A Parallel Functional Language With First-Class Continuations. Programming Style and Semantics

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

Abstract

We present an operational semantics for a functional language with first-class continuations and transparent constructs for parallelism `fork` and `pcall`. The sequential semantics of programs with first-class continuations is preserved when parallel evaluation is allowed, by verifying whether some expressions have returned a value before applying a continuation. These expressions are the ones that are evaluated before this continuation is applied in a left-to-right sequential order. An implementation is proposed using a notion of higher-order continuation that we call metacontinuation. This semantics is costless when first-class continuations are not used. Several programs also illustrate the programming style that can be adopted in such a language.

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

1 Introduction

There are essentially two trends to extend a functional language with parallel constructs. On the one hand, the approach adopted by the ML community [22] consists in adding to the language the notions of processes, channels, and communications as in calculi like CCS [21]. An operational semantics is given in [2] and several implementations were realised (PFL [17], CML [32]). Its main drawbacks are 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 [9], [10]. Such operators are said to be *transparent* since programs using them return the same results as their *sequentialised* versions (i.e. these programs where those operators were deleted). Thus, those parallel constructs can be seen as annotations for parallelism. Consequently, in order to write a parallel functional program, one has to write a sequential functional program and annotate it with parallel annotations (which is not a trivial task).

One feature of Scheme [31] is its ability to give the programmer access to the internal continuation. The function `call/cc` *reifies* the current continuation, i.e. `call/cc` packages up the current continuation as a first-class object, which is an escape procedure, also called reified continuation. When an escape procedure is applied to a value v, the current computation is aborted and the execution resumes at the point where the continuation was captured by `call/cc`; the value v being the value returned from that `call/cc` expression. First-class continuations are useful to define powerful control structures, such as escape mechanisms, abortion and resumption of computations, coroutines; several programming examples with continuations can be found in [14], [15].

As far as parallelism is concerned, the second approach, with transparent constructs, is commonly used to add parallelism to Scheme. However, first-class continuations gave a hard time to researchers to define a transparent `future` construct ([19], [16], [18], [20], [11]). Sequential programs are characterised by a fixed evaluation order (that we assume to be from left to right in this paper). This evaluation order does not exist any longer when parallel constructs are introduced in the functional language. However, by their abortive nature, the order in which continuations are applied is critical. Hence, a parallel program can produce a different result than its sequentialised version because a continuation c_1 can be applied before another continuation c_2, while c_2 would be applied first in the sequential program. In absence of a formal specification, the constructs for parallelism lose their transparency. An implementation of a transparent `future` construct was realised by Katz and Weise ([18], [20]), but, unlike the ML approach, a formal semantics was not provided.

In this paper we present an operational semantics for a subset of Scheme extended with transparent constructs for parallelism `fork`, `pcall` that we call Λ_C; the case of `future` is also considered. This operational semantics is a translation of Λ_C to a language, called $\Lambda_{//}$, itself specified by an operational semantics. $\Lambda_{//}$ is a functional language without first-class continuations to which CCS-style constructs for parallelism are added.

The basic idea of the semantics is to ensure that, before applying a continuation, all the expressions that are evaluated in parallel and that should be evaluated in the sequential version, have indeed returned a value. We call these expressions "*left expressions*" since they are evaluated before the continuation is applied in a left-to-right evaluation order. In our semantics, we implement the check of left expressions by a higher-order continuation also called metacontinuation.

This paper is organised as follows. First, in Sections 2 and 3, an informal and a formal semantics are presented for the languages Λ_C and $\Lambda_{//}$ respectively; some programming examples are also given. In Section 4, we distinguish symmetric and asymmetric continuations, and we propose a semantics of `pcall` based on symmetric continuations. In this semantics, `pcall` is not transparent for two reasons, which are studied in Sections 5 and 6: operands must be revaluated if a continuation is applied several times, and left expression should be taken into account. The details of the semantics of Λ_C are given in Section 7. Properties of this semantics are studied in Section 8, and we conclude this paper by a comparison between our contribution and related work.

2 The Source Language: Λ_C

In this paper, we present an operational semantics of Λ_C, a parallel functional language with first-class continuations. This operational semantics is based on the translation of Λ_C to another language $\Lambda_{//}$. In this section, we informally define Λ_C and we illustrate the programming style that can be adopted with such a language.

2.1 Informal Definition

Λ_C is a Scheme-like language [31]; its syntax is defined by the following grammar over a set of variables.

`M` ::= `x` | `(lambda (x) M)` | `(M M)` | `(call/cc M)`

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

1. A process p_1 evaluating `(fork exp)` in a sequence creates a process p_2 to evaluate `exp`; the value returned by `fork` is unspecified; the process p_1 continues to evaluate the sequence in parallel with p_2.

2. A process p_1 evaluating `(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.

3. 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.

   When the notion of placeholder is introduced in a language, one classically makes a distinction between strict and non-strict functions ([11]). When a process applies a strict function to a placeholder, the strict function requires the value contained in the placeholder. If the value is not yet computed, this process is suspended. It will be reactivated as soon as the value is computed. The fact of requiring the value of a placeholder is called "*touching*" the placeholder. As opposed to strict functions, non-strict functions do not require the value of a placeholder. For example, the function adding two values is strict since it requires them to be numbers while the function constructing a pair with two values is non-strict.

   One should also note that touching a placeholder is a recursive process: when the value of a placeholder is also a placeholder, this one is also touched until a value, different from a placeholder, is obtained.

   In Λ_C, we consider that all functions are non-strict and we view placeholders as a new data type. The programmer has to take care of providing functions with arguments the right data types. For this purpose, we add a new function called `touch`, which is a strict identity function. Now, touching arguments is performed only by the function `touch`, where specified by the programmer.

   One can compare `future` and `touch` in Λ_C with `delay` and `force`. *Lazy evaluation* or *call-by-need* in Scheme are available by the constructs `delay` and `force`. An expression `(delay exp)` returns an object called a *promise*. The value of `exp` may be computed later by applying the operator `force` to this promise. The pair `future/touch` is similar to the pair `delay/force` except that the argument of `future` is evaluated eagerly while an expression delayed by `delay` is only evaluated when forced by `exp`.

   The construct `future` was initially introduced in MultiLisp [10] as an annotation specifying which expressions could be evaluated in parallel. In the presence of first-class continuations, this annotation can lose its transparency as exposed in [11],

[18]. In [11, page 19], Hasltead gives three criteria for the semantics of parallel constructs and continuations in a parallel Scheme. We list them here:

(a) Programs using `call/cc` without constructs for parallelism should return the same results in a parallel implementation as in a sequential one.

(b) Programs that use continuations exclusively in the single-use style should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary expressions.

(c) Programs should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary subexpressions, with no restrictions on how continuations are used.

In Section 8, we show how our semantics satisfies these conditions.

2.2 Programming Examples in Λ_C

In this section, we show how the functional programming methodology can be used to develop parallel programs: we give some functional programs and use parallel annotations to parallelise them. For readability purpose, we use a notation similar to Scheme with multiple-arguments functions and syntactic sugar `let`, `begin` and `letrec`.

2.2.1 `search-first`

The first example consists in a depth-first search for the leftmost occurrence of an atom satisfying a predicate in a given S-expression. A sequential version can be written as follows.

```
(define search-first
  (lambda (tree pred)
    (call/cc
     (lambda (exit)
       (letrec ((loop (lambda (tree)
                        (cond ((atom? tree) (if (pred tree)
                                                (exit tree)
                                                '()))
                              (else (begin (loop (car tree))
                                           (loop (cdr tree))))))))
         (loop tree))))))
```

When an atom satisfying the predicate `pred` is found, the continuation bound to `exit` is applied; this continuation was captured when entering the function `search-first`. Since continuations are first-class objects and they have an unlimited extent, we can

slightly modify this function in order to return the atom satisfying the predicate and the current continuation.

```
(define search-first
  (lambda (tree pred)
    (call/cc
     (lambda (exit)
       (letrec ((loop (lambda (tree)
                        (cond ((atom? tree) (if (pred tree)
                                                (call/cc
                                                 (lambda (next)
                                                   (exit (list tree next))))
                                                '()))
                              (else (begin (loop (car tree))
                                           (loop (cdr tree))))))))
         (loop tree))))))
```

When the search for the leftmost occurrence of an atom succeeds, this definition of `search-first` returns a list composed of an atom and a continuation. If this continuation is applied on a value, the function `search-first` is resumed where it was aborted by the application of `exit` and the search for the next leftmost atom satisfying the predicate is launched.

In order to parallelise this program, one can search the left and right subtrees in parallel instead of sequentially. This is done by annotating the first call to `loop` on the left subtree with the annotation `fork`.

```
(define search-first
  (lambda (tree pred)
    (call/cc
     (lambda (exit)
       (letrec ((loop (lambda (tree)
                        (cond ((atom? tree) (if (pred tree)
                                                (call/cc
                                                 (lambda (next)
                                                   (exit (list tree next))))
                                                '()))
                              (else (begin (fork (loop (car tree)))
                                           (loop (cdr tree))))))))
         (loop tree))))))
```

When a process evaluates the expression `(fork (loop (car tree)))`, a new process is created to evaluate `(loop (car tree))` in parallel with `(loop (cdr tree))`. In this example, we use `fork` and not `future` because the `loop` function is not designed to return a value.

With such a definition, we can easily imagine a tree for which an atom satisfying the predicate is found in a right subtree before one is found in a left subtree. In this case, the continuation `exit` is applied on an atom that is not the leftmost. The purpose of

this paper is the design of a transparent `fork` construct which means that, although a leaf of a right subtree can be found first, the final result of the `search-first` function is the same as the result of the sequential version of `search-first`. In other words, the semantics of `fork` must guarantee that a result in a right subtree can only be returned if the search in the left subtree has not succeeded.

In the following sections, we will show that, when a continuation is applied in a process p, the processes executing in parallel with p are not suspended. Therefore, when the process p has found an atom satisfying the predicate, the processes which run in parallel with p keep on searching for other atoms. We are in the presence of a speculative computation: the following atoms are not known to be needed, although they are searched in parallel with the mandatory computation.

In order to display the atoms of an S-expression satisfying a predicate, we can use the function `display-atoms`.

```
(define display-atoms
  (lambda (tree pred)
    (let ((a-leaf (search-first tree pred)))
      (if (null? a-leaf)
          'end
          (begin (display (car a-leaf))
                 ((cadr a-leaf) '()))))))
```

In `display-atoms`, we begin to search for the leftmost occurrence of an atom. If the returned result is a non-empty list (composed of an atom and a continuation), the atom is displayed and the search for the following atom is initiated by applying the continuation. This process is repeated as long as atoms are found.

One should remark that the function `display-atoms` can use either the sequential or the parallel version of `search-first`. In the former case, the display and the search of atoms are interleaved like coroutines, while in the latter case, the atoms can be searched speculatively, the search preceding the display.

2.2.2 The Producer-Consumer Problem

In the previous example, the function `display-atoms` is not considered as a coroutine by the function `search-first` because it is always the same continuation bound to `exit` which is applied in `search-first`: the function `display-atoms` is always resumed at the same point. On the contrary, `search-first` is resumed by the continuation `next` where it was interrupted the last time.

Let us now consider the producer-consumer problem for which a sequential version is written in a coroutine style. Unlike [14], [15], we use a functional language without

assignment; so, a coroutine is called with a function `resume` that transmits a pair: the continuation of the caller coroutine and the value to transmit to the called coroutine. This continuation allows the called coroutine to resume its caller. The function `resume` is defined as follows.

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

We define a producer coroutine which computes integers and transmits them to a consumer, and a consumer coroutine which receives values from a producer and executes an operation on them.

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

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

The coroutine system, displaying numbers from 1 to the infinite is launched by the function `run`.

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

There are several ways to parallelise these functions. One of them consists in resuming the consumer coroutine in parallel with the computation of the next element; this is done, in the new definition of `producer`, by annotating `(resume (car pair) n)` with `future`. Therefore, the variable `pair` in `loop` will be bound to a placeholder. Consequently, we must take care of applying `car` on a cell and not on a placeholder, i.e. we have to `touch` the pair before accessing the cell. The parallel version is defined as follows.

```
(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)))))
```

This solution allows some speculative computation since new values are computed before they are displayed: indeed, `(resume (car (touch pair)) n)` is going to be evaluated in parallel with `(loop new-value pair)`. Therefore, an unbounded number of processes *can* be created to evaluate `(resume (car (touch pair)) n)` with the different values of `n`. But continuations will be applied in the expected order because a coroutine is resumed after touching the value received from the consumer.

We can also introduce parallelism without speculative computation: in the producer, we can compute the following element and resume the consumer coroutine in parallel (as in the previous example), but we allow the recursive call to `loop` to be performed only after the consumer coroutine has displayed the value. Therefore, only one value is computed in advance.

```
(define producer
  (lambda (producer-job)
    (lambda (consumer-value)
      (letrec ((loop (lambda (n pair)
                       (pcall loop (producer-job n)
                                   (resume (car pair) n)))))
        (loop 0 consumer-value)))))
```

Hence, the programmer can determine the kind of parallelism he wants, speculative or not, by choosing the expression to evaluate in parallel and by selecting the construct for parallelism.

The first parallel version shows that an unbounded number of processes might be created: this raises the question of *scheduling*. We did not study this problem in this paper but some solutions have been previously suggested like the sponsors in [11] and [26].

3 The Target Language: $\Lambda_{//}$

We recall the reader that we intend to define Λ_C by a translation to $\Lambda_{//}$. In Section 3.1, $\Lambda_{//}$ is given a formal semantics, and some programming examples can be found in Section 3.2.

3.1 Definition and Semantics

$\Lambda_{//}$ is the target language of the translation; it is defined by the following grammar over a set of variables.

$$\texttt{M} ::= \texttt{x} \mid \texttt{(lambda (x) M)} \mid \texttt{(M M)}$$

It is extented by a set of four *low level* primitives for concurrency.

`(spawn thunk)` The function `spawn` takes a thunk (function without argument) in argument, creates a new process that applies this thunk, and 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. In order to perform a communication, there must be a process ready to send a value on a channel and a process ready to receive a value on the same channel. The value returned by the function `send` is unspecified.

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

For readability purpose, we add the usual syntactic sugar `let`, `begin` and `letrec` in $\Lambda_{//}$ as we did in Λ_C.

An operational semantics of parallel ML is given in [2] by a set of transition rules similar to the ones of CCS [21]. Using a similar approach, we give a semantics to $\Lambda_{//}$; this semantics consists of a set of rules that allow us to infer transitions between configurations. A configuration is represented by the notation $\langle K, I, S\rangle \mid P$. In a configuration, K denotes a set of channels, I a set of process identifiers, S a set of locations, and P a set of processes $[p_i : e_i]$, where each process p_i is evaluating the expression $e_i \in \Lambda_{//}$. The initial configuration is $\langle \emptyset, \{0\}, \emptyset\rangle \mid \{p_0\}[p_0 : e]$ where e is the expression to evaluate. An expression such as

$$\langle K, I, S\rangle \mid P[p_n : e'] \rightarrow \langle K', I', S'\rangle \mid P'[p_n : e'']$$

describes a transition from a configuration $\langle K, I, S\rangle \mid P$ where process p_n is evaluating the expression e' to another configuration $\langle K', I', S'\rangle \mid P'$ where the same process p_n is evaluating the expression e''. In Figure 2, inference rules of the form

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

permit us to infer exp_2 from exp_1.

In Figure 2, rule 1 concerns the order of evaluation in an application. This rule specifies a left-to-right evaluation order of subexpressions. The notation $(v_1 \ldots v_i\ e'\ e_1 \ldots e_j)$ $0 \leq i \leq 2$, $0 \leq j \leq 2$, $0 \leq i + j \leq 2$ denotes applications composed of one, two, or three subterms. It identifies a term e whose left terms must be values and right terms are not evaluated yet. We have to consider these cases because `channel` is a no argument function, λ-expressions require one argument, and `eq?` and `send` require two arguments. Rule 1 can be read as follows, "knowing that from configuration $\langle K, I, S\rangle \mid P$ where process p_n evaluates e', there is a transition to configuration $\langle K', I', S'\rangle \mid P'$ where process p_n evaluates e'', we can infer that for the first configuration $\langle K, I, S\rangle \mid P$, where p_n evaluates $(v_1 \ldots v_i\ e'\ e_1 \ldots e_j)$, there is a transition to the second configuration where p_n evaluates $(v_1 \ldots v_i\ e''\ e_1 \ldots e_j)$ where left terms of e' are values". Rule 2 is the call-by-value β-reduction [27]. According to rule 3, the evaluation of a lambda-expression $\lambda x.M$ yields a triple $\langle x, M, \alpha\rangle$, called a function. One should remark that a function contains a fresh location α which allows us to compare functions with `eq?` (rule 7). Rule 4 is the evaluation rule of the function `channel`: it adds a new channel k to the set of channels

$$
\begin{array}{rcl}
p, q & \in & I, \text{ set of process identifiers} \\
\alpha & \in & S, \text{ Store, set of locations} \\
k & \in & K, \text{ set of channels} \\
v & \in & Value = Functions \cup BasicValues \cup \{\texttt{any}\} \\
\langle x, e, \alpha \rangle & \in & Functions \\
BasicValues & = & \{\texttt{spawn}, \texttt{eq?}, \texttt{send}, \texttt{receive}, \texttt{channel}\}
\end{array}
$$

Luc Moreau. Figure 1. Semantic objects

$$\frac{\langle K, I, S\rangle \mid P[p_n : e'] \rightarrow \langle K', I', S'\rangle \mid P'[p_n : e'']}{\langle K, I, S\rangle \mid P[p_n : (v_1 \ldots v_i\ e'\ e_1 \ldots e_j)] \rightarrow \langle K', I', S'\rangle \mid P'[p_n : (v_1 \ldots v_i\ e''\ e_1 \ldots e_j)]} \quad (1)$$

where $0 \leq i \leq 2$ and $0 \leq j \leq 2$ and $1 \leq i + j \leq 2$.

$$\langle K, I, S\rangle \mid P[p_n : (\langle x, e, \alpha\rangle v)] \stackrel{\beta}{\rightarrow} \langle K, I, S\rangle \mid P[p_n : e\{v/x\}] \quad (2)$$

$$\frac{\alpha \notin S}{\langle K, I, S\rangle \mid P[p_n : (\texttt{lambda (x) M})] \stackrel{\lambda}{\rightarrow} \langle K, I, S \cup \{\alpha\}\rangle \mid P[p_n : \langle x, M, \alpha\rangle]} \quad (3)$$

$$\frac{k \notin K}{\langle K, I, S\rangle \mid P[p_n : (\texttt{channel})] \stackrel{chn}{\rightarrow} \langle K \cup \{k\}, I, S\rangle \mid P[p_n : k]} \quad (4)$$

$$\frac{q \notin I}{\langle K, I, S\rangle \mid P[p_n : (\texttt{spawn}\langle(), \texttt{e}, \alpha\rangle)] \stackrel{frk}{\rightarrow} \langle K, I \cup \{q\}, S\rangle \mid P[p_n : \texttt{any}][p_q : e]} \quad (5)$$

$$\frac{k \in K}{\langle K, I, S\rangle \mid P[p_n : (\texttt{send}\ k\ v)][p_m : (\texttt{receive}\ k)] \stackrel{com}{\rightarrow} \langle K, I, S\rangle \mid P[p_n : \texttt{any}][p_m : v]} \quad (6)$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{eq?}\langle x_1, e_1, \alpha_1\rangle\langle x_2, e_2, \alpha_2\rangle)] \stackrel{eq}{\rightarrow} \langle K, I, S\rangle \mid P[p_n : \alpha_1 = \alpha_2] \quad (7)$$

Luc Moreau. Figure 2. Reduction rules for $\Lambda_{//}$

K. Rule 5 is the evaluation rule of the function `spawn`: 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 `spawn`. 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. Communications are synchronous since a process can send a value on a channel c is there is another process which is ready to receive a value on the same channel c (and vice-versa). According to rule 7, two functions are equal (`eq?`) if they have the same location, i.e. if they result from the evaluation of the same λ-expression by rule 3. Such an approach is also adopted in the definition of Scheme [31].

3.2 Programming Examples in $\Lambda_{//}$

Let us illustrate the programming style offered by $\Lambda_{//}$ by several functions that will be used in the following sections. With the primitives for parallelism defined in the previous section, we can define a store as a data structure created by the constructor `make-store`, accessed by `read` and modified by `write`. It can be modelled by a process as described in Figure 3. The value contained in the store is the value bound to the local variable `v`. This process infinitely sends and receives values on a given channel `c`, the last value received being the next to be sent. The store can be read by receiving a value and immediately sending back the same value afterwards; the store can be written by sending a new value after discarding a received value.

The reader might wonder why such a protocol is used in the definition of the store. An intuitive solution would be to receive a value on a channel when we want to read a store, and to send a value on a channel when we want to write the store. Such a solution requires the store to be a process able to choose between a `send` and a `receive`: this is a non-deterministic choice but $\Lambda_{//}$ does not have a non-deterministic choice instruction. We have decided to define $\Lambda_{//}$ without such an operator because it was superfluous for the purpose of the semantics of Λ_C.

When several processes have access to a store, one usually needs to consider the store as a critical section. Such a critical section can be implemented by a semaphore. In Figure 4, we implement semaphores using the function `make-store`. The actions `wait` and `signal` are performed by `receive` and `send` actions respectively. In the following sections, we associate a value to a semaphore. A version of the semaphore with a value also appears in Figure 4.

We call a *sink*, a process that infinitely receives values on a given channel `c`. On the contrary, an *emitter* is a process that infinitely sends the same value on a given channel.

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

(define read                          (define write
  (lambda (c)                           (lambda (c v)
    (let ((value (receive c)))             (begin (receive c)
      (begin (send c value)                       (send c v))))
             value))))
```

Luc Moreau. Figure 3. Definition of a store

```
(define make-semaphore
  (lambda (c)
    (make-store c 'any)))
```

```
(define wait
  (lambda (sem)
    (receive sem)))
```

```
(define signal
  (lambda (sem)
    (send sem 'any)))
```

```
(define make-semaphore-with-init-value
  (lambda (c init-value)
    (make-store c init-value)))
```

```
(define signal
  (lambda (sem val)
    (send sem val)))
```

Luc Moreau. Figure 4. Definition of a semaphore

Such processes are defined in Figure 5.

In Figure 6, we illustrate the semantics of $\Lambda_{//}$ by evaluating an expression according to the transition rules of Figure 2. We show only the configurations that are followed by a transition related to parallelism.

In order to simplify the presentation of this example, we have not explicitly represented the store (set of locations) of each configuration. Instead, we have called them α, though they might not all be the same. In fact, locations are used to uniquely name functions in order to be able to distinguish them with `eq?`. Our simplification is acceptable because we do not use `eq?` in this example.

4 Symmetric or Asymmetric Continuation-Passing Style

Figure 7 displays a translation of the sequential subset of Λ_C using the *continuation-passing style* or *CPS* for short. The CPS translation is an old idea in computer science; it was initially proposed by Fisher [8] and Reynolds [33], and further investigated by Plotkin [27]. Such a style is often used for denotational semantics and for program transformations in compilers [3], [4], [34], [35], [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 source term of Λ_C in brackets and the right-hand side is an expression of $\Lambda_{//}$. 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".

The basic idea of the CPS translation is to transform each function of one argument into a function of two arguments, the second being a continuation which represents what to do after the evaluation of the function. The third rule of Figure 7 is the translation of an application `(M N)`: `(lambda (`κ`) (`$[\![$`M`$]\!]$ `(lambda (vm) (`$[\![$`N`$]\!]$ `(lambda (vn) (vm vn `κ`))))))`. For a continuation κ, the translation of `M` is applied to the continuation `(lambda (vm) (`$[\![$`N`$]\!]$ `(lambda (vn) (vm vn `κ`))))`. As soon as `M` is evaluated, its value will be bound to `vm`, and the translation of `N` will be applied to the continuation `(lambda (vn) (vm vn `κ`))`. The value that `N` yields is bound to `vn` and the application `(vm vn `κ`)` is performed with `vm` which must be bound to a function of two arguments (resulting from the translation of a function of one argument).

We can also find in Figure 7 the meaning of `call/cc` which applies its argument to an *escape* procedure, which is of the form `(lambda (v `κ'`) (`κ` v))`, where κ is the

```
(define make-sink
  (lambda (c)
    (spawn (lambda ()
             (letrec ((loop (lambda ()
                              (begin (receive c)
                                     (loop)))))
               (loop))))))

(define make-emitter
  (lambda (c value)
    (spawn (lambda ()
             (letrec ((loop (lambda ()
                              (begin (send c value)
                                     (loop)))))
               (loop))))))
```

Luc Moreau. Figure 5. Definition of a sink and an emitter

Let e_1 be the following expression:

```
(let ((c (channel)))
  (make-store c 0)
  (write c 1)
  (print (read c)))
```

The initial configuration is

$$\langle \emptyset, \{p_0\}, \alpha \rangle \mid [p_0 : e_1]$$

meaning that there is only one process p_0 which evaluates e_1 and no channel has already been created. It is followed by the configuration:

$$\langle \{c_0\}, \{p_0\}, \alpha \rangle \mid [p_0 : e_2]$$

with e_2

```
(begin
  (make-store c0 0)
  (write c0 1)
  (print (read c0)))
```

After creation of the store, we obtain a new configuration:

$$\langle \{c_0\}, \{p_0, p_1\}, \alpha \rangle \mid [p_0 : e_3][p_1 : e_4]$$

with e_3 and e_4

```
(begin                          (begin
  (write c0 1)                    (send c0 0)
  (print (read c0)))              (loop (receive c0)))
```

where p_1 is the process modelling the store. The following configuration is

$$\langle \{c_0\}, \{p_0, p_1\}, \alpha \rangle \mid [p_0 : e_5][p_1 : e_6]$$

after the `write` operation with e_5 and e_6

```
(begin                          (begin
  (print (read c0)))              (send c0 1)
                                  (loop (receive c0)))
```

which reduces to

$$\langle \{c_0\}, \{p_0, p_1\}, \alpha \rangle \mid [p_0 : e_7][p_1 : e_6]$$

after the `read` operation with e_7

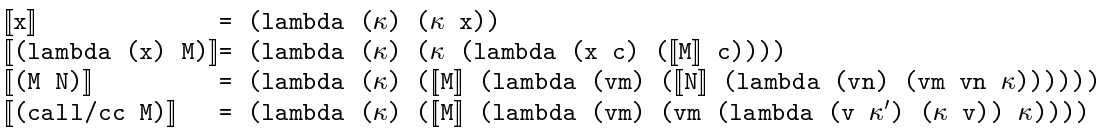
```
(begin
  (print 1))
```

The final configuration is

$$\langle \{c_0\}, \{p_0, p_1\}, \alpha \rangle \mid [p_0 :][p_1 : e_6]$$

where process p_0 has finished the evaluation of its expression and where p_1 is blocked. Indeed, there is no other process which is ready to receive a value on c_0, and since communications are synchronous, we can never apply rule 6 and the process p_1 is blocked.

Luc Moreau. Figure 6. Evaluation of an expression in $\Lambda_{//}$



Luc Moreau. Figure 7. Asymmetric continuation-passing style translation

captured continuation. Since Figure 7 contains the translation of the sequential subset of Λ_C, no parallel construct of $\Lambda_{//}$ is used. 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. Queinnec [28] 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 8. 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 of `M` stores the value of `M` in the data structure `cm`; if `N` is not yet computed, the process evaluating `M` terminates its execution. When `N` is evaluated, the continuation of `N` stores the value of `N` in a data structure, and since `M` is already evaluated, the application of the value of `M` to the value of `N` and the continuation κ is performed. The behaviour is symmetric if `N` terminates its evaluation first.

In Figure 9, we give the translation rule for the parallel application `(pcall M N)`. 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`. The operations `wait` and `signal` on a semaphore are performed when entering and exiting the critical sections. An implementation of semaphores is given in Figure 4.

2. In the translation, we do not use an `if` expression and the action `die` and the test `computed?` are implicit; this conditional expression is implemented by applying the content of the data structures `cm` and `cn` which are initialised with an empty body function and which receive a function applying `vm` to `vn` and κ.

3. The data structures `cm` and `cn` are stores for which a code is illustrated in Figure 3.

In the continuation-passing style translation (Figure 7), the continuation of `M`, (i.e. `(lambda (vm) (`$[\![$`N`$]\!]$ `(lambda (vn) (vm vn` κ`)))))` and the continuation of `N`, (i.e. `(lambda (vn) (vm vn` κ`))`) are *asymmetric* since they force the evaluation of `M` before the evaluation of `N`. They define a *left-to-right total order* of evaluation of expressions. In the definition of `pcall` (Figure 9), the continuation of `M` stores a value in store `cm`, reads store `cn` and applies its content 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

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

Luc Moreau. Figure 8. Verbose translation for a PolyScheme-style `pcall`

```
⟦(pcall M N)⟧   =
 (lambda (κ)
   (let ((cn (channel)) (cm (channel)) (sem (channel)))
     (begin (spawn (lambda () (⟦M⟧ (lambda (vm)
                                     (begin (wait sem)
                                            (write cm (lambda (vn x) (vm vn x)))
                                            (let ((fn (read cn)))
                                              (begin (signal sem)
                                                     (fn vm κ)))))))))
            (spawn (lambda () (⟦N⟧ (lambda (vn)
                                     (begin (wait sem)
                                            (write cn (lambda (vm x) (vm vn x)))
                                            (let ((fm (read cm)))
                                              (begin (signal sem)
                                                     (fm vn κ)))))))))
            (make-store cm (lambda(vn κ) '()))
            (make-store cn (lambda(vm κ) '()))
            (make-semaphore sem))))
```

Luc Moreau. Figure 9. Symmetric continuation-passing style translation for `pcall`

expressions: the body of a function is always evaluated after the subexpressions of a parallel application, but there is no order between these 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; 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 7 and 9:

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

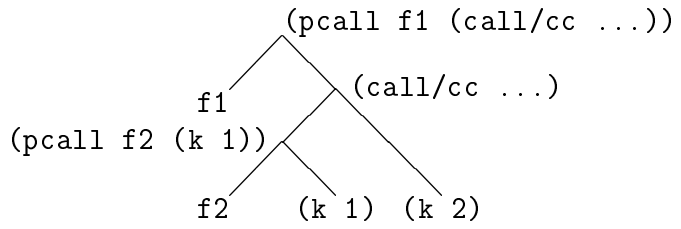
The resulting computation tree is illustrated in Figure 10. Each node represents a process evaluating an expression. When a process evaluates a `pcall` expression, two nodes are added as sons of the current node; the first argument of `pcall` being the left son and the second argument, the right son. The first node is said to be the *left brother* of the second node.

The evaluation tree of expression (8) shows that `k` is applied on `1` *and* `2` in two different processes. By analysing the translation given in Figures 7 and 9, we notice that `k` will be bound to a reified continuation[1] which is

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

with κ, the captured continuation. We can also see that when `k` is applied, the current continuation κ' is discarded. This is the reason why continuations are said to be *abortive* and can model jumps. One should remark that `pcall` creates processes and each process has its own continuation. Therefore, when a reified continuation `k` is applied, the current continuation of the *executing* process is discarded but this has no effect on processes running in parallel: i.e. processes keep on running in parallel and are not interrupted.

[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.



Luc Moreau. Figure 10. Computation tree for expression 8

Consequently, in expression (8), `k` is applied on `1` *and* `2` in two different processes. This has a strange effect: an expression can return several different results and among them some can be multiple as explained in [28].

Let us consider that `f1` is a function that adds `3` to its argument ant prints the result. Let us enumerate the different evaluation orders of expressions `f1`, `(k 1)` and `(k 2)`. Let us call `c1` and `c2` the two stores allocated when evaluating the first `pcall`.

1. According to the verbose semantics of Figure 8, if `f1` is evaluated first, the value bound to this symbol is stored in `c1` and the process evaluating `f1` dies since the argument of `f1` is not yet evaluated.

   If `(k 1)` is evaluated second, then the value `1` is stored in `c2` as the value of the `(call/cc ...)` expression, and the application is performed, yielding `4`.

   If `(k 2)` is evaluated third, then a similar execution yields the value `5`.

2. If `f1` is evaluated first, but `(k 1)` and `(k 2)` are evaluated in a reverse order, the returned values will be `5` and `4`.

3. If `(k 1)` is evaluated first, then the value `1` is stored in `c2` and the process dies since `f1` is not yet evaluated.

   If `(k 2)` is evaluated second, the value `2` is stored in `c2` and the process dies since `f1` is not yet evaluated. The value `1` which was contained in `c2` is lost.

   If `f1` is evaluated third, the application is performed yielding `5`

4. If `(k 1)` and `(k 2)` are evaluated in the reverse order, before `f1`, the result will be `4`.

In the first two cases, multiple results are returned by the expression while in the two last cases, single results are returned.

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 see that the construct `pcall`, as defined now, is not transparent since returned results are not the same for expressions (8) and (9). Indeed, among the four possible results of expression (8), only `4` is returned by expression (9).

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 number of returned solutions is not always determinate.

Furthermore, as we will see in Section 5, the solution returned by the sequentialised version of a program is not guaranteed to be returned by the current semantics of Λ_C.

5 Operand Re-Evaluation

In the asymmetric continuation-passing translation, the continuation of the operator `M` evaluates the operand `N` as indicated in the third rule of Figure 7. On the contrary, in the symmetric continuation-passing translation, the continuation of `M` does not evaluate the operand `N`, but only refers to its *last value*. Hence, if a continuation is captured in the operator and is applied several times, the symmetric and asymmetric translations can give different results as illustrated by the following example.

```
(let ((pair (pcall (pcall cons (call/cc (lambda (k1) k1)))
                   (call/cc (lambda (k2) k2)))))
  (if (and (not (number? (cdr pair)))
           (not (pair? (car pair))))
      ((cdr pair) 5)
      (if (not (pair? (car pair)))
          ((car pair) (cons 4 (car pair)))
          (number? (cdr pair)))))
```

According to the asymmetric continuation-passing translation, this program returns `#f` (after removing the `pcall` annotations). If we assume that the processes are scheduled in a left-to-right order, this program returns `#t` according to the symmetric continuation-passing translation. This difference comes from the fact that `k1` is passed a value several times in this example, which forces the reevaluation of `(call/cc (lambda (k2) k2))` in the asymmetric continuation-passing style.

In order to solve this problem, we propose in Figure 11 an asymmetric continuation-passing style translation for `pcall`, which forces the reevaluation of `N` when the continuation of `M` is passed a value several times.

In the continuation of `M`, we compare the value that is stored in `cm` with the initial value of `cm`. If they are the same (using `eq?`), it means that it is the first time that a

```
⟦(pcall M N)⟧   =
 (lambda (κ)
   (let ((cn (channel)) (cm (channel)) (sem (channel)))
     (begin (spawn (lambda ()
                     (⟦M⟧ (lambda (vm)
                            (begin (wait sem)
                                   (let ((oldcm (read sem)))
                                     (if (init-cm? oldcm)
                                         (begin
                                           (write cm (lambda (vn x) (vm vn x)))
                                           (let ((fn (read cn)))
                                             (begin (signal sem)
                                                    (fn vm κ))))
                                         (begin
                                           (signal sem)
                                           (⟦N⟧ (lambda (vn) (vm vn κ)))))))))))
            (spawn (lambda () (⟦N⟧ (lambda (vn)
                                     (begin (wait sem)
                                            (write cn (lambda (vm x) (vm vn x)))
                                            (let ((fm (read cm)))
                                              (begin (signal sem)
                                                     (fm vn κ))))))))
            (make-store cm init-cm)
            (make-store cn (lambda(vm κ) '()))
            (make-semaphore sem))))

(define init-cm (lambda(vn κ) '()))
(define (init-cm? x) (eq? x init-cm))
```

Luc Moreau. Figure 11. Asymmetric continuation-passing style translation for `pcall`

value is passed to the continuation of `M`; so we can perform the same actions as in Figure 9. Otherwise, if they differ, we have to evaluate again the operand `N`.

6 Left Expressions

Although some asymmetry was restored in the semantics presented in Section 5, the `pcall` construct is not yet transparent because programs can return several results, when continuations are applied. Let us introduce the notion of *left expression* which is related to our new semantics of continuations.

In the expression `(pcall M N)`, `M` is said to be a left expression of `N` since `M` is evaluated before `N` in the sequentialised expression `(M N)`. The set of left expressions of a given expression can be known *at run-time* by examining the execution tree. For a given node in an execution tree, the set of left expressions is the set of nodes which are left brothers of nodes between this node and the top node. For example, `f1` and `(pcall f2 (k 1))` are left expressions of `(k 2)` in expression (8) as shown in Figure 10.

Using this notion of left expression, we can state the condition that must be satisfied before applying a continuation: a continuation `k` can be applied in a subexpression `(k v)` of `E`, if all left expressions of `(k v)` in `E` have already returned a value.

We can even refine this condition. In the expression

```
(pcall ⟨f1⟩ (call/cc (lambda (k) (pcall ⟨f2⟩ (k 1)))))
```

`k` can be applied to `1` as soon as ⟨`f2`⟩ has returned a value, independently of ⟨`f1`⟩. Therefore, when a continuation is applied in the dynamic extent of the `call/cc` by which it was reified, the left expressions which should be checked are restricted to the expressions whose lifetime is included in this extent.

Hence, in terms of the computation tree, the set of left expressions of a given node (where a continuation is applied), is the set of nodes which are left brothers of nodes between this node and either the top of the tree or the node where this continuation was reified.

We can give an algorithm that implements the condition that must be satisfied before applying a continuation. Let `e` be `(k v)` an expression where the continuation bound to `k` is applied on a value. Let S be the set of left expressions of `e`; this set is computed using the execution tree of the expression `E` that contains `e` as a subexpression. We suppose that the set S is ordered: the closest left expression (i.e. the innermost) being the first and the furthest left expression (i.e. the outermost) being the last. In order to determine whether the continuation bound to `k` can be applied, we proceed using the following loop.

1. If S is empty, then we can safely apply the continuation to the value.
2. If S is not empty, then let `l` be the closest left expression of `e`,
   (a) if `l` is evaluated, go to point 1 with the new set $S\backslash\{\texttt{l}\}$
   (b) if `l` is not evaluated, the application of the continuation should be suspended. As soon as `l` gets evaluated, the algorithm can be resumed with $S\backslash\{\texttt{l}\}$.

With our notion of left expressions, we have introduced a more constrained partial order of evaluation: not only all subexpressions of a parallel application must be evaluated before the body of a function, but all left expressions of a continuation application must also be evaluated before applying this continuation as well.

7 Operational Semantics of Λ_C

In Section 4, we presented a first attempt of a semantics of Λ_C; we analysed its weakness and suggested two corrections in Sections 5 and 6. In Section 7, we follow these suggestions to write the semantics as a translation from Λ_C to $\Lambda_{//}$.

In Section 6, we described an algorithm, based on the notion of left expression, to determine whether a continuation could be applied on a value. In the translation, the test of left expressions is implemented by a *higher-order continuation* or *metacontinuation*. Although some left expressions can be determined at translation-time, we cannot find all of them at this moment. It is the role of the metacontinuation to find all left expressions at run-time. Such a metacontinuation is passed during the computation, accumulating information about left expressions. The translation is presented in Figure 12; 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 12, 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 on a three arguments function: the initial argument, κ, and γ. Therefore, metacontinuations are passed when applying λ-expressions as it was the case for continuations in Figure 7.

We use the term higher-order continuation for γ because, in a reified continuation, a metacontinuation γ is applied on a continuation κ:

```
(lambda (v c γ) ((γ κ) v))
```

where κ is the captured continuation and γ is the current metacontinuation. As in the definition of the sequential language (Figure 7), the reified continuation discards the current continuation `c`. Instead of blindly applying the captured continuation κ to `v`, the metacontinuation γ is first applied to κ, and then `v`. By definition of γ, the effect of `((`γ κ`) v)` is the same as the effect of `(`κ `v)` if all left expressions are evaluated.

If γ is the metacontinuation of expression `(pcall M N)`, the metacontinuation of `M` is also γ (since it is known at translation-time that left expressions of `M` are the same as those of `(pcall M N)`). We can now examine the metacontinuation of `N` knowing that the metacontinuation of `(pcall M N)` is γ; it is copied below.

```
(lambda (cont)
  (let ((f (wait sem)))
    (f cont
       (lambda (v) (signal sem f))
       (lambda (v)
         (begin (write cn (lambda (vm κ γ)
                            ((γ cont) v)))
                (signal sem f))))))
```

It takes a continuation `cont` in argument, and enters the critical section by `wait`ing the semaphore. The value associated with the semaphore is applied to the continuation `cont` and two functions. The initial value of the semaphore is `(lambda (cont s f) f)`.

Therefore, when we apply such a metacontinuation to a continuation, and a value `v`, the following function

```
(lambda (vm κ γ)
  ((γ cont) v))
```

is stored in `cn` and the critical section is closed. At this time, the application of the continuation is suspended because the left expression has not been evaluated. When it gets evaluated, the value stored in `cn` is applied and the the test for the application of the continuation is resumed. One should also note that when a left expression is evaluated (i.e. a term `M` in `(pcall M N)`), the value associated with the semaphore changes; it becomes

```
(begin (s v)
       ((γ cont) v))
```

which immediately applies the metacontinuation of the parent expression on the captured continuation.

The algorithm to determine whether we can apply a captured continuation κ to a value is implemented by the metacontinuation γ, it is executed by evaluating `((`γ κ`) v)`:

```
⟦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 (spawn (lambda ()
                    (⟦M⟧ (lambda (vm)
                           (let ((f (wait sem)))
                             (let ((oldcm (read cm)))
                               (if (init-cm? oldcm)
                                   (begin
                                     (write cm vm)
                                     (let ((fn (read cn)))
                                       (signal sem (lambda (cont s f)
                                                     (lambda (v)
                                                       (begin(s v)
                                                             ((γ cont) v)))))
                                       (fn vm κ γ)))
                                   (begin
                                     (signal sem f)
                                     (⟦N⟧ (lambda (vn) (vm vn κ γ)) γ))))))
                         γ)))
           (spawn (lambda ()
                    (⟦N⟧ (lambda (vn)
                           (let ((f (wait sem)))
                             (write cn (lambda (vm x γ) (vm vn x γ)))
                             (let ((fm (read cm)))
                               (begin (signal sem f)
                                      (fm vn κ γ)))))
                         (lambda (cont)
                           (let ((f (wait sem)))
                             (f cont
                                (lambda (v) (signal sem f))
                                (lambda (v)
                                  (write cn (lambda (vm κ γ)
                                              ((γ cont) v)))
                                  (signal sem f)))))))))
           (make-store cm init-cm)
           (make-store cn (lambda(vm κ γ) '()))
           (make-semaphore-with-init-value sem (lambda (cont s f) f)))))

(define init-cm (lambda(vn κ γ) '()))
(define (init-cm? x) (eq? x init-cm))
```

Luc Moreau. Figure 12. Translation of Λ_C

1. The metacontinuation γ knows the first left expression of `e` (by definition of γ at translation-time); 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 algorithm continues with `((`γ_1 κ`) v)`, i.e. the next left expression is tested.

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 algorithm by evaluating `((`γ_1 κ`) v)`.

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

It remains to determine whether we are in the dynamic extent of a `call/cc`. In the translation of `call/cc`, a new metacontinuation γ' is defined:

`(lambda (cont) (if (eq?  cont` κ`) cont (`γ `cont)))`

It compares its argument `cont` with the captured continuation κ. If they are equal, it means that we try to apply a captured continuation in the dynamic extent of the `call/cc` by which it was reified. Therefore, no left expression remains to be tested before applying `cont`.

Now that we have defined `pcall`, there are still two other parallel constructs of Λ_C which need to be defined: `fork` and `future`. The rest of this section is dedicated to their definitions.

The parallel construct `fork` must appear in a sequence: it creates a process to evaluate its argument and 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.

We can easily define `fork` thanks to `pcall`. The translation in Figure 13 prevents any escape from `N` unless `M` is computed and the sequence value is returned after `M` is evaluated.

Now let us examine the case of the `future` construct. With a few changes, the translation of `pcall` can be transformed into a translation for the `future` construct,

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

Luc Moreau. Figure 13. Translation rule for `fork`

essentially by introducing the notion of placeholder. The abstract data type placeholder is defined by the constructor `make-placeholder`, the accessor `placeholder-value` and the predicate `placeholder?`. They are defined by the following rules.

$$\frac{\alpha \notin S}{\langle K, I, S\rangle \mid P[p_n : (\texttt{make-placeholder}\ v)] \rightarrow \langle K, I, S \cup \{\alpha\}\rangle \mid P[p_n : [\alpha, v]]} \tag{10}$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{placeholder-value}\ [\alpha, v])] \rightarrow \langle K, I, S\rangle \mid P[p_n : v] \tag{11}$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{placeholder?}\ v)] \rightarrow \langle K, I, S\rangle \mid P[p_n : v \in \mathit{Placeholder} \rightarrow \#t, \#f] \tag{12}$$

According to rule 10, the constructor `make-placeholder` creates a new placeholder which is represented by a pair $[\alpha, v]$ containing a location α and a value v. By rule 11, the accessor `placeholder-value` returns the value v of a placeholder $[\alpha, v]$. Rule 12, gives the immediate definition of the predicate `placeholder?`. In appendix, we give all the rules and all the semantic domains for a complete definition of $\Lambda_{//}$.

We are going to 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))`	is equivalent to	`(M (future N))`
`(pcall (touch (future M)) (future N))`		`(M (future N))`
`(pcall (touch (future M)) N)`		`(pcall M N)`

With these properties, `(future N)` will be defined in all contexts if we give a semantics to `(M (future N))`. We display in Figure 14 the translation of `(M (future N))`. A placeholder is a data structure that holds a channel. Values are sent on this channel only by an emitter process indefinitely sending the first value received on π_1. It is the continuation of `N` which sends values on π_1 the first time it is called. The constructor for an emitter process is `make-emitter`, defined in Figure 5. With the channels π_1 and π_2, we can guarantee that the channel held in the placeholder yields the first value returned by `N`.

If the continuation of `N` is passed a value more than once, it applies the value of `M` to the value it receives and not to the placeholder. This semantics is the one proposed by Katz and Weise in [18].

In Figure 15, we give the translation for the `touch` operator. Since `touch` is a strict identity function, its operand `M` is evaluated, its value is bound to `vm`, and the function $\texttt{touch}_{//}$ is applied to `vm`. The function $\texttt{touch}_{//}$ returns its argument if it is not a

```
⟦(M (future N))⟧  =
(lambda (κ γ)
  (let ((cm(channel)) (cn(channel)) (sem (channel)) (π1(channel)) (π2(channel)))
       (κn (lambda (π cm cn)                ; continuation of N parameterised
             (lambda (vn)                   ; by placeholder π and channels cm, cn
               (let ((f (wait sem)))
                 (let ((vm    (read cm))
                       (oldvn (read cn)))
                   (if (init-cn? oldvn)
                       (begin (write cn (lambda (vm κ γ) (vm vn κ γ)))
                              (signal sem f)
                              (determine! π vn))
                       (begin (signal sem f)
                              (vm vn κ γ)))))))))

  (define init-cm (lambda(vn κ γ) '()))
  (define init-cn (lambda(vm κ γ) (vm (make-placeholder π2) κ γ)))
  (define (init-cm? x) (eq? x init-cm))
  (define (init-cn? x) (eq? x init-cn))

    (spawn (lambda ()
             (⟦M⟧ (lambda (vm)
                    (let ((f (wait sem)))
                      (let ((oldcm (read cm)))
                        (if (init-cm? oldcm)
                            (begin (write cm vm)
                                   (let ((fn (read cn)))
                                     (signal sem (lambda (cont s f)
                                                   (lambda (v)
                                                     (begin(s v)
                                                           ((γ cont) v)))))
                                     (fn vm κ γ)))
                            (let ((π1'(channel))(π2'(channel))(cm(channel))(cn(channel)))
                              (store cm vm)
                              (store cn init-cn)
                              (signal sem f)
                              (spawn (lambda () (⟦N⟧ (κn π1' cm cn) γ)))
                              (make-emitter π2' (receive π1'))
                              (vm (make-placeholder π2') κ γ))))))
                  γ)))
    (spawn (lambda () (⟦N⟧ (κn π1 cm cn)
                           (lambda (cont)
                             (let ((f (wait sem)))
                               (f cont
                                  (lambda (v) (signal sem f))
                                  (lambda (v)
                                    (write cn (lambda (vm κ γ) ((γ cont) v)))
                                    (signal sem f)))))))))
    (make-store cm init-cm)
    (make-store cn init-cn)
    (make-semaphore-with-init-value sem (lambda (cont s f) f))
    (make-emitter π2 (receive π1)))
```

Luc Moreau. Figure 14. Translation of `(M (future N))`

```
⟦(touch M)⟧  =  (lambda (κ γ)
                  (⟦M⟧ (lambda (vm) (κ (touch// vm)))
                       γ))

(define (touch// object) (if (placeholder? object)
                             (touch// (receive (placeholder-value object)))
                             object))

(define (determine! π v) (send π v))
```

Luc Moreau. Figure 15. Translation of `(touch M)`

placeholder. Otherwise, `touch`$_{//}$ receives a value on the channel contained in the placeholder, and repeats this action until the value is no longer a placeholder. The function `touch`$_{//}$ suspends the current process if `N` has not yet returned a value, by waiting for a communication with a `receive`.

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

```
(let ((c0 (channel)))
  (begin (spawn (lambda () ([[E]] κ_i γ_i)))
         (receive c0)))                                    (13)
```

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 (13).

8 Properties of the Semantics

In another paper [25], we define the CPP-calculus as an extension of the call-by-value lambda-calculus [27] with a control operator `call/cc`. The CPP-calculus differs from Felleisen and Friedman's λ_c-calculus [5], [7] because `call/cc` is not the origin of a bottleneck in the CPP-calculus. The CPP-calculus contains reduction rules that allow the capture of continuations in any context, even when left expressions are not evaluated. Furthermore, the CPP-calculus contains a mechanism to delimit the extent of a `call/cc` expression, which is used to recognise the application of a continuation in the extent of its `call/cc` (which is usually calle a *downward use*). The rules that caracterise the CPP-calculus are sound because the were proved to preserve the observational equivalence. Two expressions M and N are observationally equivalent, if for all contexts $C[\ ]$, either $C[M]$ and $C[N]$ both terminate or both do not terminate.

In conclusion, as far as termination properties are concerned, a parallel program (where continuation can be captured in any context and where continuations are applied when left expressions are evaluated) is undistinguishable from its sequentialised counterpart (that is, the same program evaluated sequentially). This proves that constructs for parallelism can be seen as annotations for parallel evaluation which do not change the meaning of programs.

Our semantics of `fork` and `pcall` satisfies Halstead's criteria enumerated in Section 2.1. According to our proof, programs not using parallel constructs return the same results as in the sequential semantics. Parallel programs are observationnally equivalent

to their sequentialised versions. In our proof, we do not have to distinguish the single or multiple use of continuations.

The semantics proposed in Figure 14 does not insure that `future` is an annotation. Indeed, we assumed that the programmer had to judiciously add the `touch` construct, so that the pair `future/construct` could be considered as transparent. Our approach is closer to `delay/force` operators for lazy evaluation, where the programmer has also to explicitly use `force`.

9 Related Work

PolyScheme was initially proposed by Queinnec [28], [29], from whom we borrowed the technique of symmetric continuations in Figure 9. In Figures 12, 13, and 14, we have added higher-order continuations to preserve the sequential semantics, and we have forced the reevaluation of operands when serveral values were passed to the continuation of an operator. PolyScheme [28] [29] unfairness is outlined when returning multiple results. Queinnec solves this problem by adding 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. Queinnec [30] distinguishes *multiplicative* `pcall` from *additive* `pcall`. An expression `(pcall M N)` is *multiplicative* if all values of `M` are applied to all values of `N` (when `M` and `N` multiply return results). An expression `(pcall M N)` is *additive* if each new value of `M` is applied to the last value of `N` or if the last value of `M` is applied to each new value of `N`. Our `pcall` is neither additive nor multiplicative because we force the reevaluation of the operand when several values are passed to the continuation of the operator .

Katz and Weise [18] 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 [18] and an implementation with unification variables associated to processes is given in [20]. 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 appli-

cations 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 extent. In [18], there must be a legitimacy path between an expression and the initial expression (the top of the computation tree).

- 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 extent of the `call/cc` which captured it: we apply a continuation if we know that it is legitimate. In [18], continuations are applied independently of the legitimacy testing. Nevertheless, in the coroutine-style examples we give in [24] and in Section 2.2, 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.

Felleisen and Friedman [5] defined the λ_c-calculus, an extension of the call-by-value λ-calculus with the control operator $\mathcal{C}$ which can model `call/cc`. In the λ_c-calculus, a $\mathcal{C}$ operator can capture a continuation iff it appears in an *applicative* context. An applicative context $C[]$ is a context such that all left expressions of [] are values. Our semantics allows more parallelism than their calculus since we do not require left expressions to be evaluated to capture a continuation. Moreover, their calculus penalises continuation application when a continuation is applied in the dynamic extent of the `call/cc` by which it was reified since all left expressions should have returned a value, even the ones outside the scope of the `call/cc`. Felleisen and Friedman [6] define the operator $\mathcal{F}$ and they describe a parallel evaluation strategy but they impose the same constraints on the capture and the application of continuations.

Hammond [12], [13] defines a semantics of ML exceptions which can 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. His work is less general than our approach because exceptions can be considered as a special case of continuations.

10 Conclusion

We have presented a semantics of first-class continuations in a parallel functional language, which requires to evaluate left expressions before being allowed to apply a continuation. This semantics is costless for programs not using continuation. We have implemented this semantics using higher-order continuations which explicitly represent the left-to-right evaluation order.

This semantics gives parallel programs the same meaning as sequential programs because parallel operators are transparent. This approach allows the re-usability of code (parallel programs can still run sequentially) and allows the programmer to use a single programming methodology for developping both sequential and parallel applications.

11 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, Vincent Kieffer, Patrice Godefroid, Pierre Wolper, and the anonymous referees for their helpful comments.

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] Dave Berry, Robin Milner, and Dave Turner. A Semantics for ML Concurrency Primitives. In *Proceedings of the Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1992.

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

[4] Olivier Danvy and Julia L. Lawall. Back to Direct Style II: First-Class Continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 299–310, June 1992.

[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*, number 259 in Lecture Notes in Computer Science, pages 206–223. Springer-Verlag, 1987.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theoretical Computer Science (North-Holland)*, 52(3):205–237, 1987.

[8] 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.

[9] Robert H. Halstead, Jr. Multilisp : A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[10] Robert H. Halstead, Jr. Parallel Symbolic Computing. *IEEE Computer*, pages 35–43, August 1986.

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

[12] 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.

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

[14] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and Coroutines. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 293–298. ACM, 1984.

[15] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.

[16] 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.

[17] Sören Holmström. PFL : A Functional Language for Parallel Programming and its Implementation. Technical Report 7, Chalmers University, 1983.

[18] 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.

[19] James S. Miller. *MultiScheme : A Parallel Processing System Based on MIT Scheme.* PhD thesis, MIT, 1987.

[20] James S. Miller and Barbara 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.*, number 441 in Lecture Notes in Computer Science, pages 138–160. Springer-Verlag, 1990.

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

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

[23] Luc Moreau. An Operational Semantics for a Parallel Language with Continuations. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE'92)*, number 14 in Lecture Notes in Computer Science, pages 415–430, Paris, June 1992. Springer-Verlag.

[24] Luc Moreau. Programming in a Parallel Functional Language with Continuations (in French). In *Avancées Applicatives. Journées Francophones des Langages Applicatifs (JFLA'92)*, volume 76–77, pages 130–153, Tréguier, France, February 1992. BIGRE.

[25] Luc Moreau and Daniel Ribbens. Sound Rules for Parallel Evaluation of a Functional Language with callcc. In *ACM conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 125–135, Copenhagen, June 1993. ACM.

[26] 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.*, number 441 in Lecture Notes in Computer Science, pages 103–137. Springer-Verlag, 1990.

[27] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.

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

[29] Christian Queinnec. Crystal Scheme. A Language for Massively Parallel Machines. In M. Durand and F. El Dabaghi, editors, *Symposium on High Peformance Computers*, pages 91–102, Montpellier (France), 1991. North-Holland.

[30] 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.

[31] Jonathan Rees and William Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[32] John Hamilton Reppy. First-Class Synchronous Operations in Standard ML. Technical report, Cornell University, Department of Computer Science, 1989.

[33] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *25th ACM National Conference*, pages 717–740, 1972.

[34] 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.

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

A Formal Definition of $\Lambda_{//}$

In this section we give the complete definition of $\Lambda_{//}$, included semantics objects like placeholders and unique values and their associated rules.

$$
\begin{array}{rcl}
p, q & \in & I, \text{ set of process identifiers} \\
\alpha & \in & S, \text{ set of locations} \\
k & \in & K, \text{ set of channels} \\
v & \in & Value = Functions \cup BasicValues \cup Placeholders \cup Unique \cup Booleans \cup \{\texttt{any}\} \\
\langle x, e, \alpha \rangle & \in & Functions \\
[\alpha, \langle x, M, \alpha' \rangle] & \in & Placeholders \\
\{\alpha, v\} & \in & Unique \\
\#t, \#f & \in & Booleans \\
BasicValues & = & \{\texttt{spawn}, \texttt{eq?}, \texttt{send}, \texttt{receive}, \texttt{channel}\}
\end{array}
$$

Luc Moreau. Figure 16. Semantic objects

$$\frac{\langle K, I, S\rangle \mid P[p_n : e'] \to \langle K', I', S'\rangle \mid P'[p_n : e'']}{\langle K, I, S\rangle \mid P[p_n : (v_1 \ldots v_i\ e'\ e_1 \ldots e_j)] \to \langle K', I', S'\rangle \mid P'[p_n : (v_1 \ldots v_i\ e''\ e_1 \ldots e_j)]} \tag{14}$$

where $0 \leq i \leq 2$ and $0 \leq j \leq 2$ and $1 \leq i + j \leq 2$.

$$\langle K, I, S\rangle \mid P[p_n : (\langle x, e, \alpha\rangle v)] \xrightarrow{\beta} \langle K, I, S\rangle \mid P[p_n : e\{v/x\}] \tag{15}$$

$$\frac{\alpha \notin S}{\langle K, I, S\rangle \mid P[p_n : (\texttt{lambda (x) M})] \xrightarrow{\lambda} \langle K, I, S \cup \{\alpha\}\rangle \mid P[p_n : \langle x, M, \alpha\rangle]} \tag{16}$$

$$\frac{k \notin K}{\langle K, I, S\rangle \mid P[p_n : (\texttt{channel})] \xrightarrow{chn} \langle K \cup \{k\}, I, S\rangle \mid P[p_n : k]} \tag{17}$$

$$\frac{q \notin I}{\langle K, I, S\rangle \mid P[p_n : (\texttt{spawn}\langle(), \texttt{e}, \alpha\rangle)] \xrightarrow{frk} \langle K, I \cup \{q\}, S\rangle \mid P[p_n : \texttt{any}][p_q : e]} \tag{18}$$

$$\frac{k \in K}{\langle K, I, S\rangle \mid P[p_n : (\texttt{send}\ k\ v)][p_m : (\texttt{receive}\ k)] \xrightarrow{com} \langle K, I, S\rangle \mid P[p_n : \texttt{any}][p_m : v]} \tag{19}$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{eq?}\langle x_1, e_1, \alpha_1\rangle\langle x_2, e_2, \alpha_2\rangle)] \xrightarrow{eq} \langle K, I, S\rangle \mid P[p_n : \alpha_1 = \alpha_2] \tag{20}$$

$$\frac{\alpha \notin S}{\langle K, I, S\rangle \mid P[p_n : (\texttt{make-placeholder}\ v)] \to \langle K, I, S \cup \{\alpha\}\rangle \mid P[p_n : [\alpha, v]]} \tag{21}$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{placeholder-value}\ [\alpha, v])] \to \langle K, I, S\rangle \mid P[p_n : v] \tag{22}$$

$$\langle K, I, S\rangle \mid P[p_n : (\texttt{placeholder?}\ v)] \to \langle K, I, S\rangle \mid P[p_n : v \in Placeholder \to \#t, \#f] \tag{23}$$

$$\frac{\langle K, I, S\rangle \mid P[p_n : e] \to \langle K', I', S'\rangle \mid P'[p_n : \#t]}{\langle K, I, S\rangle \mid P[p_n : (\texttt{if}\ e\ e_1\ e_2)] \to \langle K', I', S'\rangle \mid P'[p_n : e_1]} \tag{24}$$

$$\frac{\langle K, I, S\rangle \mid P[p_n : e] \to \langle K', I', S'\rangle \mid P'[p_n : \#f]}{\langle K, I, S\rangle \mid P[p_n : (\texttt{if}\ e\ e_1\ e_2)] \to \langle K', I', S'\rangle \mid P'[p_n : e_2]} \tag{25}$$

Luc Moreau. Figure 17. Reduction rules for $\Lambda_{//}$

List of Figures