An operational semantics for a parallel functional language with continuations

Luc Moreau
Service d'Informatique
Institut d'Electricité Montefiore, B28
University of Liège (Sart-Tilman)
4000 Liège
Belgium
Phone: +32-41-56.26.42
Fax: +32-41-56.29.84
moreau@montefiore.ulg.ac.be

November 13, 1991

Abstract

Explicit parallelism can be introduced in Scheme by adding the constructs `fork`, `pcall` and `future`. Katz and Weise gave an implementation where those constructs are transparent even when first class continuations are used. In this paper, we formalise this work by giving an operational semantics for a functional language with first class continuations and transparent constructs for parallelism. We introduce a concept of higher order continuation that we call metacontinuation which preserves sequential properties of continuations in a parallel language.

Keywords: Scheme, parallelism, transparency, continuation, metacontinuation, left expression, operational semantics.

1 Introduction

There are essentially two trends to add parallel constructs to a functional language. On the one hand, the approach adopted by the ML community [17] consists in adding to the language the notions of processes, channels and communications as in calculi like CCS [16]. An operational semantics is given in [15] and several implementations were realised (PFL [11], CML [24]). Its main drawback is that the language is no longer functional and that it requires another programming methodology to develop parallel applications. On the other hand, one can preserve the functional features of the language by adding constructs like `future` and `pcall`. These constructs were initially implemented in MultiLisp as described in [3], [4]. Such operators are said to be *transparent* since programs using them return the same results as sequential versions of these programs where those operators were deleted.

The second approach, with transparent constructs, is used to add parallelism to Scheme [23]. One feature of Scheme is the presence of first class continuations allowing the definition of powerful control structures; several programming examples with continuations can be found in [9], [8]. However, first class continuations gave a hard time to researchers to define a transparent `future` construct ([13], [10], [12], [14], [5]). An implementation of such a transparent `future` construct was realised by Katz and Weise ([12], [14]) but, unlike the ML approach, a formal semantics is not provided. In [20], Queinnec gives a denotational semantics for a parallel Scheme called PolyScheme. He introduces "symmetric continuations" to allow a maximal amount of parallelism but PolyScheme is not transparent: a parallel program may return multiple results, some of them different from the sequential semantics.

In this paper we present an operational semantics for a subset of Scheme extended with transparent constructs for parallelism `fork`, `pcall` and `future`. We call this language Λ_C. This operational semantics is a translation of Λ_C to a language, called $\Lambda_{//}$, itself specified by an operational semantics. The operational semantics can be seen as a description of a parallel evaluation on a high level parallel abstract machine. This translation uses "symmetric continuations" as in PolyScheme. In order to guarantee results identical to sequential results even for parallel programs using continuations, we introduce higher order continuations that we call *metacontinuations*. They ensure that when a captured continuation has to be applied in a parallel expression, all subexpressions which should be evaluated in the sequential version of this expression are indeed evaluated. The target language of the translation, $\Lambda_{//}$, is a functional language without continuations to which CCS-style constructs for parallelism were added. In this paper, we only comment on some essential transition rules of the operational semantics of $\Lambda_{//}$ but a complete operational semantics is given in [15] for a parallel ML with similar constructs for parallelism.

This paper is organised as follows. First, the two languages Λ_C and $\Lambda_{//}$ are presented. The second part is an attempt to define Λ_C using the technique of symmetric continuations. The nontransparency of the `pcall` operator is highlighted by an example. In the third part, the notion of metacontinuation is introduced to restore sequential properties of continuations and is used in the translation of Λ_C. After an example of a parallel program in Λ_C, we conclude this paper by related and future work.

2 The source and target languages: Λ_C and $\Lambda_{//}$

Λ_C, the source language of our translation, is a functional subset of the Scheme language. Its syntax is defined by

$$\text{If } \texttt{M} \text{ and } \texttt{N} \in \Lambda_C, \text{ then } \begin{cases} \texttt{(M N)} \\ \texttt{(lambda (x) M)} \\ \texttt{x} \\ \texttt{(call/cc M)} \end{cases} \in \Lambda_C.$$

The evaluation is sequential (left to right order) unless parallelism is explicitly introduced by three constructs.

A process p_1 evaluating `(fork exp)` in a sequence creates a process p_2 to evaluate `exp`, the value of `fork` is unspecified and process p_1 continues evaluating the sequence in parallel with p_2.

A process p_1 evaluating a `(pcall M N)` creates a process p_2 to evaluate `M` and a process p_3 to evaluate `N`. When both values are computed, the application is performed by p_2 or p_3, the other one and p_1 are killed.

A process p_1 evaluating `(future exp)` creates a process p_2 to evaluate `exp`, the returned value is an object called a placeholder. A placeholder is a data structure with a slot for *one* value aimed at containing the value of the expression `exp` when it is computed by process p_2. Moreover, there is a distinction between strict and non strict functions. Strict functions require the value contained in a placeholder when it is passed in argument and if it is not yet computed, the process is suspended. Non strict functions do not require this value. There is a strict primitive function `touch` which can be used to obtain the value of a placeholder. In this paper, we do not tackle the strict/non-strict distinction, we will suppose that all functions (except `touch`) are non strict and that the user has to explicitly touch the arguments. One can compare `future` and `touch` with `delay` and `force` except that `future` argument is evaluated eagerly. `future` was initially introduced in MultiLisp [4]; when combined with continuations, it can cause problems which were exposed in [5], [12].

$\Lambda_{//}$ is the target language of the translation and it is based on the same sequential subset as Λ_C but is extended with a set of *low level* primitives for concurrency. The four concurrency primitives are

`(fork thunk)` The function `fork` takes a thunk (function without argument) in argument and creates a new process applying this thunk. `fork` returns an unspecified value after the process creation.

`(channel)` Processes exchange data on channels. The function `channel` returns a *new* object called *channel identifier* on which processes can communicate.

`(send channel value)` Communications are synchronous as in CCS. For a communication, there must be a process sending a value on a channel and a process waiting for a value on the same channel. The value of the function `send` is unspecified.

`(receive channel)` The value of the function `receive` is the value transmitted on `channel` by the sending process during a synchronous communication.

For readability purpose, we add the usual syntactic sugar `let`, `begin` and `letrec` in both languages. We illustrate the programming style offered by $\Lambda_{//}$ in figure 1 by a function creating a process modelling a store. A store is a data structure created by the constructor `make-store`, accessed by `read` and modified by `write`.

An operational semantics of parallel ML is given in [15] by a set of transition rules similar to those of CCS ([16]). We illustrate in figure 2 some of them which are suitable for $\Lambda_{//}$. Those rules concern

```
(define (make-store c init-value)
  (fork (lambda ()
          (letrec ((loop (lambda (v)
                           (begin (send c v)
                                  (loop (receive c))))))
            (loop init-value)))))

(define (read c)
  (let ((value (receive c)))
    (begin (send c value)
           value)))

(define (write c v)
  (begin (receive c)
         (send c v)))
```

Figure 1: Definition of a store in $\Lambda_{//}$

transitions between *configurations* ($\Sigma|P$). Σ is composed of a set of channels (K) and a set of process identifiers (I); P is a set of processes $[p_i : e_i]$ where each process p_i is evaluating expression e_i. An expression such as

$$\Sigma \mid P[p_n : e'] \rightarrow \Sigma' \mid P'[p_n : e'']$$

describes a transition from a configuration $\Sigma \mid P$ where process p_n is evaluating expression e' to another configuration $\Sigma' \mid P'$ where the same process p_n is evaluating expression e''. In figure 2 we have inference rules like

$$\frac{exp_1}{exp_2}$$

meaning "from exp_1, infer exp_2".

In figure 2, rules 1 to 4 specify the evaluation of a sequential expression in process p_n for a given configuration Σ: this is a classical left to right evaluation order. Rule 5 is the evaluation rule of the function `channel`: it adds a new channel k to the set of channels K. Rule 6 is the evaluation rule of the function `fork`: it adds a new identifier q to the set of process identifiers I and creates a new process p_q to evaluate the argument of `fork`. A communication between two processes proceeds according to rule 6 if a process p_n wishes to send a value v on a channel k and a process p_m is ready to receive a value on the same channel.

$$\frac{\Sigma \mid P[p_n : e'] \rightarrow \Sigma' \mid P'[p_n : e'']}{\Sigma \mid P[p_n : (e'e)] \rightarrow \Sigma' \mid P'[p_n : (e''e)]} \quad (1)$$

$$\frac{\Sigma \mid P[p_n : e'] \rightarrow \Sigma' \mid P'[p_n : e'']}{\Sigma \mid P[p_n : (\langle x, e\rangle e')] \rightarrow \Sigma' \mid P'[p_n : (\langle x, e\rangle e'')]} \quad (2)$$

$$\Sigma \mid P[p_n : (\langle x, e\rangle v)] \xrightarrow{\beta} \Sigma \mid P[p_n : e\{v/x\}] \quad (3)$$

$$\Sigma \mid P[p_n : (\texttt{lambda (x) M})] \xrightarrow{\lambda} \Sigma \mid P[p_n : \langle x, M\rangle] \quad (4)$$

$$\frac{k \notin K}{\langle K, I\rangle \mid P[p_n : (\texttt{channel})] \xrightarrow{chn} \langle K \cup \{k\}, I\rangle \mid P[p_n : k]} \quad (5)$$

$$\frac{q \notin I}{\langle K, I\rangle \mid P[p_n : (\texttt{fork}\langle((), e\rangle)] \xrightarrow{frk} \langle K, I \cup \{q\}\rangle \mid P[p_n : ()][p_q : e]} \quad (6)$$

$$\frac{k \in K}{\Sigma \mid P[p_n : (\texttt{send } k\ v)][p_m : (\texttt{receive } k)] \xrightarrow{com} \Sigma \mid P[p_n : ()][p_m : v]} \quad (7)$$

Figure 2: Reduction rules for $\Lambda_{//}$

3 Symmetric or asymmetric continuation passing style

In figure 3, we give a translation for the sequential subset of Λ_C using the *continuation passing style* or *CPS* for short; such a style is often used for denotational semantics ([2]) and for program transformations in compilers ([25], [1]). In our notation, a translation consists of a set of translation rules having the following pattern: $[\![$`Term`$]\!]$=`exp`. The left hand side of the rule is a term of Λ_C in brackets and the right hand side is an expression in $\Lambda_{//}$. Such a rule should be read as "the text of the translation of `Term` is `exp`, in which every occurrence of $[\![e]\!]$ must be replaced by the text of the translation of e and each newly introduced variable in `exp` is supposed not to collide with existing ones".

```
[[x]]             = (lambda (κ) (κ x))
[[(lambda (x) M)]]= (lambda (κ) (κ (lambda (x c) ([[M]] c))))
[[(call/cc M)]]   = (lambda (κ) ([[M]] (lambda (vm) (vm (lambda (v κ') (κ v)) κ))))
[[(M N)]]         = (lambda (κ) ([[M]] (lambda (vm) ([[N]] (lambda (vn) (vm vn κ))))))
```

Figure 3: Continuation passing style translation

In order to completely specify Λ_C, we still have to add translation rules for the parallel constructs `pcall`, `fork` and `future`. Let us initially consider the first one. In [20], Queinnec gives a semantics for PolyScheme, a parallel dialect of Scheme. Let us use the same technique to define the `pcall` operator for which a verbose translation can be found in figure 4. For each application `(pcall M N)`, two processes are created to evaluate `M` and `N` in parallel and two new memory cells intended to contain the values of `M` and `N` are allocated. This translation is also a continuation passing translation: the continuation for `M` stores the value of `M` in the data structure `cm`; if `N` is already computed, the continuation applies the value of `M` to the value of `N` with the continuation κ otherwise the process dies.

```
[[(pcall M N)]] = (lambda (κ)
                  (let ((cn  a new memory cell) (cm  a new memory cell))
                    (begin (fork (lambda () ([[M]] (lambda (vm)
                                                     (begin  store vm in cm
                                                             (if  (computed? <value of N>)
                                                                  (vm <value of N> κ)
                                                                  (die)))))))
                           (fork (lambda () ([[N]] (lambda (vn)
                                                     (begin  store vn in cn
                                                             (if  (computed? <value of M>)
                                                                  (<value of M> vn κ)
                                                                  (die))))))))))
```

Figure 4: Verbose translation for a PolyScheme-style `pcall`

In figure 5, we give the translation rule for the parallel application. There are a few differences: (1) We must be sure that processes evaluating `M` and `N` do not both evaluate `(vm vn` κ`)`. Hence, the memory cells `cm` and `cn` must be considered as a critical section which is implemented by a semaphore `sem`. In order to enter the critical section, a value should be received on the channel representing the semaphore, and to exit the critical section a value should be sent. (2) The action `die` and the test `computed?` are implicit: the data structures `cm` and `cn` are initialised with an empty body function and

they are modified by functions applying vm to vn and κ. When the functions contained in those data structures are applied, if the body is empty, there is no more code to evaluate and the process dies else (vm vn κ) is evaluated. (3) The data structures cm and cn and the semaphore sem are stores for which a code is illustrated in figure 1.

```
⟦(pcall M N)⟧   = (lambda (κ)
                    (let ((cn (channel)) (cm (channel)) (sem (channel)))
                      (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                                    (begin (receive sem)
                                                           (write cm (lambda (vn x) (vm vn x)))
                                                           (let ((fn (read cn)))
                                                             (begin (send sem 'any)
                                                                    (fn vm κ))))))))
                             (fork (lambda () (⟦N⟧ (lambda (vn)
                                                    (begin (receive sem)
                                                           (write cn (lambda (vm x) (vm vn x)))
                                                           (let ((fm (read cm)))
                                                             (begin (send sem 'any)
                                                                    (fm vn κ))))))))
                             (make-store cm (lambda(vn κ) '()))
                             (make-store cn (lambda(vm κ) '()))
                             (make-store sem 'any))))
```

Figure 5: Translation for a PolyScheme-style pcall

In the continuation passing style translation (figure 3), the continuation of M, (lambda (vm) (⟦N⟧ (lambda (vn) (vm vn κ)))) and the continuation of N, (lambda (vn) (vm vn κ)) are *asymmetric* since they force M evaluation before N evaluation. They define a *left to right total order* of evaluation of expressions. In the pcall definition (figure 5), the continuation of M stores a value in store cm, reads store cn and applies this value to vm and the continuation κ. We see that the continuation of N is symmetric to the continuation of M. They define a *partial order* of evaluation of expressions: the body of a function is always evaluated after the subexpressions of a parallel application but there is no order between those subexpressions.

In this paper we use the term *symmetric continuations* to denote PolyScheme style continuations and we use *asymmetric continuations* to denote continuations such as those from the CPS translation and, by extension, we use the terms symmetric and asymmetric continuation passing styles (SCPS or ACPS).

While the ACPS total order of evaluation forbids parallelism, the partial order defined by the symmetric continuation passing style allows parallel evaluation of subexpressions in an application. Unfortunately, with such a meaning of pcall, an expression of Λ_C does not always return the same value as the same expression where the pcall operator is deleted: in other words, the pcall operator is not transparent.

Let us examine the evaluation of a simple program using the translations from figures 3 and 5:

```
(pcall f1 (call/cc (lambda (k)
                     (pcall (pcall f2 (k 1))
                            (k 2)))))
```
(8)

The resulting computation tree appears in figure 6 in which each node is associated to an expression and a process evaluating it. Node C is the capture point: i.e. it is the node where `call/cc` binds `k` to the current continuation. Nodes A_1 and A_2 are application points of this continuation. Node T is the top of the computation tree.

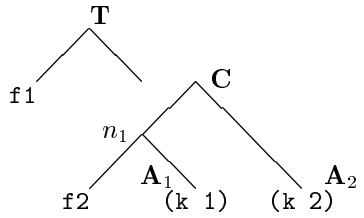

Figure 6: Computation tree of `(pcall f1 (call/cc (lambda (k) (pcall (pcall f2 (k 1)) (k 2)))))`

In expression (8), `k` is applied to `1` *and* `k` is applied to `2`. Indeed, in the symmetric continuation passing style, a reified continuation[1] is a function `(lambda (v` κ'`) (`κ `v))`; it discards the current continuation which is local to a process but does not suspend processes executing in parallel. As explained in [20], this multiple application of `k` can lead to single or multiple final results. The different solutions are the application of `f1` to `1`, the application of `f1` to `2`, the application of `f1` to `1` followed by the application of `f1` to `2` or the application of `f1` to `2` followed by the application of `f1` to `1`.

We can *sequentialise* expression (8) by replacing parallel applications by sequential applications. We obtain

```
(f1 (call/cc (lambda (k)
               ((f2 (k 1)) (k 2)))))
```
(9)

where the continuation `k` is only applied to `1` since the evaluation order is left to right.

We can summarise the properties of the current definition of Λ_C by:

- every program not using `call/cc` or not applying a continuation always returns the same result as the sequentialised program,
- when continuations are used, multiple answers can be returned; the solution returned by the sequentialised version of a program is always a solution that this program can return,
- the number of returned solutions is not always determinate.

4 Left expressions and metacontinuations

The semantics presented in the previous section is suitable for programs not using continuations; it allows a maximal amount of parallelism thanks to symmetric continuations. Since we must restore *some* sequentiality when captured continuations are applied, we introduce higher order continuations that we call *metacontinuations*.

[1] We use the term *reified* continuation to denote the object returned by a `call/cc`. This reified continuation captures a continuation κ, called the captured continuation or the implicit continuation which is the continuation resulting from the continuation passing style translation.

Before defining this notion, we introduce the concept of *left expression*. If we wish a transparent `pcall` construct, the expression (8) should return the same result as expression (9). Therefore, we must prevent the application of `k` to `2` since `k` is applied to `1` in the sequential expression and `(k 1)` is evaluated before `(k 2)` in a left to right evaluation order.

Let `e` be an expression `(k v)` applying a continuation `k` to a value `v` and let `e` be a subexpression of `E`; we call *left expressions* of `e` in `E`, all expressions of `E` which must be evaluated before we can safely apply the continuation `k` to `v` without departing from the sequential semantics. We define this notion of left expressions only for a continuation application since we know that the semantics is correct for expressions not using continuations. The set of left expressions can be defined inductively on the structure of `E`. If `E` is equal to `e`, i.e. `e` is a toplevel expression, the set of left expressions is empty. Let us suppose that the set of left expressions of `e` in `E` is α, which we write $\mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha$ and let us examine the structure of expressions of Λ_C.

$$\mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha \supset \begin{cases} \mathcal{L}_{\texttt{e}}((\texttt{E N})) = \alpha \\ \mathcal{L}_{\texttt{e}}((\texttt{M E})) = \alpha \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall E N})) = \alpha \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall M E})) = \alpha \cup \{\texttt{M}\} \\ \mathcal{L}_{\texttt{e}}((\texttt{call/cc (lambda (x) E)})) = \begin{cases} \emptyset & \textit{if } \texttt{k} \textit{ is captured by } \texttt{call/cc} \equiv (\texttt{eq? k x}) \\ \alpha \end{cases} \end{cases}$$

While a sequential application `(M N)` does not change the set of left expressions, a parallel application `(pcall M E)` adds the expression `M` to the set of left expressions of `e` in `E`. The last rule limits the set of left expressions to the dynamic scope of the `call/cc` which captured the continuation applied in `e`: indeed, if `e=(k v)` is in the dynamic scope of a `call/cc` and if the continuation `k` applied in `e` is the continuation captured by this `call/cc` then the set of left expressions of `e` is empty. For example, in expression (8), we do not require `f1` to be evaluated in order to apply `k`. This property allows a function to use an internally reified continuation without having to synchronise with the rest of the program.

Those rules are only based on the structure of `E`. The set of left expressions is also dependent of the application rule: the body of a function is evaluated after the argument. Therefore, we can write

$$\begin{array}{lcl} \mathcal{L}_{\texttt{e}}(((\texttt{lambda (x) E) M})) = \alpha & \supset & \mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall (lambda (x) E) M})) = \alpha & \supset & \mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha \end{array}$$

Now, we can state the condition of a continuation application using the notion of left expression: *a continuation* `k` *can be applied in subexpression* `e` $=$ `(k v)` *of* `E` *if all left expressions* $\mathcal{L}_{\texttt{e}}(\texttt{E})$ *are already evaluated and have returned a value.*

A metacontinuation γ is used when applying a captured continuation κ in order to test all the left expressions $\mathcal{L}_{\texttt{e}}(\texttt{E})$ of the continuation application `e` (a subexpression of `E`). In our translation, a metacontinuation is a function of one argument: a continuation. A reified continuation `k` is represented as

$$\texttt{(lambda (v } \kappa \texttt{ } \gamma\texttt{) ((}\gamma \texttt{ } \kappa'\texttt{) v))}$$

where κ' is the captured continuation and γ is the current metacontinuation. When a reified continuation is applied, $(\gamma\ \kappa')$ is evaluated executing the following sequence:

1. the leftmost expression of `e` is searched; let `l` be this expression; let `N` be its immediate right expression in `(pcall l N)` and let γ_1 be the current metacontinuation of `(pcall l N)`,

2. if `l` is evaluated and has returned a value, the same sequence is executed with $(\gamma_1\ \kappa')$, i.e. the following left expression is searched,

3. if `l` is not evaluated, the application of the captured continuation is suspended by storing in the cell `cn` associated to `N` the value `(lambda (vm` κ γ`) ((`γ_1 κ'`) v))`. When `l` is evaluated, its continuation reads `cn` and resumes the application of `k` by evaluating $(\gamma_1\ \kappa')$,

4. if there is no left expression, the continuation κ' can be safely applied.

We said previously that the Asymmetric-CPS translation defines a total order of evaluation between expressions and that Symmetric-CPS translation defines a partial order. With our notion of metacontinuation, we have introduced a stronger partial order of evaluation: not only all subexpressions of a parallel application must be evaluated before the function body but all left expressions of a continuation application must be evaluated before applying this continuation as well.

5 Operational semantics of Λ_C

The operational semantics of Λ_C is given by a translation of Λ_C to $\Lambda_{//}$ and an operational semantics of $\Lambda_{//}$ was briefly described in section 2. The translation is presented in figure 7; it is a "continuation passing and metacontinuation passing" translation. Hence, the translation of an expression is a two arguments function: κ the implicit continuation and γ the metacontinuation. In figure 7, the first four rules define the sequential subset of the language Λ_C. In the translation of a one argument function, the implicit continuation κ is applied to a three arguments function: the initial argument, κ and γ. In the translation of a variable, the implicit continuation κ is applied to this variable. The translation of a sequential application is the classical ACPS translation where γ is added as a second parameter and is simply transmitted; γ has no role in the semantics of sequential expressions. In the translation of a `call/cc` expression, `f` is bound to the reified continuation as described before:

`(lambda (v c` γ`) ((`γ κ`) v))`

which checks left expressions before applying κ to `v`.

The last rule is the translation of a parallel application: the implicit continuations are more or less the same as symmetric continuations of figure 5. A small asymmetry is introduced: on the one hand, the implicit continuation of `M` modifies the semaphore and on the other hand, the implicit continuation of `N` does not modify the semaphore (the received value is sent back to the semaphore). Therefore, the semaphore associated to a parallel application `(pcall M N)` shows whether `M` is evaluated.

If γ is the metacontinuation of expression `(pcall M N)`, then γ is the metacontinuation of `M`: indeed, $\mathcal{L}_{\mathtt{e}}(\mathtt{E}) = \alpha \supset \mathcal{L}_{\mathtt{e}}(\mathtt{(pcall\ E\ N)}) = \alpha$. According to the rule $\mathcal{L}_{\mathtt{e}}(\mathtt{E}) = \alpha \supset \mathcal{L}_{\mathtt{e}}(\mathtt{(pcall\ M\ E)}) = \alpha \cup \{\mathtt{M}\}$, the leftmost expression of `N` is `M`. The metacontinuation of `N` checks whether `M` is evaluated. If it is the case, the metacontinuation γ is applied to the captured continuation. If not, `(lambda (vm` κ γ`) ((`γ `cont) v))` is stored in `cn` as value of `N`. Therefore, when `M` is evaluated, its continuation applies the value stored in `cn` and applies the metacontinuation γ to the captured continuation. The action to be

```
⟦x⟧              = (lambda (κ γ) (κ x))
⟦(lambda (x) M)⟧ = (lambda (κ γ) (κ (lambda (x κ γ) (⟦M⟧ κ γ))))
⟦(call/cc M)⟧    = (lambda (κ γ)
                     (let ((f (lambda (v c γ) ((γ κ) v)))
                           (γ′ (lambda (cont) (if (eq? cont κ) cont (γ cont))))))
                       (⟦M⟧ (lambda (vm) (vm f κ γ′)) γ))
⟦(M N)⟧            = (lambda (κ γ)
                     (⟦M⟧ (lambda (vm) (⟦N⟧ (lambda (vn) (vm vn κ γ))
                                            γ))
                          γ))
⟦(pcall M N)⟧    =
(lambda (κ γ)
  (let ((cn (channel)) (cm (channel)) (sem (channel)))
    (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                   (begin (receive sem)
                                          (write cm (lambda (vn x γ) (vm vn x γ)))
                                          (let ((fn (read cn)))
                                            (begin (send sem (lambda (cont s f)
                                                               (lambda (s)
                                                                (begin (s v)((γ cont) v)))))
                                                   (fn vm κ γ)))))
                                 γ)))
           (fork (lambda () (⟦N⟧ (lambda (vn)
                                   (let ((f (receive sem)))
                                     (begin (write cn (lambda (vm x γ) (vm vn x γ)))
                                            (let ((fm (read cm)))
                                              (begin (send sem f)
                                                     (fm vn κ γ))))))
                                 (lambda (cont)
                                   (let ((f (receive sem)))
                                     (f cont
                                        (lambda (v) (send sem f))
                                        (lambda (v)
                                          (begin (write cn (lambda (vm κ γ) ((γ cont) v)))
                                                 (send sem f)))))))))
           (make-store cm (lambda(vn κ γ) '()))
           (make-store cn (lambda(vm κ γ) '()))
           (make-store sem (lambda (cont s f) f)))))
```

Figure 7: Translation of Λ_C

executed by the metacontinuation of `N` depends on the value of `M`. It is implicit in the code: the function stored in the semaphore is applied in the metacontinuation of `N` but the content of the semaphore is changed by the continuation of `M`.

`fork` is a parallel construct which must appear in a sequence, it creates a process to evaluate its argument, it returns an unspecified value. In the sequence `(begin (fork exp1) exp2)`, `exp1` is evaluated in parallel with `exp2` and the value of `exp1` is discarded. Thus, our semantics must guarantee that if a continuation is applied in `exp2`, it can escape from `exp2` if and only if it is applied in the sequential definition, i.e. if `exp1` is evaluated and has returned a value.

`fork` can easily be defined thanks to `pcall`. `fork` must appear in a sequence which is syntactic sugar for a lambda application. The translation in figure 8 prevents any escape from `N` unless `M` is computed and the sequence value is returned after `M` is evaluated.

The reader might wonder why we gave a translation for `pcall` and `fork` while they can be defined as macro expansions of `future`. A natural solution for `pcall` comes from the PolyScheme technique of

```
⟦(begin (fork M) N)⟧ = ⟦((lambda (x) N) (fork M))⟧
                     = ⟦(call/cc (lambda (k) (pcall (let ((x  M)) (lambda (u) u)) (k N))))⟧
```

Figure 8: Translation rule for `fork`

symmetric continuations extended with metacontinuations for transparency reasons. With a few changes, the translation of `pcall` can be transformed in a translation for the `future` construct, essentially by introducing the notion of placeholder.

We give in figure 9 the translation of `(M (future N))` which slightly differs from the translation of `(pcall M N)`. We translate `(M (future N))` and not `(future N)` alone because we need to explicitly have the two threads evaluating in parallel in order to introduce synchronisations between them. Occurrences of `future` in different contexts reduce to `(pcall M N)` or `(M (future N))`:

`((touch (future M)) (future N))`	`(M (future N))`
`((touch (future M)) N)`	`(pcall M N)`
`(pcall M (future N))`	`(M (future N))`
`(pcall (touch (future M)) (future N))`	`(M (future N))`
`(pcall (touch (future M)) N)`	`(pcall M N)`

As opposed to `(pcall M N)`, it is always the continuation of `M` in `(M (future N))` which performs the application to either the value of `N` or a placeholder if `N` is not yet computed. The placeholder holds a function receiving a value from channel `val` and the function `touch` forces the reception of a value on this channel. Values are only sent on this channel by the process `emitter` indefinitely sending the first value received on `vali`, the other values sent by the continuation of `N` are received by the process `sink` discarding them. The definitions of `emitter` and `sink` are given in figure 10. Therefore, it is guaranteed that the placeholder holds the first value returned by `N`. If `N` returns more than once, the continuation of `N` applies the value of `M` to the value of `N` and not to the placeholder. This semantics was given by Katz and Weise in [12].

The value of an expression `E` of Λ_C is given by the value of expression

```
(let ((c0(channel)))
  (begin (fork (lambda () (⟦E⟧ κi γi)))
         (receive c0)))                                    (10)
```

according to transition rules given in figure 2, where γ_i, the initial metacontinuation, is the identity function and κ_i, the initial continuation, is defined by

κ_i `= (lambda (v) (send c0 v))`

The value of an expression `E` is the value received on channel `c0` in expression 10.

```
⟦(M (future N))⟧  =
(lambda (κ γ)
  (let ((cn (channel)) (cm (channel)) (sem (channel))
        (val (channel)) (vali (channel)))
    (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                   (begin (receive sem)
                                          (write cm (lambda (vn x γ) (vm vn x γ)))
                                          (let ((fn (read cn)))
                                            (begin (send sem (lambda (cont s f)
                                                               (lambda (s)
                                                                (begin (s v)((γ cont) v)))))
                                                   (fn vm κ γ)))))
                                 γ)))
           (fork (lambda () (⟦N⟧ (lambda (vn)
                                   (let ((f (receive sem)))
                                     (begin (send vali vn)                          ; ***
                                            (write cn (lambda (vm x γ) (vm vn x γ)))
                                            (if (not(eq? (receive val) vn))         ; ***
                                                (let ((fm (read cm)))
                                                  (begin (send sem f)
                                                         (fm vn κ γ)))
                                                (send sem f)))))
                                 (lambda (cont)
                                   (let ((f (receive sem)))
                                     (f cont
                                        (lambda (v) (send sem f))
                                        (lambda (v)
                                          (begin (write cn (lambda (vm κ γ) ((γ cont) v)))
                                                 (send sem f)))))))))
           (make-store cm (lambda(vn κ γ) '()))
           (make-store cn (lambda(vm κ γ) (vm (make-placeholder (lambda () (receive val))) κ γ)))
           (make-store sem (lambda (cont s f) f))
           (let ((the-future-value (receive vali)))                             ; ***
             (begin (fork (lambda () (emitter val the-future-value)))
                    (fork (lambda () (sink vali))))))))

(define (touch object) (if (placeholder? object) (touch ((cdr object))) object)))
```

Figure 9: Translation of `(M (future N))` and definition of `touch`

6 Example

In a previous paper [18], we explained which programming methodology could be adopted in Λ_C. We showed that programs written in a coroutine style (see [9], [8]) could be parallelised by adding parallel annotations such as `pcall`, `fork` or `future`. This approach has the advantage that the same programming style can be used to develop sequential and parallel applications. Moreover, the parallel code where annotations for parallelism are deleted gives the sequential version of the program.

For example, let us consider the producer and consumer problem. We define a producer coroutine

```
(define (emitter c v)
  (begin (send c v)
         (emitter c v)))

(define (sink c)
  (begin (receive c)
         (sink c)))
```

Figure 10: Functions `emitter` and `sink`

computing integer values from 0, transmitting them to a consumer, and a consumer receiving values from a consumer and executing an operation on them. Unlike [9], [8], we do not use the function `make-coroutine` since assignment is not existent in Λ_C. When a coroutine calls another coroutine with the function `resume`, it transmits a value and its current continuation to enable the called coroutine to resume the calling one.

```
(define producer
  (lambda (producer-job)
    (lambda (consumer-value)
      (letrec ((loop (lambda (n pair)
                       (let* ((pair (future (resume (car (touch pair)) n)))
                              (new-value (producer-job n)))
                         (loop new-value pair)))))
        (loop 0 consumer-value)))))

(define consumer
  (lambda (producer consumer-job)
    (letrec ((loop (lambda (producer)
                     (let* ((pair (resume producer 'any))
                            (producer (car pair))
                            (n (cadr pair)))
                       (consumer-job n)
                       (loop  producer)))))
      (loop producer))))

(define (resume coroutine value)
  (call/cc (lambda (k)
             (coroutine (list k value)))))
```

The coroutine system is launched by the function `run`:

```
(define (run)
  (consumer (producer (lambda (n) (+ n 1)))
            (lambda (n) (display n) (newline))))
```

In the function `producer` we underlined the annotations for parallelism. If they are removed, we obtain a sequential version. Those annotations allow a `producer` to compute the next value in parallel with the transmission of a value to the `consumer`. This process is clearly speculative since it allows to compute values before they are needed.

According to the operational semantics, several processes will be created:

- a process p_0 waiting for the final value. Since the function `run` applies the coroutine `consumer` which is an infinite loop, no final result is returned,
- a process p_1 evaluating the application of `run` and the coroutine `consumer`,
- processes created by each instance of `future` in each recursive call of the coroutine `producer`,
- processes representing memory cells.

This example shows that an infinite number of processes might be created: this raise the question of *scheduling*. We do not studied this problem in this paper but some solutions have previously been suggested like the sponsors in [5] and [19].

7 Related work

PolyScheme was initially proposed by Queinnec in [20] and [21]. The translation given in figure 5 uses the same technique of symmetric continuations; figures 7, 8 and 9 are enhancements of it. In [20] and [21], PolyScheme unfairness is outlined when returning multiple results. Queinnec's solution to this problem is to add conditions on continuations applications to insure that the number of results is execution independent although possibly greater than one. Our approach is totally opposite, we add constraints on continuations applications in order to ensure only one result, the same as in the sequential version.

Katz and Weise in [12] suggest to use a notion of *legitimacy* to give a functional program a parallel semantics equivalent to the sequential one. A process is legitimate if the code it is executing would have been executed by a sequential implementation in the absence of future. When the evaluation begins the initial process is said to be legitimate. This notion is not formally defined in [12] and an implementation with unification variables associated to processes is given in [14]. A process is legitimate if there is a unification chain existing between this process and the initial one.

Our notion of metacontinuation is the device we use to restore sequential semantics but it behaves differently from Katz and Weise's notion of legitimacy:

- We also have a kind of legitimacy notion but it is related to continuations applications and not to processes, so we have to check legitimacy only when a program explicitly applies a continuation (by checking all left expressions), and not when two processes have to synchronise through a placeholder.

- It is sufficient to test the legitimacy of an application of a continuation between the application point and `call/cc` if applied in its scope. In [12], there must be a legitimacy path between an expression and the initial expression (the top of the computation tree).

- It seems that the notion of legitimacy defined in [12] refers to a total order of evaluation as opposed to γ which is a representation of a partial order. With the legitimacy notion, one can say that an expression is legitimate only when the computation has ended while metacontinuations guarantee the legitimacy during evaluation.

However our approach is probably more conservative when applying a continuation outside the dynamic scope of the `call/cc` which captured it: we apply a continuation if we know that it is legitimate. In [12], continuations are applied independently of the legitimacy testing. Nevertheless, in the coroutine style examples we give in [18] and in section 6, continuations are applied to transmit a result to a coroutine; thus, there is no point to transmit another result if it is not needed although the next result can be searched speculatively.

In [10], Hieb and Dybvig introduced a `spawn` operator to control tree-based concurrency. They do not tackle the transparency problem since they define a parallel-or operator among others.

In [6], [7] Hammond defined a semantics of ML exceptions which could be preserved in a parallel implementation. If expression e_2 in application $e_1(e_2)$ returns an exception, it can only be raised if e_1 returns a value. If e_1 returns an exception, it will be raised. This is a definition at the level of evaluation rules à la ML without description of process interactions. Exceptions can be seen as a special case of continuations but they differ in two aspects. First, there is only one left expression to test since

exceptions (at the semantic level) are transmitted from expression to expression. Secondly, speculative computation cannot be obtained without introducing non strict operators because exceptions have a dynamic extend.

As far as we know, this is the first attempt to give a formal semantics of a functional language with continuations and transparent parallel constructs. We use a notion of higher order continuation: a metacontinuation receives a continuation in argument and returns a continuation if all left expressions are evaluated else the continuation application is suspended. We saw in the translation that a metacontinuation results from the composition of a function and the previous metacontinuation. Some related concepts were already introduced in a sequential context to define "functional jumps". In [2], Danvy and Filinski define a hierarchy of continuations allowing to abstract, as functions, contexts delimited by `shift` and `reset` operators. In [22], Queinnec and Serpette define partial continuations which can be composed.

8 Conclusion and future work

In this paper, we formalised the implementation given by Katz and Weise of a parallel functional language with first class continuations and transparent constructs for parallelism. After introducing a new concept of higher order continuation for parallelism, we gave an operational semantics for such a language.

Continuation semantics is usually used to express sequentiality. When this can be relaxed and when parallelism can be introduced, metacontinuations seem to be suitable to keep some sequential properties in a parallel framework.

Future work will follow several directions. On the theoretical side, we still have to provide a proof that each expression of Λ_C returns the same result as the same expression where parallel operators are deleted. Several steps will compose this proof: it must be proved that each expression returns only one result and that this result is the same for all execution. Then it must be shown that this result is the same as result returned by the sequentialised expression. On the practical side, we will try to apply metacontinuations to other examples where parallelism can be introduced although some sequential features should be maintained. It would also be interesting to implement a parallel system using this method and compare it with Katz and Weise's notion of legitimacy.

9 Acknowledgements

This work was partially done while visiting the laboratory for foundations of Computer Science (LFCS), University of Edinburgh. I would like to thank Rod Burstall for allowing this visit and Daniel Ribbens, my supervisor, with whom I had numerous conversations during this period. I would also like to thank David N. Turner, Christian Queinnec, Patrice Godefroid and Pierre Wolper for reading a previous draft of this paper.

References

[1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the*

sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 293–302. ACM, June 1989.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–1160, June 1990.

[3] Robert H. Halstead, Jr. Multilisp : A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[4] Robert H. Halstead, Jr. Parallel symbolic computing. *IEEE Computer*, pages 35–43, August 1986.

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

[6] Kevin Hammond. Exception handling in a parallel functional language: PSML. Technical Report CSC/89/R17, University of Glasgow. Department of Computing Science, 17 Lilybank Gardens, Glasgow, G12 8QQ., 1989.

[7] Kevin Hammond. *Implementing a Parallel Functional Languages: PSML.* Pitman, 1991.

[8] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.

[9] C.T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM conference on LISP and functional programming*, pages 293–298. ACM, 1984.

[10] 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.

[11] Soren Holmstrom. PFL : A functional language for parallel programming and its implementation. Technical Report 7, Chalmers University, 1983.

[12] 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.

[13] James S. Miller. *MultiScheme : A parallel processing system based on MIT Scheme.* PhD thesis, MIT, 1987.

[14] 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.

[15] R. Milner, D. Berry, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings of the nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1992.

[16] Robin Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall, 1989.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, 1990.

[18] Luc Moreau. Programmer dans un langage fonctionel parallèle avec continuations. In *Journées Francophones des Langages Applicatifs*, 1992.

[19] Randy B. Osborne. Speculative computation in Multilisp. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, pages 103–137. Lecture Notes 441 in Computer Science. Springer-Verlag, 1990.

[20] Christian Queinnec. Polyscheme, a semantics for a concurrent scheme. In *High Performance and Parallel Computing in Lisp Workshop*, Twickenham, England, November 1990. Europal.

[21] Christian Queinnec. Crystal Scheme. A language for massively parallel machines. In *Symposium on High Peformance Computers, Montpellier*, 1991.

[22] 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.

[23] Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language scheme. Technical report, MIT AI Lab and Indiana University Comp. Science, 1986.

[24] J. H. Reppy. First-class synchronous operations in Standard ML. Technical report, Cornell University, Department of Computer Science, 1989.

[25] Guy Lewis Steele, Jr. Rabbit: a compiler for scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.