ if } x \in Csts \text{ or} \in Vars \\
\mathcal{T}^{\rho}_{\sigma}[\![\mathbf{suspend}(\langle p, \kappa \rangle, V)]\!] = (\mathcal{T}^{\rho}_{\sigma}[\![\langle p, \kappa \rangle]\!]\ \mathcal{T}^{\rho}_{\sigma}[\![V]\!]) & \\
\mathcal{T}^{\rho}_{\sigma}[\![\langle p, \kappa \rangle]\!] = \lambda v.\mathcal{A}\ (k_i\ \mathcal{C}^{\rho}_{\sigma}[\![\kappa, v]\!]) \text{ with } k_i = \rho(g_i) \text{ if } g_i \equiv \langle \alpha_i, \kappa_i \rangle, \kappa \sqsupseteq^{s}_{\sigma} \kappa_i &
\end{array}$$

The translation of a continuation κ for a term M, written $\mathcal{C}^{\rho}_{\sigma}[\![\kappa, M]\!]$, is defined as:

$$\begin{array}{rl}
\mathcal{C}^{\rho}_{\sigma}[\![(\mathbf{init}), M]\!] = M & \\
\mathcal{C}^{\rho}_{\sigma}[\![(\kappa\ \mathbf{arg}\ N), M]\!] = \mathcal{C}^{\rho}_{\sigma}[\![\kappa, (M\ \mathcal{T}^{\rho}_{\sigma}[\![N]\!])]\!] & \\
\mathcal{C}^{\rho}_{\sigma}[\![(\kappa\ \mathbf{fun}\ V), M]\!] = \mathcal{C}^{\rho}_{\sigma}[\![\kappa, (\mathcal{T}^{\rho}_{\sigma}[\![V]\!]\ M)]\!] & \\
\mathcal{C}^{\rho}_{\sigma}[\![(\kappa\ \mathbf{left}(\alpha_m, \alpha_n, N)), M]\!] = \mathcal{C}^{\rho}_{\sigma}[\![\kappa, (M\ \mathcal{T}^{\rho}_{\sigma}[\![N]\!])]\!] & \\
\mathcal{C}^{\rho}_{\sigma}[\![(\kappa\ \mathbf{right}(\alpha_m, \alpha_n)), M]\!] = \mathcal{C}^{\rho}_{\sigma}[\![\kappa, (\mathcal{T}^{\rho}_{\sigma}[\![\sigma(\alpha_m)]\!]\ M)]\!] & \text{if } \sigma(\alpha_m) \neq \bot \\
\mathcal{C}^{\rho}_{\sigma}[\![(\kappa\ \mathbf{right}(\alpha_m, \alpha_n)), M]\!] = M & \text{if } \sigma(\alpha_m) = \bot
\end{array}$$

□

Lemma 18 states that for any transition between two configurations of the PCKS-machine, terms that result from the translation of the configurations and that correspond to a same target are provably equal in the λ_v-C-calculus.

Lemma 18 Let $\mathcal{M}_t \equiv \langle P_t, \sigma_t \rangle$ be a PCKS-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}. For any configuration $\mathcal{M}_s \equiv \langle P_s, \sigma_s \rangle$ 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 $g_{i_s} \equiv g_{i_t}$.

The terms e_{i_s} and e_{i_t} satisfy $\lambda_v\text{-}C \vdash e_{i_s}\{c_u/k_u\}^* = e_{i_t}$ for any k_u such that $\exists \rho_u,\ \rho_s(g_u) = k_u, \rho_t(g_u) = \bot$, and $c_u = \mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa_u \rangle]\!]$. □

Proofs of Theorem 11 and Lemma 18 can be found in Section 9.

6 Related Work

This paper extends previous results [12, 10, 11] by devising a new criterion for invoking continuations, which preserves the sequential semantics and provides parallelism for upward uses.

Another annotation for parallelism is `future` [7]; its semantics in a purely function language was stated by Flanagan and Felleisen [5]. They use this semantics to statically analyse programs in order to perform a "touch optimisation", i.e. to remove the touch operator when it can be predicted at compile-time that its argument is never a placeholder. We could apply similar optimisations to improve programs efficiency by predicting that an application site always invokes a "local" continuation, i.e. a continuation that satisfies the side-condition of invoke_{up}.

Research on the interaction of `future` and continuations were mainly concerned with the implementation. Halstead [7, page 19] gives three criteria for the semantics of parallel constructs and continuations in a parallel Scheme. We list 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 semantics satisfy these conditions. The first criterion is proved by the following proposition: let M be a program of Λ_{ck}, $\mathsf{eval}|_{\mathsf{pcks}}(M) = \mathsf{eval}|_{\mathsf{ck}}(M)$. The second and third criterion are satisfied by Theorem 11. Rule $\mathsf{ret\text{-}1}_l$ deals with the single-use style, while $\mathsf{ret\text{-}n}_l$ with the multiple-use.

Katz and Weise [9] proposed a implementation technique to provide a transparent `future` annotation for a language with first-class continuation; this technique was successfully implemented by Feeley [3]. In Katz and Weise's approach, continuations are invoked speculatively; that is, they are invoked as soon as possible, without verifying whether their invocation preserves the sequential semantics. In addition, in order to preserve the transparency of the annotation `future`, processes are threaded by a *legitimacy* link. A process is *legitimate* if the code it is executing would have been executed by a sequential implementation in the absence of parallelism. A result is legitimate if it is returned by a legitimate process.

In an implementation where continuations are invoked speculatively, one can expect more speed up, at least theoretically, but more unnecessary computations might be performed than in an implementation with non-speculative invocations. Hence, the non-speculative approach allows the user to have a better control on speculative computations. Furthermore, in the non-speculative approach, a first-class continuation encodes the partial order that must be respected to preserve the sequential semantics. So, first-class continuations can be seen as control operators for synchronising computations; a program illustrating this property can be found in [10].

7 Conclusion

In this paper, we have presented the PCKS-machine, an abstract machine that is able to evaluate parallel functional programs with first-class continuations. This machine is sound with respect to the sequential semantics. In the PCKS-machine, continuations are invoked non-speculatively, i.e. their invocations are allowed only if they do not infringe the sequential semantics. Although this mode of invocation intuitively seems to reduce parallelism by very stringent conditions, the PCKS-machine proves that parallelism can be preserved in programs with first-class continuations. The PCKS-Machine can be considered as a guideline for implementation of continuations in an annotation-based parallel language.

8 Acknowledgement

I am grateful to Daniel Ribbens and the anonymous referees for their helpful comments.

9 Appendix: Proofs

Proof Lemma 18. We proceed by induction on the number of transitions t and by case on the last transition. First, we consider the transitions that do not change the sets of targets and for which the translations of $\mathcal{M}_s$ and $\mathcal{M}_t$ are the same (init, $\mathsf{ret\text{-}n}_l$, stop_r, $\mathsf{ret\text{-}n}_l$, ret_r, $\mathsf{operator}$, $\mathsf{operand}$, $\mathsf{suspend}_r$).

(ret-n$_l$) $\rho_s = \rho_t$ and $\sigma_s = \sigma_t$. Let g_i be the target of $\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle$ with $\rho_s(g_i) = \rho_t(g_i) = k_i$. There exists an evaluation context $E[\ ]$, such that $e_{is} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]] \equiv e_{it} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_t}_{\sigma_t}[\![V]\!]\ \mathcal{T}^{\rho_t}_{\sigma_t}[\![N]\!]]$.

Some rules do not change the set of targets and correspond to a reduction in the λ_v-C-calculus (β_v, capture, δ, prune$_f$, prune$_a$, prune$_l$, prune$_r$). For instance,

(capture) $\rho_s = \rho_t$ and $\sigma_s = \sigma_t$. Let g_i be the target of $\langle\langle p, \kappa\rangle, (\kappa\ \mathbf{fun}\ V)\rangle$, with $\rho_s(g_i) = \rho_t(g_i) = k_i$. There exists an evaluation context $E[\ ]$, such that $e_{is} \equiv \mathsf{callcc}\lambda k_i.E[\mathsf{callcc}\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]]$ and $e_{it} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_t}_{\sigma_t}[\![V]\!]\ \mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa\rangle]\!]]$. Furthermore, $\mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa\rangle]\!] \equiv \lambda v.\mathcal{A}(k_i\ \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa, v]\!]) \equiv \lambda v.\mathcal{A}(k_i\ E[v])$ because the evaluation context $E[\ ]$ was defined as $E[\ ] \equiv \mathcal{C}^{\rho_s}_{\sigma_s}[\![\kappa, [\ ]]\!] \equiv \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa, [\ ]]\!]$.

$$e_{is} = e_{it} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \lambda v.\mathcal{A}(k_i\ E[v])] \text{ by } (C_{lift}) \text{ and } (C_{idem})$$

Rule pcall creates a new target.

(pcall) This rule creates a new target $g_x \equiv \langle \alpha_n, \kappa_x\rangle$, with $\kappa_x \equiv (\kappa\ \mathbf{right}\ (\alpha_m, \alpha_n))$. So, we have $\rho_t = \rho_s[g_x \leftarrow k_x]$ and $\sigma_t = \sigma_s[\alpha_m \leftarrow \bot][\alpha_n \leftarrow \bot]$.
In the configuration $\mathcal{M}_t$, there is no other target with a continuation κ_x because α_m and α_n were freshly allocated. We have $e_{xt} \equiv \mathsf{callcc}\lambda k_x.\mathcal{T}^{\rho_t}_{\sigma_t}[\![N]\!]$, with k_x the fresh variable associated with target g_x. So, $k_x \notin FV(\mathcal{T}^{\rho_t}_{\sigma_t}[\![N]\!])$.
Let g_i be the target of $\langle(\mathsf{pcall}\ M\ N), \kappa\rangle$, with $\rho_s(g_i) = \rho_t(g_i) = k_i$. There exists an evaluation context $E[\ ]$, such that $e_{is} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![M]\!]\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]] \equiv e_{it} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_t}_{\sigma_t}[\![M]\!]\ \mathcal{T}^{\rho_t}_{\sigma_t}[\![N]\!]]$

A target g_j in $\mathcal{M}_s$ disappears after application of rules ret-1$_l$, stop$_l$, ret-1$_l$, resume$_r$. Here, we use the inductive hypothesis. For instance,

(ret-1$_l$) Let g_i be the target of $\langle V, (\kappa\ \mathbf{left}\ (\alpha_m, \alpha_n, N))\rangle$ with $\rho_s(g_i) = k_i$. Let g_j be the target with continuation $(\kappa\ \mathbf{right}(\alpha_m, \alpha_n))$ in configuration $\mathcal{M}_s$. After transition, target g_j disappears in $\mathcal{M}_t$. We have that $\forall g \neq g_j, \rho_t(g) = \rho_s(g)$, and that $\rho_s(g_j) = k_j$, but $\rho_t(g_j)$ is not defined. $\sigma_t = \sigma_s[\alpha_m \leftarrow V]$. There exists an evaluation context $E[\ ]$, such that

$$e_{is} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]] \text{ and } e_{it} \equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_t}_{\sigma_t}[\![V]\!]\ \mathcal{T}^{\rho_t}_{\sigma_t}[\![\sigma_t(\alpha_n)]\!]]$$

In $\mathcal{M}_s$, the term associated with target g_j is $e_{j\,s} \equiv \mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma_s(\alpha_n)]\!]$. We can find a transition pcall that has allocated locations α_m and α_n. Let s' be this transition. By inductive hypothesis and by Lemma 19, we have that

$$\mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_{s'}}_{\sigma_{s'}}[\![N]\!]\{c_u/k_v\}^* = \mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!] = \mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma_s(\alpha_n)]\!]$$

with $k_j \notin FV(\mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!])$, with k_u, k_v such that $\exists g_u, g_v, \rho_{s'}(g_u) = \rho_s(g_u) = k_u$, $\rho_{s'}(g_v) = k_v, \rho_{s'}(g_v) = \bot$, with $\kappa_v \sqsupseteq^s_{\sigma_t} \kappa_u$, and $c_u \equiv \mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa_v\rangle]\!]$. So,

$$\begin{aligned}
e_{is} &\equiv \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]] \\
&= \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]] \text{ by } (C_{elim}) \\
&= \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ \mathsf{callcc}\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma_s(\alpha_n)]\!]] \text{ by Inductive Hypothesis} \\
&= \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ ((\lambda k_j.\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma_s(\alpha_n)]\!])\ (\lambda v.\mathcal{A}(k_i E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ v])))] \\
&\qquad \text{by } (C_{lift}) \text{ and } (C_{idem}) \\
&= \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ (\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma(\alpha_n)]\!]\{(\lambda v.\mathcal{A}(k_i E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ v]))/k_j\})] \text{ by } (\beta_v) \\
&= \mathsf{callcc}\lambda k_i.E[\mathcal{T}^{\rho_t}_{\sigma_t}[\![V]\!]\ \mathcal{T}^{\rho_t}_{\sigma_t}[\![\sigma_t(\alpha_n)]\!]] \equiv e_{it} \text{ by Lemma 19}
\end{aligned}$$

We have $\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!] \equiv \mathcal{T}^{\rho_t}_{\sigma_t}[\![V]\!]$ because the targets of any continuation point in F are the same in $\mathcal{M}_s$ and $\mathcal{M}_t$ (put differently, there is no continuation point in F with a target g_j, by Lemma 20).

Although the content of α_n has not changed during the transition, the translation $\mathcal{T}^{\rho_t}_{\sigma_t}[\![\sigma(\alpha_n)]\!]$ differs from $\mathcal{T}^{\rho_s}_{\sigma_s}[\![\sigma(\alpha_n)]\!]$ because g_j is no longer a target in $\mathcal{M}_t$. Each continuation that belongs to target g_j in configuration $\mathcal{M}_s$ belongs to target g_i in configuration $\mathcal{M}_t$.

Now, let us consider any term e_{us} which is not associated with targets g_i or g_j. We have $e_{us}\{(\lambda v.\mathcal{A}(k_i E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]\ v]))/k_j\} = e_{ut}$ by Lemma 19.

Both invoke_{init} and invoke_{up} rely on Lemma 20, and their soundness is proved by C_{cur}. For instance,

(invoke_{up}) $\rho_s = \rho_t$ and $\sigma_s = \sigma_t$. Let $g_i \equiv \langle \alpha_i, \kappa_i \rangle$ be the target of $\langle V, ((\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))\ \mathbf{fun}\langle p, \kappa\rangle)\rangle$, with $\kappa_i \equiv (\kappa_1\ \mathbf{right}(\alpha_m, \alpha_n))$ and $\rho_s(g_i) = \rho_t(g_i) = k_i$.

By Lemma 20, since $\langle p, \kappa\rangle$ is accessible by the process, the target $g_j \equiv \langle \alpha_j, \kappa_j\rangle$ of κ must be an ancestor of target g_i. So, we have $\kappa \sqsupseteq^s_{\sigma_s} \kappa_j$, and $\kappa \sqsupseteq \kappa_i \sqsupseteq \kappa_j$. Therefore, we have $\kappa \sqsupseteq^s_{\sigma_s} \kappa_i \sqsupseteq^s_{\sigma_s} \kappa_j$. Since κ_i and κ_j are both continuations of targets, $\kappa_i \equiv \kappa_j$. Hence, there exists an evaluation context $E[\ ]$, such that

$$e_{is} \equiv \mathsf{callcc}\lambda k_i.(\mathcal{T}^{\rho_s}_{\sigma_s}[\![\langle p, \kappa\rangle]\!]\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]) \equiv \mathsf{callcc}\lambda k_i.((\lambda v.\mathcal{A}k_i(E[v]))\ \mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!])$$
$$= \mathsf{callcc}\lambda k_i.(E[\mathcal{T}^{\rho_s}_{\sigma_s}[\![V]\!]]) \text{ by } (\beta_v) \text{ and } (C_{cur}) \quad \equiv e_{it}$$

□

Lemma 19 For all $M \in \Lambda_{pcks}$, for all g_u, g_v such that $\rho_s(g_u) = \rho_t(g_u) = k_u$, and $\rho_s(g_v) = k_v$ but $\rho_t(g_v)$ is undefined,

$$\mathcal{T}^{\rho_s}_{\sigma_s}[\![M]\!]\{c_u/k_v\}^* = \mathcal{T}^{\rho_t}_{\sigma_t}[\![M]\!]$$

with $c_u \equiv \mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa_v\rangle]\!]$, $\kappa_v \sqsupseteq^s_{\sigma_t} \kappa_u$, and $\{c_u/k_v\}^*$ denoting the substitution of all k_v by c_u. □

Proof Lemma 19. By induction on the size of M. The interesting case is the continuation point. Let $g_i \equiv \langle \alpha_i, \kappa_i\rangle$ be the target of κ in $\mathcal{M}_s$, $\kappa \sqsupseteq^s_{\sigma_s} \kappa_i$, and let $g_j \equiv \langle \alpha_j, \kappa_j\rangle$ be the target of κ in $\mathcal{M}_t$, $\kappa \sqsupseteq^s_{\sigma_t} \kappa_j$, with $\kappa_i \sqsupseteq^s_{\sigma_t} \kappa_j$.

$$\mathcal{T}^{\rho_s}_{\sigma_s}[\![\langle p, \kappa\rangle]\!]\{c_u/k_v\}^*$$
$$\equiv (\lambda v.\mathcal{A}(k_i\{c_u/k_v\}^*\ \mathcal{C}^{\rho_s}_{\sigma_s}[\![\kappa, v]\!]\{c_u/k_v\}^*)) \equiv (\lambda v.\mathcal{A}(c_j\ \mathcal{C}^{\rho_s}_{\sigma_s}[\![\kappa, v]\!]\{c_u/k_v\}^*))$$
$$\equiv (\lambda v.\mathcal{A}((\lambda v.\mathcal{A}(k_j\ \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa_i, v]\!]))\ \mathcal{C}^{\rho_s}_{\sigma_s}[\![\kappa, v]\!]\{c_u/k_v\}^*))$$
$$= (\lambda v.\mathcal{A}((\lambda v.\mathcal{A}(k_j\ \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa_i, v]\!]))\ E'[v])) \ (*), \text{ by IH}$$
$$= (\lambda v.\mathcal{A}(k_j\, \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa_i, E'[v]]\!])) \text{ by } \beta'_{\Omega} \text{ and } \mathsf{abort}$$
$$= (\lambda v.\mathcal{A}(k_j\, \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa, v]\!])) \text{ by dfn. of } \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa, v]\!] \quad \equiv \mathcal{T}^{\rho_t}_{\sigma_t}[\![\langle p, \kappa\rangle]\!]$$

(*) $E'[\ ]$ is obtained by replacing in $E[\ ] \equiv \mathcal{C}^{\rho_t}_{\sigma_t}[\![\kappa_i, [\ ]]\!]$ all terms that have a free variable k_v : $E'[\ ] \equiv E[\ ]\{\mathcal{T}^{\rho_t}_{\sigma_t}[\![N]\!]/\mathcal{T}^{\rho_s}_{\sigma_s}[\![N]\!]\}$. □

Let us prove that the target of a continuation is always an ancestor of the target of a process that can access it.

Lemma 20 If $\langle p, \kappa\rangle$ is a subexpression of M in process $\langle M, \kappa'\rangle$. Let κ_i be the continuation of the target of κ, and let κ'_i be the continuation of the target of κ'. Then $\kappa'_i \sqsupseteq \kappa_i$. □

Sketch of Proof of Theorem 11

If $\mathsf{eval}|_{\mathsf{pcks}}(M) = V$, we know that $\lambda_v\text{-}C \vdash \mathcal{T}^{\rho}_{\sigma}[\![M]\!] = \mathcal{T}^{\rho}_{\sigma}[\![V]\!]$ by Lemma 18. So $\mathsf{eval}|_{\mathsf{vc}}(\mathcal{S}[\![M]\!])$ is defined. By Felleisen and Friedman's Corollary 5.8 and Theorem 5.1 [4], $\mathsf{eval}|_{\mathsf{ck}}(\mathcal{S}[\![M]\!])$.

Symmetrically, if $\mathsf{eval}|_{\mathsf{ck}}(\mathcal{S}[\![M]\!])$ is defined, then $\mathsf{eval}|_{\mathsf{pcks}}(M)$ is also defined because the PCKS-machine is able to simulate the CK-machine by always reducing the mandatory process. □

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.
2. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
3. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.
4. Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers.
5. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995.
6. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol. 28, pages 237–247, 1993.
7. Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.
8. Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.
9. 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.
10. 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.
11. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.
12. 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.
13. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.
14. Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.
15. 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.
16. Christopher Strachey and Christopher P. Wadsworth. A Mathematical Semantics for Handling Full Jumps. Technical Monography PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.

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