Sound Rules for Parallel Evaluation of a Functional Language with callcc

Luc Moreau and Daniel Ribbens
Service d'Informatique
Institut d'Electricité Montefiore, B28
University of Liège (Sart-Tilman)
4000 Liège Belgium
moreau@montefiore.ulg.ac.be ribbens@montefiore.ulg.ac.be

Abstract

Observationally equivalent programs are programs which are indistinguishable in all contexts, as far as their termination property is concerned. In this paper, we present rules preserving observational equivalence, for the parallel evaluation of programs using `call/cc`. These rules allow the capture of continuations in any applicative context and they prevent from aborting the whole computation when a continuation is applied in the extent of the `call/cc` by which it was reified. As a consequence, these results prove that one can design a functional language with first-class continuations which has transparent constructs for parallelism.

1 Introduction

Some programming languages, like Scheme and Standard ML of New Jersey, provide a control operator `call/cc` which gives the programmer the possibility to reify the current continuation as a first-class object. When such a reified continuation is applied to a value v, the current computation is aborted and the execution resumes at the point where the continuation was captured; the value v being the value returned from this `call/cc` expression. When parallelism is introduced in such languages the meaning of continuations does not appear to be clear. There have been several attempts to give a semantics to continuations and parallelism [11], [12], [16], [17], [22]. In a previous paper [19], we presented a new semantics for a functional language with continuations and transparent constructs for parallelism. In this paper, we formalise this approach and prove some interesting properties.

1.1 Intuitive semantics

A construct for parallelism is said to be *transparent* if all programs using this construct return the same result as those programs written without this construct. Thus, a transparent construct can be seen as an annotation for parallel execution which preserves the meaning of programs. Thanks to this property, parallel applications can be developed in two phases: programs can be written using the functional programming methodology and then, they can be annotated by constructs for parallelism as described in [18].

Let us consider the purely functional subset of Scheme extended with `call/cc` and let us suppose that evaluation is sequential (left-to-right order) unless parallelism is introduced by the construct `pcall`. `pcall` requires two arguments; `(pcall f e)` applies the value of `f` to the value of `e` after evaluation of `f` and `e` in parallel.

Let us show by several examples the behaviour of programs using simultaneously `call/cc` and `pcall`. In the following program, which is the same as program 1 (in figure 1) without the `pcall` annotation,

```
(call/cc (lambda (k)
           ((f1 (k 1)) (k 2))))
```

`k` is applied to 1 because evaluation is supposed to be from left to right. In program 1, the evaluations of `(k 1)` and `(k 2)` proceed in parallel. Since `pcall` is transparent, only one application can actually be performed, and it must be `k` to `1`. In fact, with a transparent `pcall`, `k` must not be applied to 2 because expression `(f1 (k 1))` escapes and this expression, appearing to the left of `(k 2)`, is evaluated before `(k 2)` in the sequential program. We can state this intuitive rule: *before applying a continuation in a parallel program, expressions appearing to the left of the application of this continuation should have returned a value.*

```
(call/cc (lambda (k)
            (pcall (f1 (k 1)) (k 2))))                    1

(pcall ⟨e⟩ (call/cc (lambda (exit)                        2
                      ( ..  (exit 2)))))

(pcall (call/cc (lambda (k1) ...))                        3
       (call/cc (lambda (k2) ...)))
```

Figure 1: Three small examples

In the following example,

```
(let ((α (call/cc (lambda (x) x))))
  (α (lambda (x) x)))
```

the continuation returned by the `call/cc` expression is applied outside this `call/cc` expression. The continuation is said to be applied outside the extent of this `call/cc`. On the other hand, in program 2 (in figure 1), the continuation bound to `exit` is applied while evaluating the `call/cc` expression, i.e. in its extent. If we apply our intuitive rule to program 2, we have to wait for the value of $\langle e \rangle$ before

To appear in the proceedings of the FPCA'93 conference, Copenhagen, June 93, ACM.

applying `exit` to 2 because $\langle e \rangle$ appears to the left of `(exit 2)`. However, we understand that we can invoke `exit` independently of the behaviour of $\langle e \rangle$, since the application of `exit` aims at returning the value of the `call/cc` expression.

In program 3, we would like to capture two continuations in parallel. Intuitively, there does not seem to be any reason to sequentialise those captures: indeed, the continuation to be bound to `k2` can be found independently of the value of `(call/cc (lambda (k1) ...))` and vice-versa.

Those three examples illustrate the principles about the parallel evaluation we want to formalise using syntactic theories. First, continuations can be applied if expressions to their left have returned a value. Second, if a continuation is applied in the extent T of the `call/cc` by which it was reified, one has to consider only expressions appearing to the left and which have an extent which is included in T. Third, a continuation can be captured in any context, independently of expressions appearing to their left.

1.2 Syntactic theories of control

Syntactic theories of control were introduced by Felleisen *et al.* [5], [7], [8], [6]. These theories extend the call-by-value λ-calculus defined by Plotkin [21] with control operators like $\mathcal{C}$ and $\mathcal{A}$. $\mathcal{C}$ allows to capture a continuation and $\mathcal{A}$ aborts a computation. Felleisen *et al.* proved these systems to be Church-Rosser. This property states that if M reduces to P and M reduces to Q by different reduction paths, there exists N such that P and Q reduce to N. This entails that there is an evaluation function.

Initially, we defined a reduction system which was also an extension of Plotkin's call-by-value λ-calculus with control operators callcc and $\mathcal{A}$. It was composed of several reduction rules: the β-value reduction, rules to eliminate callcc (i.e. reify a continuation) in *any* context and rules to apply a continuation. Those rules were supposed to be a formalisation of the intuitive semantics given in section 1.1. We defined a leftmost, outermost reduction strategy which corresponds to a sequential (left-to-right evaluation order) semantics. A parallel evaluation was also possible in this system since several redices could be reduced at each evaluation step.

Unfortunately, it appeared that such a system was not Church-Rosser: we could find a program M which reduced to P and Q by the sequential and parallel strategy but we could not find N such that both P and Q reduced to N. The rule which made the Church-Rosser property collapse was the rule which allowed the capture of continuation in any context. Although Church-Rosser is a nice property, it appears to be too strong for our purpose. Indeed, there are programs which cannot be proved to be equal although they behave "*externally*" in the same way, in any context they are used. This notion of equivalence is usually called observational equivalence. Given two observational equivalent programs, there is no context in which one program terminates and the other not: both programs are indistinguishable as far as termination is concerned.

Therefore, we adopted another approach to formalise the intuitive semantics of section 1.1. We initially defined a core reduction system C, which extends the call-by-value λ-calculus, with control operators callcc and $\mathcal{A}$. Unlike our first attempt, we do not allow the capture of continuation in all contexts and we do not consider the problem of the application of a continuation in the extent of the callcc by which it was reified. This system C looks very like Felleisen's theory [5] and it suffers from the same defaults: a parallel evaluation strategy can be defined but far less parallelism is obtained than in our intuitive semantics because control operators are bottlenecks. Although it does not capture our intuitive semantics, this system is interesting since it can be proved to be Church-Rosser and can be used to define a notion of observational equivalence. Then, we extend this initial system with a set of equations which encode the intuitive semantics. We show that those equations preserve observational equivalence. It means that programs reduced with those equations and the same programs reduced without them return results which are observationally equivalent. Our goal is reached: we can define a sequential reduction strategy and a parallel reduction strategy (with the semantics of 1.1) for which returned results are observationally indistinguishable. It means that parallel evaluation does not change the meaning of programs and transparent constructs can be added to a sequential language and be considered as annotations for execution.

This paper is organised as follows. In section 2, we present the core reduction system C and in section 3, we define the observational equivalence in C and prove a major result relating the CPS translation to the observational equivalence. In section 4, we extend C by equations for the capture of continuations in any context and equations which improve the evaluation, all of them being proved to be sound. At this point, our reduction system is able to evaluate examples 1 and 3 as explained in section 1.1. The optimisation of the application of a continuation in the extent of a callcc requires a new representation of continuation objects and a new syntactic construct to mark the extent. This is presented in two steps: in section 5, we describe an intermediate reduction system CP (for Continuation Points) which allows to uniquely name a reified continuation. In section 6, we define the system CPP (for Continuation Points and Prompts) with a mechanism of prompt delimiting the extent of a callcc. This system is able to evaluate example 2 as explained in section 1.1. All reduction systems C, CP and CPP are proved to have the same notions of observational equivalence.

2 The C calculus

Let us consider Λ_c, the language defined by

$$\text{Term } M ::= \begin{cases} V & \text{Value} \\ (M\,M) & \text{Application} \\ (\text{callcc } M) & \text{callcc-Application} \\ (\mathcal{A}\ M) & \mathcal{A}\text{-Application} \end{cases}$$

$$\text{Value } V ::= \begin{cases} a, b, \ldots & \text{Constants} \\ x, y, \ldots & \text{Variables} \\ \lambda x.M & \text{Lambda Abstraction} \end{cases}$$

and let us use Barendregt's [1] conventions, for substitution, free variables, closed values. This language is a λ-calculus extended with two control operators: callcc which captures the current continuation and $\mathcal{A}$ which aborts a computation. We also define several kinds of contexts used in this paper, Contexts $C[\,]$, Applicative $A[\,]$, Evaluation contexts $E[\,]$:

$$\begin{array}{lcl} C[\,] & ::= & [\,] \mid C[M[\,]] \mid C[[\,]M] \mid C[\lambda x.[\,]] \\ & & \quad \mid C[\text{callcc } [\,]] \mid C[\mathcal{A}\ [\,]] \\ A[\,] & ::= & [\,] \mid A[M[\,]] \mid A[[\,]M] \\ E[\,] & ::= & [\,] \mid E[V[\,]] \mid E[[\,]M] \end{array}$$

$(\lambda x.M)V$	$\stackrel{\beta_v}{\rightarrow}$	$M\{V/x\}$ with V a value	(C1)
(ab)	$\stackrel{\delta}{\rightarrow}$	$\delta_c(a,b)$ if this is defined	(C2)
$M(\mathsf{callcc}\ N)$	$\stackrel{\mathsf{callcc}_R}{\rightarrow}$	$\mathsf{callcc}\ \lambda k.M(N\ (\lambda v.\mathcal{A}(k(Mv))))$ with M a value	(C3)
$(\mathsf{callcc}\ M)N$	$\stackrel{\mathsf{callcc}_L}{\rightarrow}$	$\mathsf{callcc}\ \lambda k.(M(\lambda f.\mathcal{A}(k(fN))))N$	(C4)
$(\mathcal{A}M)N$	$\stackrel{\mathcal{A}_L}{\rightarrow}$	$\mathcal{A}M$	(C5)
$M(\mathcal{A}N)$	$\stackrel{\mathcal{A}_R}{\rightarrow}$	$\mathcal{A}N$ with M a value	(C6)
$\mathcal{A}\mathcal{A}M$	$\stackrel{\mathcal{A}_{idem}}{\rightarrow}$	$\mathcal{A}M$	(C7)
$\mathcal{A}(\mathsf{callcc}\ M)$	$\stackrel{\mathsf{callcc}\ _{in\mathcal{A}}}{\rightarrow}$	$\mathcal{A}(M(\lambda x.\mathcal{A}x))$	(C8)

Figure 2: One step reduction $\rightarrow_c$ for C

The C reduction system is defined by the set of rules in figure 2; it is similar to Felleisen's reduction system [8]. Let us sketch some of its features. The first rule is the β-value reduction [21] and the second rule is the δ-reduction applying a primitive a to a constant b. As Plotkin [21], we suppose that δ is defined on the following domain: Constants $\times$ Constants $\rightarrow$ Closed Value. Rules C5 and C6 are the same as in [8]; in an application $(\mathcal{A}M)N$, when the expression in operator position aborts, the operand N is removed and, in $M(\mathcal{A}N)$, an expression in operand position can abort if the expression in operator position M is a value.

Rules C3 and C4 are adapted from Felleisen's rules for $\mathcal{C}$. callcc is intended to reify the current continuation and to apply its argument to it. In this reduction system, the current continuation is reified as a functional abstraction, which is built step by step according to the context in which callcc appears. The two rules allow callcc to appear in an application either in operator or operand position. Let us consider rule C3, rule C4 being symmetric. Supposing that the continuation of $M(\mathsf{callcc}\ N)$ is k, the continuation of callcc N is represented by $\lambda v.\mathcal{A}(k(Mv))$, an abortive function waiting for a value, applying M to this value, the result being transmitted to the continuation k. The value of the continuation k is also found by a callcc.

By successively, applying rules C3 and C4, a callcc-expression appearing internally in a expression is moved step by step, inside-out, until it appears at the top level of this expression. Instead of using computation rules as Felleisen, we suppose that a computation is performed in an $\mathcal{A}$-application. Rule C8 transforms a callcc at the top level by an application of its argument to the initial continuation $(\lambda x.\mathcal{A}x)$ and similarly, rule C7 eliminates an $\mathcal{A}$ at the top level.

As long as no callcc or $\mathcal{A}$ appears in a program, this reduction system behaves as the call-by-value λ-calculus. When a callcc is used in a program, it is intended to capture its current continuation. This continuation is represented by a functional abstraction of the context which is built step by step by *bubbling up* [6] callcc-applications (using rules C3 and C4) until a callcc reaches the top level. This phase, called the *construction phase* [7], accumulates all "application frames". At this point, rule C8 can be applied (since we suppose that evaluation proceeds in an $\mathcal{A}$-application) and a series of β_v reductions are performed; this phase, called the *collection phase* [7], "concatenates all application frames". After these two phases, we are in the situation where the initial callcc-application is replaced by the application of its argument to the functional abstraction of the context. If a continuation is applied to a value, it immediately aborts the current computation (with $\mathcal{A}$). In order to abort a computation, terms are pruned step by step using rules C5 and C6 until the top level is reached where rule C7 can be applied.

With the first six rules, we can define a notion of reduction $\rightarrow^c$

$$\rightarrow^c = \stackrel{\beta_v}{\rightarrow} \cup \stackrel{\delta}{\rightarrow} \cup \stackrel{\mathsf{callcc}_R}{\rightarrow} \cup \stackrel{\mathsf{callcc}_L}{\rightarrow} \cup \stackrel{\mathcal{A}_R}{\rightarrow} \cup \stackrel{\mathcal{A}_L}{\rightarrow}$$

and its compatible closure $\rightarrow_c$ is defined by

$$\begin{array}{lcl} M \rightarrow^c N & \Rightarrow & M \rightarrow_c N \\ M \rightarrow_c N & \Rightarrow & \lambda x.M \rightarrow_c \lambda x.N \\ M \rightarrow_c N & \Rightarrow & ZM \rightarrow_c ZN, MZ \rightarrow_c NZ, \text{with } Z \in \Lambda_c \\ M \rightarrow_c N & \Rightarrow & \mathsf{callcc}\ M \rightarrow_c \mathsf{callcc}\ N \\ M \rightarrow_c N & \Rightarrow & \mathcal{A}\ M \rightarrow_c \mathcal{A}\ N \end{array}$$

Classically we define $\rightarrow_c^*$, the reflexive, transitive closure of $\rightarrow_c$ and $=_c$ the equivalence relation generated by $\rightarrow_c^*$.

We suppose that an evaluation is performed within an $\mathcal{A}$ operator. Rules C7 and C8 can only be used at the top level. We call them *top level rules*. Those rules are different from rules C1 to C6 since we do not define their compatible closures. We define a *computation* $\rightarrow_c^{\triangleright}$ by

$$\rightarrow_c^{\triangleright} = \rightarrow_c^* \cup \stackrel{\mathcal{A}_{idem}}{\rightarrow} \cup \stackrel{\mathsf{callcc}\ _{in\mathcal{A}}}{\rightarrow}$$

and we note $\rightarrow_c^{\triangleright *}$ its transitive closure.

The C reduction system has the following properties:

Theorem 2.1 (Church-Rosser)

- *The notion of reduction $\rightarrow^c$ is Church-Rosser.*
- *The relation $\rightarrow_c^{\triangleright}$ satisfies the diamond property.*

Sketch of Proof of Theorem 2.1

The proof is similar to the one proposed by Felleisen [8]. First, from the definition of $\rightarrow^c$, we define a parallel reduction $\twoheadrightarrow_\ell$. We prove the diamond property for $\twoheadrightarrow_\ell$ which leads to the diamond property of $\rightarrow_c^*$. Then we show that $\rightarrow_c^{\triangleright}$ also satisfies the diamond property. □

We can abstract the evaluation process by $\mathsf{eval_C}$:

Definition 2.2 (Eval) $\mathsf{eval_C}(M) = V$ *iff* $\mathcal{A}M \rightarrow_c^{\triangleright *} \mathcal{A}V$ *with* V *a value.*

$\mathsf{eval}_{\mathsf{C}}$ is defined if the reduction $\mathcal{A}M \rightarrow_c^{\triangleright *} \mathcal{A}V$ terminates, else it is undefined. We can also define a standard reduction function $\mapsto_s$ which reduces the leftmost, outermost redex and which does not allow reduction under λ-abstraction, $\mathcal{A}$ and callcc-applications. We naturally extend it to a standard reduction function with top level reduction $\mapsto_s^T$.

Definition 2.3 (Standard Reduction Function $\mapsto_s$)

$$\begin{array}{rcl} M \rightarrow^c N & \Rightarrow & M \mapsto_s N \\ M \mapsto_s M' & \Rightarrow & MN \mapsto_s M'N \\ N \mapsto_s N' & \Rightarrow & MN \mapsto_s MN' \text{ with } M \text{ is a value} \\ M \mapsto_s^T N & \Leftrightarrow & \mathcal{A}M \overset{\mathsf{callcc}\ in\mathcal{A}}{\rightarrow} \mathcal{A}N \text{ or } \mathcal{A}M \overset{\mathcal{A}_{id\mathcal{E}m}}{\rightarrow} \mathcal{A}N \\ & & \text{or } M \mapsto_s N \end{array}$$

These standard reductions are related to the notion of evaluation by the following theorem:

Theorem 2.4 $\mathsf{eval}_{\mathsf{C}} M = V$ *iff* $M \mapsto_s^{T^*} V'$ *for some value* V'.

Sketch of Proof of Theorem 2.4

Again, similar to Felleisen's [8] and Plotkin's [21] proofs. We define a standard reduction sequence and show that for any reduction $M \rightarrow_c^* N$ there is a standard reduction sequence $M, \ldots, N$. Then we extend standard reduction sequences to standard reduction sequences with top level reductions and relate them to the notion of evaluation. □

The standardisation theorem and the Church-Rosser property entail that there is an evaluation function, and that there is an algorithm to compute it which corresponds to a left-to-right evaluation order. We can also define a parallel evaluation strategy where the subexpressions of an application are evaluated in parallel, and where evaluation can be performed under an $\mathcal{A}$ or callcc.

Definition 2.5 (Parallel Evaluation Strategy)

$$\begin{array}{rcl} M \rightarrow^c N & \Rightarrow & M \rightarrow_p N \\ & & M \rightarrow_p M \\ M \rightarrow_p M', N \rightarrow_p N' & \Rightarrow & MN \rightarrow_p M'N' \\ M \rightarrow_p N & \Rightarrow & \mathsf{callcc}\ M \rightarrow_p \mathsf{callcc}\ N \\ M \rightarrow_p N & \Rightarrow & \mathcal{A}\ M \rightarrow_p \mathcal{A}\ N \end{array}$$

We call $\rightarrow_p^*$ *the reflexive, transitive closure of* $\rightarrow_p$ *and* $M \rightarrow_p^{\triangleright} N$ *is defined by* $M \rightarrow_p^* N \cup \mathcal{A}M \overset{\mathcal{A}_{id\mathcal{E}m}}{\rightarrow} \mathcal{A}N \cup \mathcal{A}M \overset{\mathsf{callcc}\ in\mathcal{A}}{\rightarrow} \mathcal{A}N$.

By examining the parallel evaluation strategy of C, we can conclude that it does not allow parallel evaluation as we described in section 1.1. As a matter of fact, in order to capture a continuation, callcc must be bubbled up to the top level; this requires all expressions appearing to the left of callcc to be values, i.e. a continuation can be captured in an evaluation context. Moreover, when a continuation is applied, expressions appearing to the left, up to the top level, are pruned if they are values but there is no optimisation when the continuation is applied in extent of the callcc by which it was reified.

Consequently, we could change the side condition of rule C3 to solve the first problem: we could allow the capture of the context even if M is not a value:

$$M(\mathsf{callcc}\ N) \rightarrow \mathsf{callcc}\ \lambda k.M(N\ (\lambda v.\mathcal{A}(k(Mv))))$$

Unfortunately, adding such a rule makes the Church-Rosser property disappear. However, we can define a more general notion of equivalence based on C which is called *observational equivalence*.

3 Observational Equivalence

From a programmer's point of view, two behaviours can be observed: 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 3.1 (Observational Equivalence) $M \cong_c N$ *iff* $\forall$ *context* $C[\,]$, *such that* $C[M]$ *and* $C[N]$ *are programs, either both* $\mathsf{eval}_{\mathsf{C}}(C[M])$ *and* $\mathsf{eval}_{\mathsf{C}}(C[N])$ *are defined or both are undefined.*

Observational equivalence allows to prove the correctness of some *optimisations*.

> Abstractly, an optimisation of a program $C[M]$ is the replacement of M by a more "efficient" expression N such that a programmer cannot distinguish the observational behaviour of the programs $C[M]$ and $C[N]$. [26, section 2,page 230]

Therefore, a correct optimisation is an equation $M = N$ such that $M \cong_c N$. It is our intention to define some optimisations which allow parallel evaluation in the sense of section 1.1. Proving the observational equivalence of two terms is not an immediate task but Plotkin gave a powerful technique for this purpose relating the CPS translation to the observational equivalence.

The CPS translation is an old idea in computer science. It was first formalised by Fischer and Reynolds [10], [25]; it is defined by the following equations where $[\![.]\!]$ maps a call-by-value term to a lambda-term:

Definition 3.2 (CPS translation)

$$\begin{array}{rcll} [\![V]\!] & = & \lambda k.k\Psi(V) \text{ with } V \text{ a value} & \text{(cps1)} \\ [\![(MN)]\!] & = & \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnk)) & \text{(cps2)} \\ \Psi(x) & = & x \text{ with } x \text{ a variable or constant} & \text{(cps3)} \\ \Psi(\lambda x.M) & = & (\lambda x.[\![M]\!]) & \text{(cps4)} \end{array}$$

Some essential properties of the CPS translation were proved by Plotkin; λ_v and λ_n represent the call-by-value and call-by-name theories respectively, $\mathsf{eval}_{\mathsf{V}}$ and $\mathsf{eval}_{\mathsf{N}}$, the corresponding evaluation functions, $\cong_v$, observational equivalence in λ_v.

Theorem 3.3 (Plotkin)

1. $\Psi(\mathsf{eval}_{\mathsf{V}}(M)) = \mathsf{eval}_{\mathsf{N}}([\![M]\!](\lambda x.x))$*: the value of* M *according to the call-by-value evaluation strategy is related to the value of the CPS translation of* M *according to the call-by-name strategy.*
2. $\lambda_v \vdash M = N \Rightarrow \lambda_n \vdash [\![M]\!] = [\![N]\!]$*: the call-by-value* λ*-calculus is sound with respect to the CPS translation (but it is not complete).*
3. $\lambda_n \vdash [\![M]\!] = [\![N]\!] \Rightarrow M \cong_v N$*: equality in* λ_n *of the CPS translations of* M *and* N *implies observational equivalence of* M *and* N *in* λ_v *(the converse does not hold).*

We proved similar results for C:

Theorem 3.4 (Simulation)

$$\Psi_o(\mathsf{eval}_\mathsf{C}(M)) = \mathsf{eval}_\mathsf{N}([\![M]\!]\lambda x.x)$$

Theorem 3.5 $\lambda_n \vdash [\![M]\!] = [\![N]\!] \;\Rightarrow\; M \cong_c N$

Sketch of Proof of Theorems 3.4, 3.5

Theorem 3.5 comes from theorem 3.4 as Plotkin's corollary 2 comes from theorem 6.2 [21]. For theorem 3.4, we adopt the same technique as Plotkin in theorem 6.2. If $\mathsf{eval}_\mathsf{C}(M)$ is defined, there is a leftmost, outermost reduction path $M \mapsto_s^T M_1 \mapsto_s^T M_2 \mapsto_s^T \ldots V$ by theorem 2.4. By lemma A.1, we can conclude that

$$[\![M]\!]_o^{\lambda x.x} \to^* [\![M_1]\!]_o^{\lambda x.x} \to^* \ldots [\![V]\!]_o^{\lambda x.x} \equiv \Psi_o(V)$$

which concludes the proof since $\mathsf{eval}_\mathsf{C}(M) = V$. □

We now have the tool to prove that two expressions are observationally equivalent. In the following section we present some optimisations which preserve the observational equivalence. The appendix is dedicated to the description of the optimised cps translation that we use in theorem 3.4.

4 Optimisations and Parallel evaluation

Using theorem 3.5, it is now easy to state some optimisations which are suitable for parallel evaluation. An optimisation is an equation $M = N$, where $\lambda_n \vdash [\![M]\!] = [\![N]\!]$; this implies $M \cong_c N$. A set of such optimisations is displayed in figure 3.

With equation OPT1, it is allowed to capture a continuation in any applicative context. This rule is essentially the same as rule C3 where the side condition is removed. One should also note that in the right-hand side,

$$\mathsf{callcc}\ \lambda k.(\lambda f.f(N\ (\lambda v.\mathcal{A}(k(fv))))))M$$

the expression M is not duplicated; this was necessary to have equality between CPS translations of both sides of this equation. This has a strange consequence: one could have expected that if the application of N to the continuation reduces to another callcc-application, a new continuation could have been captured. Unfortunately, this new callcc-application does not appear in an applicative context but under a λ-expression; however, rule OPT2 gives a solution to this problem, allowing to reduce several callcc to only one. This is also the purpose of equation OPT3. Equations OPT4 and OPT5 allow to simplify some callcc-applications independently of the context. This bunch of equations encodes the rules 1 and 3 of the semantics given in 1.1.

Rule OPT6 is the equivalent of $\mathcal{C}_{top}$ [9]; it allows to evaluate a callcc-application without capturing the context by applying the argument to a continuation $\lambda v.\mathcal{A}(kv)$.

The equations OPT7 and OPT8 show that the top level rules C7 and C8 can, in fact, be used in any context and that, they preserve observational equivalence.

Equations OPT9 to OPT12 are presented here and will be used in several proofs in the following sections. Equation OPT9 proves that a special operator like $\mathcal{A}$ is not necessary. Indeed, $(\lambda x.\mathcal{A}x)$ represents the initial continuation and equation OPT9 says that every expression $\mathcal{A}M$ can be replaced by the application of the initial continuation to M. This equation is generalised by equation OPT10 for any evaluation context. We can even further generalise this equation with equation OPT12, where $K[\ ]$ is a captured context which will be defined in the following section.

5 The CP calculus

There is still a notion we presented in the intuitive semantics which is not yet axiomatised: when a continuation is applied in the extent of the callcc by which it was reified, it is not necessary to abort the whole computation; it is sufficient to abort the computation up to this callcc. We intend to formalise this idea by a prompt mechanism which is explained in the following section. In this section, we present an intermediate system where continuations can be uniquely named.

In C, continuations are represented by anonymous functions and the abortive effect comes from the $\mathcal{A}$ operator. In CP (standing for Continuation Points), we introduce a new object $\langle p, K[\ ]_p\rangle$, called *continuation point* which abstracts a context $K[\ ]_p$. A continuation point object is given a name p which is also given to the hole of the context. When the name of the continuation point is unimportant (which is the case in CP), we conventionally use p. CP is based on the following language Λ_{cp}

$$\text{Term } M \quad ::= \quad \begin{cases} V & \text{Value} \\ (MM) & \text{Application} \\ (\mathsf{callcc}\ M) & \text{callcc-Application} \end{cases}$$

$$\text{Value } V \quad ::= \quad \begin{cases} a, b, \ldots & \text{Constants} \\ x, y, \ldots & \text{Variables} \\ \lambda x.M & \text{Lambda Abstraction} \\ \langle p, K[\ ]_p\rangle & \text{Continuation point, } p \text{ name} \end{cases}$$

where $\mathcal{A}$-applications were removed since they were showed to be optional according to rule OPT9. The new reduction system CP is defined by the rules displayed in figure 4.

Rules CP1 and CP2 are the same as rules C1 and C2. Rules CP3 and CP4 allow the capture of a continuation in any applicative context; the continuation is now represented by a continuation point. Rules CP5 and CP6 model the fact that the application of a continuation is abortive.

Rules CP7 allows to reduce a callcc in a continuation. Top level reductions are performed with rules CP9 and CP10.

We call a *captured context*, the context K appearing in a continuation point. Such a context satisfies the following grammar

$$K ::= [\ ] \mid K[[\ ]M] \mid K[V[\ ]] \mid K[\mathsf{callcc}\lambda k.[\ ]] \mid K[(\lambda v.[\ ])V]$$

Rule CP9 allows the composition of captured contexts; this was proved to be sound in C according to optimisation OPT12.

Similarly to C, we can define $\to^{cp}$, a notion of reduction,

$$M \to^{cp} N \text{ if } M \to N \text{ using rules CP1 to CP8}$$

$\to_{cp}$ its compatible closure,

$$\begin{array}{lcl} M \to^{cp} N & \Rightarrow & M \to_{cp} N \\ M \to_{cp} N & \Rightarrow & \lambda x.M \to_{cp} \lambda x.N \\ M \to_{cp} N & \Rightarrow & ZM \to_{cp} ZN, MZ \to_{cp} NZ \text{ with } Z \in \Lambda_{cp} \\ M \to_{cp} N & \Rightarrow & \mathsf{callcc}\ M \to_{cp} \mathsf{callcc}\ N \\ M \to_{cp} N & \Rightarrow & \langle \alpha, M\rangle \to_{cp} \langle \alpha, N\rangle \end{array}$$

$\to_{cp}^*$ the reflexive, transitive closure of $\to_{cp}$, and $=_{cp}$ the equivalence relation generated by $\to_{cp}^*$.

Similarly to C, we call a computation the relation $\to_{cp}^{\triangleright}$ defined by

$$M \to_{cp}^{\triangleright} N = M \to_{cp}^* N \cup\ M \overset{\text{CP9}}{\to} N\ \cup M \overset{\text{CP10}}{\to} N$$

$M(\mathsf{callcc}\ N)$	$=$	$\mathsf{callcc}\ \lambda k.(\lambda f.f(N\ (\lambda v.\mathcal{A}(k(fv)))))M$	(OPT1)
$\mathsf{callcc}\lambda k.((\lambda f.f(\mathsf{callcc}\ \lambda k'.N))M)$	$=$	$\mathsf{callcc}\lambda k.((\lambda f.f((\lambda k'.N)(\lambda v.k(fv))))M)$	(OPT2)
$\mathsf{callcc}\lambda k.\mathsf{callcc}\lambda k'.M$	$=$	$\mathsf{callcc}\lambda k.(\lambda k'.M)k$	(OPT3)
$\mathsf{callcc}\lambda k.M$	$=$	M if $k \notin FV(M)$	(OPT4)
$\mathsf{callcc}\lambda k.(kM)$	$=$	M if $k \notin FV(M)$	(OPT5)
$\mathsf{callcc}\ M$	$=$	$\mathsf{callcc}(\lambda k.(M\lambda v.\mathcal{A}(kv)))$	(OPT6)
$\mathcal{A}\mathsf{callcc}\ M$	$=$	$\mathcal{A}M(\lambda x.\mathcal{A}x)$	(OPT7)
$\mathcal{A}\mathcal{A}M$	$=$	$\mathcal{A}M$	(OPT8)
$(\lambda x.\mathcal{A}x)M$	$=$	$\mathcal{A}M$	(OPT9)
$((\lambda x.\mathcal{A}E[x])M)$	$=$	$\mathcal{A}(E[M])$ with E an evaluation context	(OPT10)
$(\mathcal{A}\ ((\lambda x.(\mathcal{A}\ (\mathsf{callcc}\ x)))Q))$	$=$	$(\mathcal{A}\ (\mathsf{callcc}\ Q))$	(OPT11)
$((\lambda x.\mathcal{A}K[x])M)$	$=$	$\mathcal{A}(K[M])$ with K an captured context	(OPT12)

Figure 3: Optimisations for parallel evaluation

$(\lambda x.M)V$	$\rightarrow$	$M\{V/x\}$ with V a value	(CP1)
(ab)	$\rightarrow$	$\delta_{cp}(a,b)$ if this is defined	(CP2)
$M(\mathsf{callcc}\ N)$	$\rightarrow$	$\mathsf{callcc}\ \lambda k.(\lambda f.f(N\langle\alpha, k(f\ [\ ]_\alpha)\rangle))M$	(CP3)
$(\mathsf{callcc}\ M)N$	$\rightarrow$	$\mathsf{callcc}\ \lambda k.(M\langle\alpha, k([\ ]_\alpha\ N)\rangle)N$	(CP4)
$(\langle p, K[\ ]_p\rangle\ V)N$	$\rightarrow$	$(\langle p, K[\ ]_p\rangle\ V)$ with V a value	(CP5)
$M(\langle p, K[\ ]_p\rangle\ V)$	$\rightarrow$	$(\langle p, K[\ ]_p\rangle\ V)$ with M, V values	(CP6)
$\langle\alpha, (\mathsf{callcc}\ M)\rangle$	$\rightarrow$	$\langle\alpha, (M\ \langle\delta, [\ ]_\delta\rangle)\rangle$	(CP7)
$\langle\alpha, \langle\beta, K_1[\ ]_\beta\rangle\ (K_2[\ ]_\alpha)\rangle$	$\rightarrow$	$\langle\alpha, K_1[K_2[\ ]_\alpha]\rangle$	(CP8)
$(\langle p, K[\ ]_p\rangle\ V)$	$\rightarrow^T$	$K[V]$ with V a value	(CP9)
$\mathsf{callcc}\ M$	$\rightarrow^T$	$M\ \langle\delta, [\ ]_\delta\rangle$	(CP10)

Figure 4: Reduction system with continuation points: $\rightarrow_{cp}$

and we note $\rightarrow_{cp}^{\triangleright *}$ its transitive closure. $\mathsf{eval_{CP}}$ can be defined by

$$\mathsf{eval_{CP}}(M) = V \text{ iff } M \rightarrow_{cp}^{\triangleright *} V \text{ with } V \text{ a value}$$

The observational equivalence for CP is also a simple adaptation of definition 3.1

Definition 5.1 (Observational Equivalence) *$M \cong_{cp} N$ iff $\forall$ context $C[\ ] \in \Lambda_{cp}$, such that $C[M]$ and $C[N]$ are programs, either both $\mathsf{eval_{CP}}(C[M])$ and $\mathsf{eval_{CP}}(C[N])$ are defined or both are undefined.*

We can easily show that the system C and the system CP are equivalent. For this purpose, we define two translations which essentially map $\mathcal{A}$-applications to continuation points and vice-versa.

$\forall M \in \Lambda_{cp}, \overline{M}^{\varphi} \in \Lambda_c$:

$\overline{\delta_{cp}(a,b)}^{\varphi}$	$=$	$\delta_c(a,b)$		$\overline{\mathsf{callcc}\ M}^{\varphi}$	$=$	$(\mathsf{callcc}\ \overline{M}^{\varphi})$	
$\overline{x}^{\varphi}$	$=$	x		$\overline{\langle\alpha, K[\ ]_\alpha\rangle}^{\varphi}$	$=$	$\lambda x.\mathcal{A}\overline{K[\ ]_\alpha}^{\varphi[x/\alpha]}$	
$\overline{\lambda x.M}^{\varphi}$	$=$	$(\lambda x.\overline{M}^{\varphi})$		$\overline{[\ ]_\alpha}^{\varphi}$	$=$	$\varphi(\alpha)$	
$\overline{MN}^{\varphi}$	$=$	$(\overline{M}^{\varphi}\ \overline{N}^{\varphi})$					

$\forall P \in \Lambda_c, \underline{P} \in \Lambda_{cp}$:

$\underline{\delta_c(a,b)}$	$=$	$\delta_{cp}(a,b)$		$\underline{\mathsf{callcc}M}$	$=$	$(\mathsf{callcc}\ \underline{M})$	
$\underline{x}$	$=$	x		$\underline{\mathcal{A}M}$	$=$	$(\langle\delta, [\ ]_\delta\rangle\underline{M})$	
$\underline{\lambda x.M}$	$=$	$\lambda x.\underline{M}$		$\underline{(MN)}$	$=$	$(\underline{M}\ \underline{N})$	

These translations satisfy the following properties:

Theorem 5.2 $\forall P \in \Lambda_c,\ \underline{\overline{P}^{\varphi}} \cong_c P$
$\forall M \in \Lambda_{cp}, \overline{\underline{M}^{\varphi}} \cong_{cp} M$

Sketch of Proof of Theorem 5.2
We proceed by a straightforward induction on the size of P or M. □

The main result of this section is that the observational equivalences in C and CP are preserved.

Theorem 5.3 $M \cong_{cp} N \Leftrightarrow \overline{M}^{\varphi} \cong_c \overline{N}^{\varphi}$
$P \cong_c Q \Leftrightarrow \underline{P} \cong_{cp} \underline{Q}$

Sketch of Proof of Theorem 5.3
First, we prove that for each possible reduction $M \rightarrow_{cp} N$, $\overline{M}^{\varphi} \cong_c \overline{N}^{\varphi}$. We can then prove that $\mathsf{eval_{CP}}(M)$ is defined iff $\mathsf{eval_C}(\overline{M}^{\varphi})$ is defined. Then, supposing that $M \cong_{cp} N$, we try to prove that $\overline{M}^{\varphi} \cong_c \overline{N}^{\varphi}$, i.e. $\forall C[\ ]$, $\mathsf{eval_C}(C[\overline{M}^{\varphi}])$ and $\mathsf{eval_C}(C[\overline{N}^{\varphi}])$ are simultaneously both defined or undefined. We show that for any $C[\ ] \in \Lambda_c$, we can find $K[\ ] \in \Lambda_{cp}$ such that $K[\ ] \equiv \underline{C[\ ]}$. Therefore, $\mathsf{eval_C}(C[\overline{M}^{\varphi}])$ is defined, iff $\mathsf{eval_{CP}}(K[M])$ is defined, iff $\mathsf{eval_{CP}}(K[N])$ is defined, iff $\mathsf{eval_C}(C[\overline{N}^{\varphi}])$ is defined which concludes the proof. □
This theorem has a corollary. There is a bijection between the classes of observational equivalent programs in C and CP.

Corollary 5.4 *If we consider programs composed of variables, constants, applications, lambda abstraction,* callcc *applications,* $\cong_c\ =\ \cong_{cp}$

The optimisation rules given in figure 3 can now be adapted to CP where they also preserve observational equivalence $\cong_{cp}$.

We are now ready to define a reduction system where the notion of extent is made explicit.

6 The CPP calculus

The notion of extent is not easy to define for a parallel language with first-class continuations. First, we informally define it and then we represent it explicitly in a new reduction system CPP.

According to [13], [28], the *extent* refers to a period of time: the lifetime of an object or a construct.

"In Scheme, the extent of the application of a function f on its argument v is the time during which is computed the body of the function f, this includes the time taken by the computation of all subforms that appear in the body[24, page 175, section 1]".

The extent of an expression in which parallelism is introduced encompasses all the processes evaluating parts of this expression [23].

On a sequential machine, when first-class continuations are not used, the extent of an expression is a single interval of time, or a *contiguous* time period. When first-class continuations with indefinite extent are introduced as in Scheme, the evaluation of an expression E can be aborted by applying a continuation, and the evaluation of E can be resumed later by a continuation which was captured in E. In this situation, the extent is composed of several intervals of time, or a *non-contiguous* time period. On a parallel machine, the evaluation order is non-deterministic (while results of our programs are deterministic). Therefore, time intervals can be interleaved.

We define the extent of an expression callcc M by the extent of the application of M to the current continuation. In the reduction system CP, callcc is bubbled up to the top level in order to build the continuation. Let us suppose that rule CP4 is used,

$$(\mathsf{callcc}\ M)N \quad \rightarrow \quad \mathsf{callcc}\ \lambda k.(M\langle \alpha, k([\,]_\alpha\ N)\rangle)N \qquad \text{(CP4)}$$

The extent of callcc M in the left-hand side is the extent of the application of M to the continuation $\langle \alpha, k([\,]_\alpha\ N)\rangle$, while the extent of the callcc-expression in the right-hand side is the extent of the application of $\lambda k.(M\langle \alpha, k([\,]_\alpha\ N)\rangle)N$ to the current continuation. Consequently, the extent of the callcc in the right-hand side includes the lifetime of N which is not the case in the left-hand side. Eventually, when we apply a top level reduction CP10, the extent of callcc is the lifetime of the whole program.

This example illustrates that it is difficult to define the extent of a callcc in a reduction system like CP. This is the reason why we introduce a new construct, that we call prompt, which is used to mark the extent of a callcc. We define a new set of expressions, Λ_{cpp}:

$$\text{Term } M ::= \begin{cases} V & \text{Value} \\ (MM) & \text{Application} \\ (\mathsf{callcc}\ M) & \text{callcc-Application} \\ \#_\alpha(M) & \text{Prompt} \end{cases}$$

$$\text{Value } V ::= \begin{cases} a, b, \ldots & \text{Constants} \\ x, y, \ldots & \text{Variables} \\ \lambda x.M & \text{Lambda Abstraction} \\ \langle p, K[\,]_p\rangle & \text{Continuation point} \end{cases}$$

where $K[\,]$ is a captured context:

$$K \quad ::= \quad [\,] \mid K[[\,]M] \mid K[V[\,]] \mid K[\mathsf{callcc}\lambda k.[\,]] \mid K[\#_\alpha([\,])] \mid K[(\lambda v.[\,])V]$$

Similarly, we define contexts $C[\,]$ and evaluation contexts $E[\,]$:

$$C[\,] ::= [\,] \mid C[M[\,]] \mid C[[\,]M] \mid C[\lambda x.[\,]] \mid C[\mathsf{callcc}\ [\,]] \mid C[\#_\alpha([\,])]$$
$$E[\,] ::= [\,] \mid E[V[\,]] \mid E[[\,]M] \mid E[\#_\alpha([\,])]$$

The reduction system is based on equations displayed in figure 5. Equations CPP1 to CPP10 are the same as equations CP1 to CP10. Those equations are independent of the prompt construct. CPP15 is the rule which introduces a prompt; it is similar to the equation OPT6:

$$\mathsf{callcc}\ M \quad \cong_c \quad \mathsf{callcc}(\lambda k.(M\langle \alpha, k[\,]_\alpha\rangle)) \qquad \text{(OPT6)}$$

$$\mathsf{callcc_u}\ M \quad \rightarrow \quad \mathsf{callcc}\lambda k.\#_\alpha(M\ \langle \alpha, k[\,]_\alpha\rangle) \quad \text{with a fresh } \alpha \qquad \text{(CPP15)}$$

The equation CPP15 wraps the application $(M\langle \alpha, k[\,]_\alpha\rangle)$ in a prompt $\#_\alpha(\ldots)$ where the continuation point $\langle \alpha, k[\,]_\alpha\rangle)$ and the prompt $\#_\alpha(\ldots)$ are given the same new fresh name.

When a callcc is a redex for the first time, rule CPP15 should be applied to mark the extent of this callcc. Afterwards, there is no need to use this rule again: indeed, one prompt is enough, moreover if we re-apply this rule, we obtain:

$$\begin{aligned} &\mathsf{callcc}\ M \\ &\quad \rightarrow \quad \mathsf{callcc}\lambda k.\#_\alpha(M\ \langle \alpha, k[\,]_\alpha\rangle) \\ &\quad \rightarrow \quad \mathsf{callcc}\lambda k''.\#_\beta((\lambda k.\#_\alpha(M\ \langle \alpha, k[\,]_\alpha\rangle))\ \langle \beta, k[\,]_\beta\rangle) \\ &\quad \rightarrow \quad \mathsf{callcc}\lambda k''.\#_\beta(\#_\alpha(M\ \langle \alpha, \langle \beta, k[\,]_\beta\rangle\ [\,]_\alpha\rangle)\ ) \\ &\quad \rightarrow \quad \mathsf{callcc}\lambda k''.\#_\beta(\#_\alpha(M\ \langle \alpha, k\ [\,]_\alpha\rangle)\ ) \end{aligned}$$

where the continuation point named β has disappeared and the prompt β is useless. This is the reason why we have added the subscript u to callcc in the left-hand side of the rule. This $\mathsf{callcc_u}$ is a construct which originally appears in the user program while the callcc in the right-hand side is an internal callcc generated by the reduction system.

In order to be able to perform the same reductions as we could before introducing the prompt, we must consider the different reductions of $(M\langle \alpha, k[\,]_\alpha\rangle)$ that might appear in the prompt:

1. $(M\langle \alpha, k[\,]_\alpha\rangle)$ reduces to a value V. We can use equation CPP11

   $$\#_\alpha(V) \quad \rightarrow \quad V \text{ with } V \text{ a value} \qquad \text{(CPP11)}$$

   which eliminates the mark $\#_\alpha()$, meaning that the application $(M\ \langle \alpha, k[\,]_\alpha\rangle)$ has reached its end.

2. A continuation named β is invoked and escapes from $(M\langle \alpha, k[\,]_\alpha\rangle)$. By equation CPP13

   $$\#_\alpha(\langle \beta, K[\,]_\beta\rangle\ V) \quad \rightarrow \quad \langle \beta, K[\,]_\beta\rangle\ V \text{ with } V \text{ a value} \qquad \text{(CPP13)}$$

   the mark $\#_\alpha()$ is removed to allow the escape of this continuation.

$(\lambda x.M)V$	$\rightarrow$	$M\{V/x\}$ with V a value	(CPP1)
(ab)	$\rightarrow$	$\delta_{cpp}(a,b)$ if this is defined	(CPP2)
$M(\mathsf{callcc}\ N)$	$\rightarrow$	$\mathsf{callcc}\ \lambda k.(\lambda f.f\ (N\ \langle p, k(f\ [\,]_p)\rangle))))M$	(CPP3)
$(\mathsf{callcc}\ M)N$	$\rightarrow$	$\mathsf{callcc}\ \lambda k.(M\ \langle p, k([\,]_p\ N)\rangle)N$	(CPP4)
$(\langle p, K[\,]_p\rangle\ V)N$	$\rightarrow$	$(\langle p, K[\,]_p\rangle\ V)$ with V a value	(CPP5)
$M(\langle p, K[\,]_p\rangle\ V)$	$\rightarrow$	$(\langle p, K[\,]_p\rangle\ V)$ with M, V values	(CPP6)
$\langle\alpha, (\mathsf{callcc}\ M)\rangle$	$\rightarrow$	$\langle\alpha, (M\ \langle\delta, [\,]_\delta\rangle)\rangle$	(CPP7)
$\langle\alpha, \langle\beta, K_1[\,]_\beta\rangle\ (K_2[\,]_\alpha)\rangle$	$\rightarrow$	$\langle\alpha, K_1[K_2[\,]_\alpha]\rangle$	(CPP8)
$(\langle p, K[\,]_p\rangle\ V)$	$\rightarrow^T$	$K[V]$ with V a value	(CPP9)
$\mathsf{callcc}\ M$	$\rightarrow^T$	$M\ \langle\delta, [\,]_\delta\rangle$	(CPP10)
$\#_\alpha(V)$	$\rightarrow$	V with V a value	(CPP11)
$\#_\alpha(\langle\alpha, K[\,]_\alpha\rangle\ V)$	$\rightarrow$	V with V a value	(CPP12)
$\#_\beta(\langle\alpha, K[\,]_\alpha\rangle\ V)$	$\rightarrow$	$\langle\alpha, K[\,]_\alpha\rangle\ V$ with V a value	(CPP13)
$\mathsf{callcc}\ \lambda k.M$	$\rightarrow$	M with $k \notin FV(M)$	(CPP14)
$\mathsf{callcc}_{\mathsf{u}}\ M$	$\rightarrow$	$\mathsf{callcc}\lambda k.\#_\alpha(M\ \langle\alpha, k[\,]_\alpha\rangle)$ with a fresh α	(CPP15)
$\#_\alpha(\mathsf{callcc}\ M)$	$\rightarrow$	$\mathsf{callcc}\ \lambda k.\#_\alpha(M\langle p, k(\#_\alpha([\,]_p))\rangle)$	(CPP16)

Figure 5: Reduction system with continuation points and prompts: $\rightarrow_{cpp}$

3. $(M\langle\alpha, k[\,]_\alpha\rangle)$ reduces to a callcc expression.

$$\#_\alpha(\mathsf{callcc}\ M) \rightarrow \mathsf{callcc}\ \lambda k.\#_\alpha(M\langle p, k(\#_\alpha([\,]_p))\rangle) \quad \text{(CPP16)}$$

In the equation CPP16, the callcc is passed out of the mark and the mark is also copied in the continuation.

Now, we can define the extent of a callcc by the extent of its corresponding mark. Moreover, we say that an expression M is evaluated in the extent of a callcc if M is a redex in an applicative context appearing in the mark associated with this callcc:

$$\#_\alpha(A[M]) \rightarrow \#_\alpha(A[M']) \quad \text{with } A[\,] \text{ an applicative context}$$

We can easily optimise the invocation of a continuation in the extent of the callcc by which it was reified. Since a continuation and the mark delimiting the extent of this callcc are given a unique name, the following rule can detect such an invocation:

$$\#_\alpha(\langle\alpha, K[\,]_\alpha\rangle\ V) \rightarrow V \text{ with } V \text{ a value} \quad \text{(CPP12)}$$

Therefore, rule CPP13 should be applied when a continuation escapes from a mark with a different name while rule CPP12 is used for the application of a continuation in a mark with the same name.

The main result of this section is the following theorem; it says that two programs are observationally equivalent in CP if they are observationally equivalent in CPP.

Theorem 6.1 $\forall M \in \Lambda_{cp}, M \cong_{cp} N$ *iff* $M \cong_{cpp} N$

The proof of theorem 6.1 requires the definition of an intermediate reduction system CP′: it is based on the set of expressions Λ_{cp} and it is composed of all reduction rules of CPP except CPP12. In this system CP′, we define a notion of reduction, its compatible closure, equivalence relations, evaluation and observational equivalence as we did in the previous systems. We can prove that CP′ and CP satisfy the following theorem:

Theorem 6.2 $\mathsf{eval_{CP}}(M) = V \iff \mathsf{eval_{CP'}}(M) = V'$ *with* $\mathcal{S}(V') = V$ *and* $\mathcal{S}(M)$ *defined by*

$$\begin{array}{rclcrcl} \mathcal{S}(\lambda x.M) & = & \lambda x.\mathcal{S}(M) & & \mathcal{S}(x) & = & x \\ \mathcal{S}(MN) & = & (\mathcal{S}(M)\ \mathcal{S}(N)) & & \mathcal{S}([\,]_\alpha) & = & [\,]_\alpha \\ \mathcal{S}(\mathsf{callcc}\ M) & = & \mathsf{callcc}\ \mathcal{S}(M) & & \mathcal{S}(\#_\alpha(M)) & = & \mathcal{S}(M) \\ \mathcal{S}(\langle\alpha, M\rangle) & = & \langle\alpha, \mathcal{S}(M)\rangle & & & & \end{array}$$

Although rule CPP12 does not belong to CP′, we can show that this rule preserves observational equivalence $\cong_{cp'}$.

Lemma 6.3 $\#_\alpha(\langle\alpha, K[\,]_\alpha\rangle\ V) \cong_{cp'} V$

Sketch of Proof of Lemma 6.3
In C, we have proved that $\mathsf{callcc}\ \lambda k.(kV) \cong_c \mathsf{callcc}\ \lambda k.V$. So, it is true in CP by th. 5.4 and in CP′ by th. 6.2. By applying rule CPP15, we have $\mathsf{callcc}\ \lambda k'.\#_\alpha(\langle\alpha, k'\ [\,]_\alpha\rangle V') \cong_{cp'} \mathsf{callcc}\ \lambda k'.\#_\alpha(V')$ from which we can derive that $\#_\alpha(\langle\alpha, K[\,]_\alpha\rangle V') \cong_{cp'} V'$. □

Sketch of Proof of Theorem 6.1
Rule CPP12 is shown to preserve observational equivalence in CP′ and CPP is defined to be CP′ extended by CPP12. Therefore, by theorem 6.2, $M \cong_{cp} N$ iff $M \cong_{cp'} N$ and by lemma 6.3, $M \cong_{cp'} N$ iff $M \cong_{cpp} N$ which concludes the proof. □

CPP is a reduction system which allows the capture of continuations in any applicative context and which does not abort the whole computation when a continuation is applied in the extent of the callcc which created it. With theorems 5.4 and 6.1, observational equivalence is the same in CPP and C, meaning that programs can be evaluated with a sequential or a parallel strategy. Therefore, we have succeeded in formalising the intuitive semantics of section 1.1.

Instead of CPP16, we could have used another equation

$$\#_\alpha(\mathsf{callcc}\ M) \rightarrow \mathsf{callcc}\ \lambda k.\#_\alpha(M\langle p, k[\,]_p\rangle) \quad \text{(CPP16}')$$

where the prompt in not copied in the continuation. In this case, the prompt is the explicit representation of the *dynamic* extent of the callcc.

7 Related work

Queinnec [22] has also proposed a semantics for continuations in a parallel framework but his `pcall` construct is not transparent. Moreover, the concept of returned value has changed: an expression may return several results (at different times) and, for a given expression, the number of returned results can change with execution.

Katz and Weise have implemented a system with a transparent `future` construct [16], [17]. It is based on a notion of *legitimacy*: a process is legitimate if the code it is executing would have been executed by a sequential implementation in the absence of future. For a given program, their implementation returns one or more results without knowing if they are legitimate. The legitimacy is determined later when all subcomputations have completed and a total order of evaluation can be found as in the sequential semantics. In a sense, we also have a notion of legitimacy: we have to determine whether it is legal to apply a continuation in a parallel program. But, `pcall` can be more optimised than `future`: a continuation can be applied in an evaluation context and if an expression returns a value using parallel evaluation rules, the result is behaviourly equivalent to the one returned by a sequential evaluation strategy.

In [19], we presented the intuitive semantics we describe in section 1.1. We can see [19] as an implementation of the system CPP on a machine with multiple processors and a shared memory.

The system C is very similar to Felleisen's λ_c [8] except that C is based on callcc and not $\mathcal{C}$. The notion of observational equivalence has already been used by Felleisen and others in several papers [5], [8], [9]. However, Felleisen hardly investigated the problem of parallelism in reduction systems with control operators. This is partly due to the choice of the control operator: $\mathcal{C}$ is abortive, it applies its argument to the current continuation in an empty context while callcc is not abortive. Therefore, it was required for $\mathcal{C}$ to be in an evaluation context which is not the case for callcc. In [6], a parallel evaluation strategy is proposed but $\mathcal{C}$ is designated as the cause of bottlenecks.

Sabry and Felleisen [26], [27] present extensions of the λ_v and λ_c-calculi which are complete with respect to CPS translation ($\forall M, N \in \Lambda_{v,c}, \lambda_{v,c} \vdash M = N \Leftrightarrow \lambda_n \vdash [\![M]\!] = [\![N]\!]$). It appears that their axioms $\beta_{lift}, \beta_{flat}, \beta_{id}, \beta_\Omega, C_{lift}, C_{abort}, C_{tail}$ all satisfy our relation of observational equivalence. However, we did not investigate if C was complete by adding this set of axioms.

Kanneganti *et al.* [15] use the axiom $\mathsf{callcc}\lambda k.C[E[kv]] \rightarrow \mathsf{callcc}\ \lambda k.C[kv]$ which also preserves observational equivalence $\cong_c$. When added to C, this axiom optimises the application of a continuation in the *dynamic* extent of a callcc. While this approach does not require to introduce a prompt construct and continuation point objects, it suffers from the defaults suggested at the beginning of section 6. callcc is given the role of both marking the dynamic extent and capturing a continuation: therefore, after capturing a continuation, the mark of the dynamic extent has disappeared.

Optimised CPS translations were proposed by Sabry and Felleisen [26], [27], Danvy and Filinski [3]. For the purpose of the proof, we had to specialise our CPS translation but the applicability of this approach in other circumstances does not appear to be immediate.

The mechanism of prompt introduced in section 6 is essentially different from Felleisen's prompt [4], or Danvy and Filinski's `reset` [2] or Queinnec and Serpette's `splitter` facility [24]. In their approach, a prompt is used to delimit a partial continuation while we use the prompt to mark the extent of callcc.

Jouvelot and Gifford [14] present a static analysis of programs with `call/cc`. Their type system can detect programs that use internally `call/cc`. While they prove their type system gives a safe approximation, we show that the optimisation of the application of a continuation in the extent of its `call/cc`, *always* preserves observational equivalence.

8 Conclusion

To our knowledge, it is the first time that reduction rules for control operators are investigated in the perspective of parallelism. It appears from our results that continuations can be captured in any applicative context. It is also the first time that the notion of extent of a callcc is used in a reduction system in order to avoid to abort the whole computation.

Allowing the capture of a continuation in any applicative context entails that a control operator like callcc does not introduce sequentiality in a parallel language. However, application of a continuation introduces some sequentiality, especially when the continuation is applied outside of the extent of the callcc by which it was reified.

9 Acknowledgements

To Vincent Kieffer for providing helpful comments on formal proofs, to Olivier Danvy for reading an earlier version of this paper, to the anonymous referees for their comments, to Amr Sabry and Matthias Felleisen for an encouraging discussion in a Chinese coffee shop in SF.

A The optimised cps translation

As observed by Plotkin [21], by Danvy and Filinski [3] and by Sabry and Felleisen [26], the CPS translation introduces "administrative" redices. Indeed, for a standard reduction in the call-by-value λ-calculus, $M_0 \mapsto_v M_1$, we have a sequence of administrative reductions, followed by a reduction $M_0' \rightarrow M_1'$ which corresponds to the original reduction:

$$[\![M_0]\!](\lambda x.x) \rightarrow^* M_0' \rightarrow M_1'$$

but this term M_1' does not reduce to $[\![M_1]\!](\lambda x.x)$, it is in fact the reduction of administrative redices of $[\![M_1]\!](\lambda x.x)$:

$$[\![M_1]\!](\lambda x.x) \rightarrow^* M_1'$$

Therefore, we have this unfortunate property

$$M_0 \mapsto_s M_1 \quad \text{but} \quad [\![M_0]\!](\lambda x.x) \not\rightarrow^* [\![M_1]\!](\lambda x.x)$$

Hence, for the purpose of the proof of the first proposition of theorem 3.3, Plotkin introduces an optimised CPS translation where some administrative redices are eliminated. In our proof, we also define an optimised cps translation. We write $[\![M]\!]_o^K$ for the optimised cps translation of M with the continuation K. We have the following property

$$M_0 \mapsto_s M_1 \ \Rightarrow\ [\![M_0]\!]_o^K \rightarrow^* [\![M_1]\!]_o^K$$

which gives lemma A.1.

If $M \mapsto_s N$ using the standard reduction, the optimised cps-translation of M reduces to the optimised cps-translation of N.

Lemma A.1 *If $M \mapsto_s N$ then $[\![M]\!]_o^K \to^* [\![N]\!]_o^K$ (if K is a closed value and M and N are terms)*

Sketch of Proof of Lemma A.1

The proof similar to Plotkin's proof of lemma 6.3; we proceed by induction on the size of M and by cases according to the definition of $\mapsto_s$. □

Optimised cps translations were proposed by Plotkin [21], by Danvy and Filinski [3] and by Sabry and Felleisen [26], [27]. In [21] and [26], the translations concern call-by-value terms while, in [27] and [3] they concern call-by-value terms extended by control operators callcc or `escape` x `in` M (which is a special form equivalent to callcc$\lambda x.M$).

As Sabry, in the original Fischer's translation, we mark by an overline all lambda-abstractions which are not present in the original term. Rules Ocps1 to Ocps5 in definition A.2 are similar to Sabry's. In those rules, each continuation is a $\overline{\lambda}$-abstraction. We add rules Ocps6 and Ocps7 for the translation of $\mathcal{A}$ and callcc. In the translation of $\mathcal{A}M$, the initial continuation is marked as administrative and in the translation of callcc$M$, the reified continuation $\overline{\lambda}vk'.kv$, standing for $\overline{\lambda}v.\overline{\lambda}k'.kv$ is also marked as administrative.

Definition A.2 (Optimised CPS translation)

$$
\begin{array}{rcll}
[\![V]\!] & = & \overline{\lambda}k.k\Psi(V) \quad \textit{with } V \textit{ a value} & \text{(Ocps1)} \\
[\![\delta_c(a,b)]\!] & = & \delta_n(a,b) & \text{(Ocps2)} \\
[\![(MN)]\!] & = & \overline{\lambda}k.[\![M]\!](\overline{\lambda}m.[\![N]\!](\overline{\lambda}n.mnk)) & \text{(Ocps3)} \\
\Psi(x) & = & x \textit{ with } x, \textit{ variable or constant} & \text{(Ocps4)} \\
\Psi(\lambda x.M) & = & (\lambda x.[\![M]\!]) & \text{(Ocps5)} \\
[\![(\mathcal{A}M)]\!] & = & \overline{\lambda}k.[\![M]\!](\overline{\lambda}x.x) & \text{(Ocps6)} \\
[\![(\mathsf{callcc}\ M)]\!] & = & \overline{\lambda}k.[\![M]\!](\overline{\lambda}m.m(\overline{\lambda}vk'.kv)k) & \text{(Ocps7)}
\end{array}
$$

We define the optimised CPS translation as a three-pass process, where lambda-expressions marked as administrative are reduced at translation-time, and where unreduced administrative expressions are unmarked.

Definition A.3 (Three-Pass Translation) *The three-pass optimised translation of M, indexed by the continuation K, $[\![M]\!]_o^K$ is N iff $\mathcal{U}(\mathsf{eval}^-([\![M]\!]K)) = N$ where $\mathsf{eval}^-(P) = Q$ iff $\overline{\beta\eta} \vdash P \to^* Q$ with $\to_{\overline{\beta}}$ and $\to_{\overline{\eta}}$ reductions defined by*

$$(\overline{\lambda}x.M)V \quad \to_{\overline{\beta}} \quad M\{V/x\} \qquad (1)$$

$$(\overline{\lambda}x.Mx) \quad \to_{\overline{\eta}} \quad M \textit{ with } x \notin FV(M) \qquad (2)$$

and $\mathcal{U}(P)$ removes the marks on administrative abstractions:

$$
\begin{array}{rclrcl}
\mathcal{U}(\lambda x.M) & = & \lambda x.\mathcal{U}(M) & \mathcal{U}(PQ) & = & (\mathcal{U}(P)\mathcal{U}(Q)) \\
\mathcal{U}(\overline{\lambda}x.M) & = & \lambda x.\mathcal{U}(M) & \mathcal{U}(x) & = & x
\end{array}
$$

We also note $\Psi_o(V)$, the result of a translation of a value $[\![V]\!]_o^K = K\Psi_o(V)$. Therefore, $\Psi_o(V) = \mathcal{U}(\mathsf{eval}^-(\Psi(V)))$

This solution is not yet satisfactory; indeed, in C, let us consider (callcc $M)N$ and its reduction by rule C4. Let us cps-translate the two terms, we obtain the following diagram:

$$
\begin{array}{ccc}
(\mathsf{callcc}\ M)N & \overset{\mathsf{callcc}_L}{\to} & \mathsf{callcc}\ \lambda k.(M(\lambda f.\mathcal{A}(k(fN))))N \\
\downarrow_{cps} & & \downarrow_{cps} \\
[\![(\mathsf{callcc}\ M)N]\!]_o^K & \leftarrow^* & [\![\mathsf{callcc}\ \lambda k.(M(\lambda f.\mathcal{A}(k(fN))))N]\!]_o^K
\end{array}
$$

where the reductions in λ_n go in the opposite direction to the reduction in C. This situation comes from the fact that the rule C4 introduces in the right-hand term two λ-abstractions; the new argument of callcc is called a continuation receiver, $\lambda k.(M(\lambda f.\mathcal{A}(k(fN))))N$ and the other abstraction, $(\lambda f.\mathcal{A}(k(fN)))$, is a functional representation of the continuation. Since the translation of those λ-abstractions are not administrative, they are not reduced in the three-pass translation. As a matter of fact, both abstractions should be considered as administrative abstractions introduced by the reduction system. Indeed, it is the internal way of handling callcc in C. Therefore, in the reduction system, let us mark by a star the λ-expressions which are continuation receivers or representation of a continuation created by the system.

$$
\begin{array}{rcl}
M(\mathsf{callcc}\ N) & \to & \mathsf{callcc}\ \lambda^*k.M(N\ (\lambda^*v.\mathcal{A}(k(Mv)))) \\
(\mathsf{callcc}\ M)N & \to & \mathsf{callcc}\ \lambda^*k.(M(\lambda^*f.\mathcal{A}(k(fN))))N \\
\mathcal{A}(\mathsf{callcc}M) & \to & \mathcal{A}(M(\lambda^*x.\mathcal{A}x)) \\
(\lambda^*x.M)V & \to & M\{V/x\}
\end{array}
$$

and let us add the new following rules to the cps translation

$$\Psi(\lambda^*x.M) \quad = \quad (\overline{\lambda}x.[\![M]\!]) \qquad \text{(Ocps8)}$$

where λ^*-expressions are marked as administrative and are reduced at translation-time. With these two rules, we have now

$$
\begin{array}{ccc}
(\mathsf{callcc}\ M)N & \overset{\mathsf{callcc}_L}{\to} & \mathsf{callcc}\ \lambda^*k.(M(\lambda^*f.\mathcal{A}(k(fN))))N \\
\downarrow_{cps} & & \downarrow_{cps} \\
[\![(\mathsf{callcc}\ M)N]\!]_o^K & \equiv & [\![\mathsf{callcc}\ \lambda^*k.(M(\lambda^*f.\mathcal{A}(k(fN))))N]\!]_o^K
\end{array}
$$

Of course, the user is not given the right to write programs with λ^*-abstractions. Since they are only created by the reduction system, we can prove that this optimised cps translation is always defined.

Lemma A.4 *The optimised cps translation is a total function.*

Sketch of Proof of Lemma A.4

Sabry and Felleisen proved that this cps translation is a total function for terms belonging to Λ. Using the same technique, we can show that $[\![\mathcal{A}M]\!]$ does not duplicate any $\overline{\lambda}$-expression. However, the introduction of callcc and λ^*-abstraction increases the number of possible redices in the cps terms. We can show that $[\![\mathsf{callcc}M]\!]$ duplicates its continuation but there is a bound on the number of duplications of this continuation which depends on M and which can be computed. □

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.

[2] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, June 1990.

[3] Olivier Danvy and Andrzej Filinski. Representing Control. A Study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[4] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract Continuations : A Mathematical Semantics for Handling Full Functional Jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, July 1988.

[5] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).

[6] Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *Proc. Conf. on Parallel Architecture and Languages Europe*, pages 206–223. Lecture Notes 259 in Computer Science. Springer-Verlag, 1987.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. Reasoning with Continuations. In *Proceedings of the Symposium on Logic in Computer Science*, pages 131–141, Washington DC, June 1986. IEEE Computer Society Press.

[8] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theoretical Computer Science (North-Holland)*, (52):205–237, 1987.

[9] Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992.

[10] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109. SIGPLAN Notices 7(1), 1972.

[11] Robert H. Halstead, Jr. New ideas in parallel lisp : Language design, implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, pages 2–57. Lecture Notes 441 in Computer Science. Springer-Verlag, 1990.

[12] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[13] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[14] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 24, pages 218–226, Portland, Oregon, June 1989. ACM Press.

[15] R. Kanneganti, R. Cartwright, and M. Felleisen. SPCF: its Model, Calculus and Computational Power. In *Proceedings of the REX Workshop on Semantics and Concurrency*. Lecture Notes in Computer Science. Springer Verlag, 1992.

[16] Morry Katz and Daniel Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, June 1990.

[17] James S. Miller and B. S. Epstein. Garbage collection in MultiScheme. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, pages 138–160. Lecture Notes 441 in Computer Science. Springer-Verlag, 1990.

[18] Luc Moreau. A Parallel Functional Language with First-Class Continuations. Programming Style and Semantics. To appear in, *Computers and Artificial Intelligence*. Journal version of [19] and [20].

[19] 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)*, pages 415–430, Paris, June 1992. Lecture Notes in Computer Science 605. Springer-Verlag.

[20] Luc Moreau. Programmer dans un langage fonctionel parallèle avec continuations. In *Avancées Applicatives. Journées Francophones des Langages Applicatifs*, Tréguier, France, February 1992. BIGRE.

[21] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[22] Christian Queinnec. Polyscheme, a Semantics for a Concurrent Scheme. In *High Performance and Parallel Computing in Lisp Workshop*, Twickenham, England, November 1990. Europal.

[23] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In A. Agarwal, R. H. Halstead, and Takayasu Ito, editors, *Proceedings of the Workshop on Parallel Symbolic Computing: Languages, Systems and Applications*, Boston, Massachussetts, October 1992.

[24] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1991.

[25] John Reynolds. Definitional interpreters for higher-order programming languages. In *25th ACM National Conference*, pages 717–740, 1972.

[26] Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 288–298, June 1992.

[27] Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. To appear in *Lisp and Symbolic and Computation, Special Issue on Continuations*, 1993.

[28] Guy Lewis Steele, Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.