PolyScheme: a Semantics for a Concurrent Scheme

"High Performance and Parallel Computing in Lisp" workshop, 12-13 November 1990, Twickenham, UK.

Christian Queinnec*
École Polytechnique & INRIA–Rocquencourt

Abstract

The Scheme language does not fully specify the semantics of combination: the evaluation order of the terms composing a combination is left indeterminate. We investigate in this paper a different semantics for Scheme where terms of combinations are evaluated concurrently. The resulting semantics models a language with concurrent threads sharing a common workspace. The semantics is given in terms of denotational semantics and uses resumptions as well as a choice operator: *oneof* which mimics a scheduler. An alternate definition for this operator lets appear the classical powerdomains. The main interest of this operator is to offer a formalization that can be read with an operational point of view while keeping a firm theoretical base.

Scheme also offers first class continuations with indefinite extent; we examine some semantics for continuations with respect to concurrency. Each of these semantics is a natural extension of the sequential case of regular Scheme. Still they strongly differ in their observed behaviours.

The resulting language, named PolyScheme, offers much of the features of current concurrent Lisp (or Scheme) dialects thanks to the sole extension of its combination semantics and without any explicit specialized construct dealing with concurrency.

Keywords: Lisp, Scheme, continuation, resumption, concurrency, denotational semantics.

The Scheme language [Rees & Clinger 86] does not specify the evaluation order of the terms of a combination i.e. application. The usual behaviour for Lisp-like languages is to evaluate terms from left to right due to the nature of lists which head is easily extracted. Clever compilers can improve the generated code if they can rearrange the computation of the arguments of the application provided that no side-effects occur. Scheme goes further since not only the arguments of the application can be computed in whatever order but so can be the functional term. Moreover side-effects must explicitely be sequentialized thanks to `begin` (or `progn` in Lisp). Nevertheless Scheme is always a sequential language since the body of a function is computed after its arguments.

To say that the evaluation order is indeterminate only means that there exists a sequential evaluation order but it is left unspecified. Thus one must not write code depending on it. This remark is crucial since the following Scheme program always returns true: `(let ((x 1)) (car (list (equal? x x) (set! x 2) )))`

During the comparison, performed by `equal?`, the variable `x` is equal to 1 or 2 depending on the evaluation order but in any case will always be equal to itself. If we were to accept that terms composing applications are computed concurrently then the final result of the previous excerpt may be true or false according to the exact interleaving of the assignment within the comparison.

*Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, Internet: `queinnec@poly.polytechnique.fr` This work has been partially funded by Greco de Programmation.

This paper contains the denotational semantics of a variant of Scheme where we relax the sequentiality of the evaluation order. Terms of a form are evaluated concurrently: when they are all computed, the function obtained from the first term is applied on the values of the other terms. This semantics is presented in section 2 and uses resumptions [Schmidt 86, page 279]. A comparison of this semantics against Multilisp and Qlisp appears in section 7. Scheme also offers first class continuations which behaviour in such a concurrent universe is imprecise. We successively give two semantics for continuations in section 3 which lead to a notion of independent process. These semantics are natural extensions of the sequential case but they strongly differ and thus lead to different uses. Some programming examples appear in section 5.

The denotation uses a choice operator able to pick one element from a collection thus simulating a random scheduling policy. This trick along with metacontinuations makes easy to reformulate the denotation with powerdomains, see section 6.

This paper does not present new constructs to deal with concurrency, it only investigates how to denotationally specify a slightly modified Scheme where some constraints are relaxed and investigate the relationship between concurrency and continuations. It may as well be seen as an exercize in denotational semantics for a language that may be evaluated on a multiprocessor with a single shared memory. The presented denotation is executable: it is a regular Scheme program which simulates a multiprocessor with a single shared memory. Needless to say, it is not efficient. Eventually PolyScheme is the nickname we adopt for this attempt.

1 Current State of Scheme

It seems interesting to briefly comment the current state of the denotation of Scheme since we will depart from it. The *standard* reference is that of [Rees & Clinger 86]. The fact that "*the operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments*" is specified by the following "trick"[1]: a *permute* function and its inverse *unpermute* are introduced such that arguments are scrambled before sequential evaluation and then recast in order before the effective application:

$$\begin{aligned}\mathcal{E}[\![(E_0\ E^*)]\!] = \lambda\rho\kappa.\ \mathcal{E}^*(&permute(< E_0 > \S E^*))\\ &\rho\\ &(\lambda\epsilon^*.((\lambda\epsilon^*.applicate(\epsilon^* \downarrow 1)(\epsilon^* \dagger 1)\kappa)\\ &\qquad(unpermute\ \epsilon^*)))\end{aligned}$$

It is also explicitely said that two different calls to *permute* with similar arguments may not lead to the same permutation i.e. the evaluator is free to choose any order it wants and is not stuck to use the same order each times it processes the same application. Thus the following excerpt: `(let loop ((n 0)(p 4)) (loop (display (+ 1 n)) (display (+ 1 p))) )`

may well print the sequence[2]: `15627348...`

Naturally a sequential choice such as always processing terms from right to left is allowed and is sufficiently unusual to, sometimes, remind programmers that they have to use explicit sequentialization via `progn` or `begin` whenever they want things to be done orderly. We propose to go further and investigate a variant of Scheme where terms of a form are computed concurrently.

2 Semantics of a Concurrent Scheme

Denotational semantics [Stoy 77, Schmidt 86] is a standard tool for specifying languages but is not so much used for the description of concurrent features. While we might use powerdomains (see section 6) for the denotation, we consider for now an alternative. We introduce an operator called *oneof* which, when given a list of items, is able to return an arbitrary element from it. This operator will found our simulation: given the list of resumptions that may be taken concurrently, one will be elected and run, yielding a modified store and probably some further potential resumptions. As usual the store materializes the time parameter but

[1] The whole semantics of Scheme appears in annex of [Rees & Clinger 86].

[2] We know of no compiler doing such a thing which requires a run-time random choice with extra overhead, but such a behaviour is nonetheless legal.

also is the synchronizing element. On a multiprocessor machine, a single shared memory cannot be accessed concurrently, all reading and writing operations on the memory are then sequentialized one at a time.

The denotations will be given in Scheme but injections and projections to and from disjoint sum domains will not appear, the context simply allows to restitute them. We chose Scheme instead of more traditional greek letters since it appears that more people can read it. The semantics is complex and the redundancy[3] of Scheme makes easier to understand it. The names and the behaviours of the PolyScheme special forms are taken from Lisp: `lambda`, `if`, `prog2`, `setq` and `quote`.

The domains are the following:

v $\in$	**Val**	=	**Fun** + **Integer** + **Cons** + ...
	Fun	=	**Val*** $\times$ **Cont** $\rightarrow$ **Res**
k, kk $\in$	**Cont**	=	**Val** $\rightarrow$ **Res**
r, rr $\in$	**Env**	=	**Id** $\rightarrow$ **Loc**
th $\in$	**Res**	=	**Store** $\times$ **Res*** $\times$ **MetaCont** $\rightarrow$ **Answer**
m $\in$	**MetaCont**	=	**Store** $\times$ **Res*** $\rightarrow$ **Answer**
s, ss, sss $\in$	**Store**	=	**Loc** $\rightarrow$ **Val** + {*not-yet-computed*}
a $\in$	**Loc**	=	*the set of locations*
e $\in$	**Form**	=	*the set of valid programs*
	`meaning`	:	**Form** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

The domains of this denotation are somewhat different from the domains of the standard denotation of Scheme. The domain **Res** represents the domain of resumptions [Schmidt 86][4]. Such elements, we will also call "threads", are returned by the denotations of functions or continuations but also by the `meaning` valuation function. A resumption takes the current store, the list of all other potential resumptions (i.e. active threads) and a metacontinuation (something like a scheduler) and calls it on the resulting store and the new list of active threads. A resumption implements a computation step acting on the current state of the machine: the store and the list of active threads. When a resumption has finished its job, it invokes its metacontinuation on the resulting store and on the new list of active threads (which is generally the original one plus the thread which represents the continuation of the currently processed thread). We also add the element *not-yet-computed* to the domain **Val** to mean that a value is still under computation.

The initial store, environment, continuation and metacontinuation are:

```
(define (s.init a) ;; Loc → Val
   (wrong "No value for" a) )

(define (r.init id) ;; Id → Loc
   (wrong "No location for" id) )

(define (k.init v) ;; Val → Answer
  (lambda (s th* m) (m s th*)) )

(define (m.init s th*) ;; Store × Res* → Answer
  (if (pair? th*)
      (oneof th* (lambda (th th*th*) (th s th*th* m.init)))
      end-of-PolyScheme ) )
```

The constant `end-of-PolyScheme` occurring in the definition of `m.init` belongs to the **Answer** domain[5]. The metacontinuation `m.init` picks up a thread in the list of active threads `th*` and runs it. This metacontinuation will be varied in a following section. The exact behaviour of *oneof* is left indeterminate. *oneof* nevertheless takes a list of threads as first argument and a continuation, picks a thread from this list and invokes the continuation on the picked thread and the list but this very thread. The simplest implementation of `oneof` is:

```
(define (oneof th* q) ;; A* × ( A × A* → B) → B
  (if (pair? th*)
      (q (car th*) (cdr th*))
```

[3] The number of letters of keywords as well as the layout of Scheme programs lessen the mental load until a reasonable speed. Conversely and once mastered, the greek representation is probably easier to reread. These are only personal notations.

[4] Also see [Queinnec 90] for another example of resumptions.

[5] It is actually the sole element of **Answer**. We will vary the definition of this domain in section 6.

```
    (wrong "No possible choice") ) )
```

This order corresponds to the usual evaluation order of Lisp which is a left to right and depth-first order since new threads are added in front of the list of active threads.

The denotation is straightforward except for the application and should probably not pose problems albeit the strange interleaving of resumptions. The `meaning` function takes care of the syntactic analysis of the form to evaluate: `(define (meaning e) ;;` **Form** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (if (atom? e)
      (cond ((symbol? e)(variable-meaning e))
            ((number? e)(quote-meaning e))
            (t (wrong "not a legal term" e)) )
      (case (car e)
       (quote (quote-meaning (cadr e)))
       (if (if-meaning (cadr e) (caddr e) (cadddr e)))
       (prog2 (prog2-meaning (cadr e) (caddr e)))
       (setq (setq-meaning (cadr e) (caddr e)))
       (lambda (lambda-meaning (cadr e) (caddr e)))
       (else (application-meaning e))
        ) ) )
```

Quotation returns its `data` to its continuation `k` yielding a new thread which is added to the list of active threads. The store is left unmodified. `(define (quote-meaning data) ;;` **Val** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (lambda (r k)
    (lambda (s th* m)
      (m s (cons (k data) th*)) ) ) )
```

Variable reference is similar except that the value of the variable is looked for in the lexical environment `r`. `(define (variable-meaning id) ;;` **Id** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (lambda (r k)
    (lambda (s th* m)
      (m s (cons (k (s (r id))) th*)) ) ) )
```

Alternatives are just as simple as they should be: `(define (if-meaning bool form1 form2) ;;` **Form** $\times$ **Form** $\times$ **Form** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (lambda (r k)
    ((meaning bool)
     r
     (lambda (v)
       ((if v (meaning form1) (meaning form2)) r k) ) ) ) )
```

Assignment in regular Scheme yields an unspecified value. We choose to return the old content of the location associated to the identifier `id`. Note that the updating of the involved location is atomic. This property will be used below since `setq` acts now as a primitive "exchange" instruction [Perrott 87, p 37]. `(define (setq-meaning id form) ;;` **Id** $\times$ **Form** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (lambda (r k)
    ((meaning form)
     r
     (lambda (v)
       (lambda (s th* m)
         (m (extend s (r id) v)
            (cons (k (s (r id))) th*) ) ) ) ) ) )

(define (extend fn pt im) ;;( A → B) × A × B → ( A → B)
  (lambda (x) (if (equal? pt x) im (fn x))) )
```

Sequentiality might have been avoided as in Scheme since it may be simulated by abtractions. We nevertheless introduce `prog2` which computes sequentially its two parameters. `prog2` may easily be extended to `progn`. `(define (prog2-meaning form1 form2) ;;` **Form** $\times$ **Form** $\rightarrow$ **Env** $\times$ **Cont** $\rightarrow$ **Res**

```
  (lambda (r k)
    ((meaning form1)
     r
```

```
    (lambda (v)
      ((meaning form2) r k) ) ) ) ) )
```

Abstraction is standard. It takes values, extends the closed over lexical environment and yields a new thread which will compute the body of the abstraction. This thread will be added to the list of active threads. The freshly created abstraction is just returned to the continuation. Since we do not compare abstractions with `eq?`, we do not tag them with a location as done in [Rees & Clinger 86], the store is therefore left unmodified when creating an abstraction. `(define (lambda-meaning variables form) ;;` **Id*** × **Form** → **Env** × **Cont** → **Res**

```
  (lambda (r k)
    (lambda (s th* m)
      (m s (cons (k (lambda (v* kk)
                    (lambda (ss th*th* mm)
                      (if (equal? (length v*) (length variables))
                          (allocate ss (length variables)
                            (lambda (sss a*)
                              (mm (extend* sss a* v*)
                                  (cons ((meaning form)
                                         (extend* r variables a*)
                                         kk )
                                        th*th* )) ) )
                          (wrong "incorrect number of arguments") ) ) ))
                 th* )) ) ) )
```

`(define (extend* fn pts ims) ;;`(**A** → **B**) × **A*** × **B*** → (**A** → **B**)

```
  (if (pair? pts) (if (pair? ims)
                    (extend (extend* fn (cdr pts) (cdr ims))
                            (car pts) (car ims) )
                    (wrong "Too less images") )
      (if (pair? ims) (wrong "Too much points") fn) ) )
```

The most difficult part is the semantics of the application since it has to create multiple simultaneous threads, one for each term of the application. We use the utility function `allocate` to allocate `n` fresh locations in a store `s`. The fresh locations are initialized to *not-yet-computed*. The resulting store and locations are given to the third argument of `allocate`. A list of locations may be called a *frame*.`(define (application-meaning forms) ;;` **Form*** → **Env** × **Cont** → **Res**

```
  (lambda (r k)
    (lambda (s th* m)
      (allocate s (length forms)
        (lambda (ss a*)
          (m ss (add-threads forms a* r k a* th*)) ) ) ) ) )
```

We associate each application to a frame where the values of the terms of the application will be stored. When all terms are computed, the function will be applied. The application semantics enriches the list of active threads with as many threads as there are terms in the combination. Each newly created thread is given a continuation generated by the utility function `gen-cont`. When invoked this continuation will store at its right place the computed value, then will check if the frame is totally computed: if this is the case the function will be called otherwise the thread is simply withdrawn from the list of active threads.
`(define (add-threads forms locs r k a* th*) ;;` **Loc*** × **Env** × **Cont** × **Loc*** × **Res*** → **Res***

```
  (if (pair? forms)
      (cons ((meaning (car forms)) r (gen-cont (car locs) a* k))
            (add-threads (cdr forms) (cdr locs) r k a* th*) )
      th* ) )
```

`(define (gen-cont a a* k) ;;` **Loc** × **Loc*** × **Cont** → **Cont**

```
  (lambda (v)
    (lambda (s th* m)
      (let ((ss (extend s a v)))
        (if (computed? a* ss)
            (((ss (car a*)) (mapcar ss (cdr a*)) k) ss th* m)
            (m ss th*) ) ) ) ) )
```

```
(define (computed? a* s) ;; Loc* × Store → Bool
  (if (pair? a*)
      (and (not (equal? (s (car a*)) not-yet-computed))
           (computed? (cdr a*) s) )
      #!true ) )
```

The continuations given to each term slightly differ: they store the computed value in its correct place in the current frame. The rest of the continuation is common to all terms: check if the function can be applied and invokes it if it is the case.

The whole denotation exposed above determines the level of atomicity of operations. For instance, in the assignment denotation, to store a new value in the location associated to an identifier is performed in one step only. This atomicity is mainly due to the instantaneous computation of `(r id)`[6] which is clearly wrong in some implementations. Thus it seems hard to produce a canonical denotation of a concurrent Scheme leaving some freedom in the atomicity of operations. We nevertheless consider the above denotation as a good starting point to discuss the effect of continuations on concurrency.

3 Continuations

We will now show that the introduction of continuations while raising subtle questions definitely adds power to PolyScheme.

So far we can now enrich the initial environment and initial store with usual and appropriate functions such as `cons`, `car`, `print` ... We suppose to benefit from a `definitial` macro which installs a binding between a name and a value. To install that binding, the initial environment `r.init` and the initial store `s.init` are extended to become part of `r.standard` and `s.standard` with which PolyScheme is run. For instance, the two following functions are very simple and just kill a thread or the whole computation:

```
(definitial global-end
  (lambda (v* k)
    (lambda (s th* m)
      end-of-PolyScheme ) ) )
```

```
(definitial local-end ;; Fun
  (lambda (v* k)
    (lambda (s th* m)
      (m s th*) ) ) )
```

A much more interesting function is `call/cc` which may be defined as:

```
(definitial call/cc  ;; Fun
  (lambda (v* k)
    (lambda (s th* m)
      (if (equal? (length v*) 1)
          (m s (cons ((car v*)
                      (list (lambda (v*v* kk)
                              (lambda (ss th*th* mm)
                                (if (equal? (length v*v*) 1)
                                    (mm ss (cons (k (car v*v*)) th*th*))
                                    (wrong "a continuation requires one argument" ) ) ) ))
                      k⁷ )
                     th* ))
          (wrong "CALL/CC requires one functional argument") ) ) ) )
```

The denotation of `call/cc` wraps the current continuation `k` into a somewhat complex form giving to `k` the appearance of a first class object: a function. A new thread is created to return that function which when invoked on a value will create a new thread to send that value to the old caught continuation `k`. This denotation is directly inspired from the definition of `call/cc` of [Rees & Clinger 86]. A major difference with standard Scheme is that the whole computation is not stopped at all by throwing a value to a continuation. Consider for instance the following excerpt:

```
(call/cc (lambda (k)
           (list (display 1) (k 2) (display 3)) ))
```

[6] The same effect is visible in the denotation of variable.

[7] `k` can alternatively be turned into the black hole continuation: `(lambda (v) (lambda (s th* m) (m s th*)))` which kills any computation trying to yield a value without sending it explicitely to the continuation. This kind of `call/cc` makes `local-end` an user-definable function: `(setq local-end (lambda () (call/cc (lambda () 1))))`.

The final value may be returned before the numbers 1 and 3 be printed. With this behaviour for `call/cc`, termination has to be carefully defined. In standard Scheme, termination is when the initial continuation `k.init` is invoked. On the other hand we may also consider that termination is reached when no more active threads exist. We will keep the first definition which seems to stick more closely to standard Scheme. We therefore observe that returning a value may precede the end of the whole computation.

It is well known in Scheme that first class continuations with indefinite extent can be applied more than once:

```
(display (call/cc (lambda (k)
                    (list (k 1) (k 2)) )))
```

The numbers 1 and 2 will be printed (in some order) since the continuation is simultaneously called twice[8]. This feature brings us a flavor of non determinism[9]. Resumptions (or threads) are synchronized since their results are collected into the frame of an application and the body of the associated abstraction will be invoked only afterthat. The behaviour shown in the previous excerpt offers more freedom since the two continuation invokations are completely unrelated and do not have any kind of synchronization between them. They thus may be called *processes*.

This behaviour may be viewed as a multiple value feature but not in the sense of COMMON LISP[Steele 90] where numerous values are simultaneously given to the continuation. Here values are produced one at a time and concurrently given to the continuation. This feature may be used to implement Prolog-like clauses or the `results` feature of Miles [Bellot et al. 90].

We can benefit from this behaviour and write[10]:

```
(define (visit tree do)
  (if (pair? tree)
      (prog2 (visit (car tree) do)
             (visit (cdr tree) do) )
      (call/cc (lambda (return) (list (return tree)
                                      (return (do tree)) ))) ) )

(call/cc (lambda (exit)
           (prog2 (visit '(((a . b) c d) (e) . f) exit)
                  (local-end) ) ))
```

The `visit` function successively returns the various leaves of the binary tree `(((a . b) c d) (e) . f)`. It merely calls `do` on each leaf of `tree`. Note, inside `visit`, the contruction using `call/cc` which simultaneously returns a value while escaping thanks to `exit`. To return a value allows surrounding `prog2` to resume its computation and explore `cdr` subtrees. The invokation to `list` is just a trick to get its two arguments concurrently evaluated; one can alternatively use any other binary (or n-ary) functions such as `+` or `cons` which would be more adequately called `fork` or `cobegin`[11]:

```
(define (fork e1 e2)                         (define cobegin fork)
   (call/cc (lambda (run)
              (+ (run e1) (run e2)) )) )
```

The initial call to `visit` is followed by `(local-end)` to suicide the current process and to prevent the production of other values.

What can be the contract of `visit` ? We propose that `visit` applies its second argument to every leaf of a tree explored in a left to right depth-first manner. The result of the applications of `do` on every leaf is not waited for but proceeds concurrently to the finding of next leaves.

In fact, the previous version of `visit` shortly fails since it does not ensure that `do` is orderly invoked. The concurrent evaluation of `(return tree)` may trigger a faster `do` on the next leaf.

[8] One may wonder why the frame allocated for the computation of the terms of the application is not directly given to the abstraction rather than duplicated. The reason is named `call/cc`. Consider for instance:
`((lambda (x) (display x)) (call/cc (lambda (k) (list (k 1) (k 2)) )) )`
If the frame were not copied then this expression would print twice the content of an unique location ! This would result in printing two numbers out of 11, 12, 22 or 21 rather than just 12 or 21.

[9] Note that `k.init` is similar to `local-end` and kills the current thread. Since `k.init` is the toplevel continuation, `k.init` may as well print the result and start a new toplevel iteration as in [Wand & Friedman 86] therefore yielding strange interleaving of toplevel loops.

[10] This example was suggested by [Bellot et al. 90].

[11] The word `cobegin` may suggest that a `coend` may eventually exist conversely `fork`, after a well known operating system, does not suggest any further synchronization.

To correct this we can adopt a programming style using continuations and improve `visit` such that:

```
(define (visit tree do)
  (if (pair? tree)
      (prog2 (visit (car tree) do)
             (visit (cdr tree) do) )
      (call/cc (lambda (return)
                 (return (do tree return)) )) ) )
```

The function `do` is now applied on every leaf but is also given the continuation of the visiting process. As soon as `do` is given the current leaf, the visiting process can be resumed, in a coroutine manner, and reentrantly invokes `do` on the next leaf. The `visit` function thus fulfills its contract: leaves are given sequentially to `do` (which now has the burden to process them orderly if desired). For instance to display all the squares of the leaves of a tree of numbers, one can write:

```
(define (print-squares tree)
  (visit tree (lambda (leaf c)
                (fork (display (square leaf))
                      (c leaf) ) )) )
```

The square of the leaf is computed and displayed concurrently to the search of the next leaf. Changing `fork` to `prog2` would reintroduce a sequentialized tree processing.

4 A Semantical Misfeature (and its correction)

As can be seen, continuations and concurrency do not fit neatly. Moreover the previous PolyScheme denotation does not support exactly what we said and thus is semantically bugged [Gordon 75]. The continuation taken by `call/cc` differs from what is taken in regular Scheme so effects may be different. Consider for instance:

```
(cons (call/cc (lambda (k)
                 (list (k 1)
                       (big-computation (lambda () (k 2))) ) ))
      (huge-computation) )
```

The functions `big-computation` and `huge-computation` do what we expect from their names. Moreover the former will finally invoke the thunk given as its argument while the latter returns, say, 3 for value. Consider now the following scenarios:

1) When the form `(k 1)` is evaluated and since `(huge-computation)` is not finished, the value 1 is stored in the first slot of the frame associated to `cons`.

2a) When `(huge-computation)` finishes, `cons` is then called on 1 and 3.

3a) When `big-computation` invokes `(k 2)`, `cons` is applied a second time on 2 and 3 and everything is perfect.

Consider now this second scenairo:

1) is the same as before

2b) `big-computation` invokes `(k 2)`, the number 2 supersedes the number 1 already waiting in the first slot of the `cons` frame.

3b) `(huge-computation)` yields 3 then `cons` is applied on 2 and 3.

Since we want to more or less follow the sequential semantics of Scheme then two processes ought to be *always* created and the first one must not disappear even if executed too early.

In fact this behaviour may even appear in some of the previous examples: continuations may be invoked concurrently but may disappear if they return a value to an application where some other terms are not yet computed. There is thus a slight chance that:

```
((lambda (x) (display x))
 (call/cc (lambda (k) (list (k 1) (k 2)) )) )
```

only prints 1 or 2 exclusively and nothing else if the first invokation to `k` is performed before the end of the computation of the function `(lambda (x) (display x))`.

It is simple to correct this misfeature. First when starting to evaluate a combination, we allocate a frame with an extra location. Before writing a value in a location of this frame, we check if another value is already present. If it is the case then we do not update the location but record the modification still to be made in the extra location of the frame[12]. When the frame is totally computed, we apply the function (value of the first term of the original combination) on it, then, one at at time, each recorded modification is performed and the same function is applied on each resulting frame. The modifications are processed in the same order they were recorded. We thus keep the right count of continuation invokations. The new definitions are straightforward: `(define (application-meaning forms)`

```
  (lambda (r k)
    (lambda (s th* m)
      (allocate s (+ 1 (length forms))
        (lambda (ss a*)
          (let ((km (appl-cont a* k)))
            (m (extend ss (car a*) km)
               (add-threads forms (cdr a*) r km a* th*) ) ) ) ) ) ) )

(define (add-threads forms locs r km a* th*)
  (if (pair? forms)
      (cons ((meaning (car forms)) r (gen-cont (car locs) a* km))
            (add-threads (cdr forms) (cdr locs) r km a* th*) )
      th* ) )

(define (appl-cont a* k)
  (lambda (s th* m) ;;assume (computed? (cdr a*) s)
    (((s (cadr a*)) (mapcar s (cddr a*)) k) s th* m) ) )

(define (gen-cont a a* km)
  (lambda (v)
    (lambda (s th* m)
      (if (equal? (s a) not-yet-computed)
          (let ((ss (extend s a v)))
               (if (computed? (cdr a*) ss)
                   ((ss (car a*)) ss th* m)
                   (m ss th*) ) )
          (if (computed? (cdr a*) s)
              (km (extend s a v) th* m)
              (m (extend s (car a*)
                     (lambda (ss th*th* mm)
                       (km (extend ss (car a*) km)
                           th*th*
                           (lambda (sss th*th*th*)
                             (km (extend sss a v)
                                 th*th*th*
                                 mm ) ) ) ) )
                 th* ) ) ) ) ) )
```

With this new semantics a call to a continuation always creates a new process. Processes are finally killed by the initial continuation `k.init`. Processes may be waiting for a frame to be completely computed to run or they may execute without a chance that their value will serve to anything. Since some computations may become useless the garbage collector may reclaim these processes. To reclaim these processes is not innocuous since they may side-effect other useful processes but it is as if they were indefinitely deferred which is a legal behaviour since no fairness was ever introduced in the model.

[12] This location will contain the list of modifications that must be performed on the frame.

5 Programming in PolyScheme

Mutual exclusion is not out of reach of PolyScheme: a critical section protected by a boolean flag can be implemented on top of our atomic `setq`:

```
(defmacro mutex (bool critical-section)      ;;bool is free if false.
  (let ((local (gensym)) (result (gensym)))
    `(let ((,local #!true))  ;;occupied
       (while ,local (setq ,local (setq ,bool ,local))) ;exchange local and bool
       ;;evaluates the critical section
       (let ((,result ,critical-section))
          ;;only keeps the first result
          (if (setq ,local #T) (suicide))
          (setq ,bool #!false)      ;;and frees bool
          result ) ) ) )
```

The `mutex` macro cannot be used without precaution since waiting is active and thus consumes CPU time. Moreover an escape from the critical section may leave the protected resource `bool` in a locked state forever. Better use of `mutex` is through semaphores which enqueue wainting processes:

```
(define make-semaphore ()      ;;returns P and V
  (let ((bool #!false)         ;;free
        (queue-flag #!false)   ;;free
        (queue '()) )
    (list (lambda ()      ;;P(rolagen)
            (call/cc (lambda (k)
                       (if (setq bool #t)
                         (mutex queue-flag
                           (setq queue (cons k queue)) )
                         ;;bool was free
                         (k 'void) ) )) )
          (lambda ()      ;; V(erhogen)
            ((mutex queue-flag
              (if (null? queue) (lambda (void) (setq bool #!false))
                  (let ((proc (car queue))
                    (setq queue (cdr queue))
                    proc ) ) ) )
              'void ) ) ) ) )
```

The `make-semaphore` function returns a list of two functions acting as the classical Dijkstra's P and V. Of course other interfaces are possible.

A classical problem is to program a parallel `or`. Ours takes two expressions and evaluates them concurrently. If either of these expressions return a non-false value then `or` will return this value. The other computation will suicide itself as soon as it completes. The parallel `or` is a macro:

```
(defmacro //or (e1 e2)
  (let ((return (gensym)) (already-done (gensym)) (bool (gensym)))
    `(call/cc (lambda (,return)
                (let ((,bool #!false)
                      (,already-done #!false)
                  ((lambda (value)
                     (prog2 (mutex ,bool
                              (if ,already-done
                                (local-end)   ;;suicide
                                (setq ,already-done #!true) ) )
                            (,return value) ) )
                   (cobegin ,e1 ,e2) ) ) )) ) )
```

It is not possible in PolyScheme to kill a computation once it started so we can only ask useless processes to suicide themselves.

Many other constructs may be written in PolyScheme such as spawning a detached process or calling a function with arguments computed sequentially. These are left as exercices to the interested reader.

6 Power Domains

Concurrency may also be formalized thanks to powerdomains. The goal of this section is to show that while the previous denotation may be read from a rather operational point of view (`oneof` implementing the scheduler policy), alternate definitions may change the denotation to use powerdomains.

The architecture of the denotation makes easy to let powerdomains appear. The **Answer** domain will be changed to: **Answer** $=$ $\mathbb{P}(\ \mathbf{Val}^*)$ That is, answers are sets of sequences of values. Each path of evaluation leads to an ordered sequence of values i.e. the values on which `k.init` is applied. Since many evaluation paths exist, they may produce different sequences of values: the total answer is the set of all possible sequences of values that can be observed.

One can view **Store** $\times$ **Res*** as a state. A resumption takes a state and returns a state (given to the metacontinuation). It is simple to take a state and generates all states that can be derived by running each resumption. It is also simple to collect all the produced results. We just have to modify the previous `k.init`, `m.init` and `oneof`. We change their names to `k.pwd`, `m.pwd` and `oneof.pwd`.

The final continuation collects every result and prefixes any further answer by this very result:

```
(define (k.pwd v)
  (lambda (s th* m)
    (mapcar (lambda (v*) (cons v v*))
            (m s th*) ) ) )
```

If there is no active threads, the metacontinuation returns a set containing an empty sequence of values:

```
(define (m.pwd s th*)
  (if (pair? th*)
      (oneof.pwd th* (lambda (th th*th*) (th s th*th* m.pwd)))
      (list (list)) ) )
```

Instead of choosing an arbitrary thread, the new version of `oneof` chooses every active thread and returns the union of their associated answers (the union is represented with `append`):

```
(define (oneof.pwd th* q)
  (let loop ((th* th*)
             (oth* (list))
             (v** (list)) )
    (if (pair? th*)
        (loop (cdr th*)
              (cons (car th*) oth*)
              (let ((v**v** (q (car th*) (append oth* (cdr th*)))))
                   (append v**v** v**) ) )
        v** ) ) )
```

With these new definitions, a call to PolyScheme returns the list of possible sequences of values. For instance, the simple form `(cons 'a 'b)` leads to the `(a . b)` dotted pair by not less than 90 different ways[13].

Our point was to show that the architecture of the PolyScheme denotation, with metacontinuation and `oneof` operator, allows to express in a programmatic way the usual scheduling of threads but still does not forbid a more traditional presentation with powerdomains.

7 Related Works

At this point it seems interesting to compare Polyscheme with other concurrent languages such as Multilisp [Halstead 85] and Qlisp [Gabriel & McCarthy 84, Goldman & Gabriel 88]. The first obvious difference is that PolyScheme only exists thanks to its (fortunately executable) semantics while the two others have been implemented on realistic multiprocessors. The three languages allow side-effects and use a shared memory model. Contrarily to PolyScheme, Multilisp and Qlisp have explicit constructs for parallel execution. A new construct is advocated in [Hieb & Dybvig 90] which will be hereafter discussed.

[13] The first form offering multiple results looks like `(call/cc (lambda (k)(cons (k 'foo) (k 'bar))))`. This form offers at least 10^9 ways to be computed ! The PolyScheme powerdomain denotation is therefore obviously only of theoretical interest.

Multilisp offers the `future` concept. The form (`future` e) spawns a new process which computes the value of the expression e. The `future` construct immediately returns a promise that when accessed will block the caller until the spawned expression is fully computed. To access a future is transparent and does not require an explicit construct. This behaviour is different from the `delay-force` mechanism of regular Scheme which allows lazy computations but needs values to be explicitely `force`-d to be accessed. Atomicity can be controlled in Multilisp thanks to a primitive test-and-set called `replace`. The form (`replace` *location value*) guarantees that *location* is atomically written with *value* and the old content of *location* is returned as value of the whole form. Many other constructs (i.e. macros) are offered upon these more basic constructs.

In a sense PolyScheme nearly makes a future of every expression except that the semantics of its special forms and combination explicitely waits for their realizations. The gain is that it is never required to check if the value of a variable is an unrealized future. The loss is that less parallelism is encountered in PolyScheme due to the fork-join semantics of combination[14]. On the other hand Multilisp is sequential by default: it is thus easier to write programs without race conditions. A less general test and set mechanism is offered in PolyScheme thanks to the atomic semantics of `setq`.

The `qlet` construct of Qlisp allows to concurrently compute expressions, `qlet` can even produce futures if required thanks to the `eager` keyword. Qlisp also offers a concept of monitor [Hoare 74] with `qlambda`. `qlambda` returns a "process closure", that can only be called one at a time thus providing a "region" (the body of the `qlambda` function) with mutual exclusion. Moreover Qlisp has an interesting feature based on `catch` and `throw`, the dynamic escape special forms: whenever a `throw` is evaluated, all concurrent processes which were created in the lifetime of the associated `catch` are killed. This allows to explicitely control concurrent processes. Details may be found in [Gabriel & McCarthy 84]. PolyScheme nearly offers the same power through less means, for example critical sections were presented in section 5. The main exception is the process control through `catch` and `throw` which cannot be achieved in PolyScheme. Conversely a problem of Qlisp, mentionned by Halstead [Halstead 85], is its unclear semantics: a default which is obviously corrected in PolyScheme.

Hieb and Dybvig propose a `spawn` primitive which allows to control *process continuation* i.e. tree-structured continuations. The main difference with our work is that we try to extend in a natural way the semantics of `call/cc` while they propose another construct. But `spawn` offers a quality that is worth the price since it allows to suspend or resume a whole bunch of concurrent computations gathered in a process. `spawn` thus extends the Qlisp `catch` and `throw` mechanism. On the other hand, no concurrent operator nor other synchronization primitive is offered.

PolyScheme offers means to build most of the high level constructs of Multilisp and Qlisp and still retains a high observed level of concurrency. PolyScheme is very simple and as such seems impractical for realistic jobs; for instance, once started a computation cannot be affected. The formalism used for the semantics of PolyScheme can also be used to describe Multilisp which semantics is just a little more complex.

8 Conclusions

We presented the denotation of a Scheme-like language which major aspect is to allow concurrency. The denotation uses resumptions and clearly describes a shared memory model. The resulting language inherits from Scheme all its feature except that — combination computes its terms concurrently and — assignment is an atomic operation. No specialized construct is needed to deal with other aspects of concurrency but the major concepts of concurrency as expressed in Multilisp or Qlisp are offered.

PolyScheme also inherits the mighty `call/cc` function which can be denoted but also offers some curious effects. The current continuation is abandonned, the transfer of control is done but concurrent computations continue, some of which are useful while others are needless but still, may side-effect the future computation. We therefore described variant semantics for `call/cc` and show their difference on some examples.

[14] Simulations show that the average number of active threads is near 60 when computing (fibonacci 10) with the usual definition. Less concurrency still leads to a huge number of threads !

Our work uses a `oneof` choice operator along with metacontinuations. These are concepts close from reality: metacontinuations represent schedulers while `oneof` mimics one scheduling policy. We argue that the resulting denotational semantics is easier to write and understand although equivalent to a formulation using powerdomains.

References

[Bellot et al. 90] Patrick Bellot, Véronique Jay, Rémy Legrand, Eric Perottet, *A new step toward the integration of Logic and Functions* Actes des Journées Francophones des Langages Applicatifs (JFLA'90), (Cointe, Gautron & Queinnec eds.), La Rochelle, France, Janvier 1990.

[Birell 89] Andrew D. Birell, *An Introduction to Programming with Threads*, DEC SRC Report 35, January 6, 1989.

[Gabriel & McCarthy 84] Richard P. Gabriel, John McCarthy, *Queue-based Multi-processing Lisp*, 1984 ACM Conference on Lisp and Functional Programming, pp 9-17, Austin, Texas.

[Goldman & Gabriel 88] Ron Goldman, Richard P. Gabriel, *Preliminary Results with the Initial Implementation of Qlisp*, 1988 ACM Conference on Lisp and Functional Programming, pp 143-152, Snowbird, Utah.

[Gordon 75] Michael J. C. Gordon, *Operational Reasoning and Denotational Semantics*, Actes du Colloque IRIA "Constructions et Justifications de Programmes", G.Huet & G.Kahn (eds.), Arc et Senans Juillet 1975, pp 83–98.

[Halstead 85] Robert H. Halstead Jr., *Multilisp: A language for concurrent symbolic computation*, ACM TOPLAS, Volume 7, Number 4, October 1985, pp 501-538.

[Hieb & Dybvig 90] Robert Hieb and R. Kent Dybvig, *Continuations and Concurrency*, Second Symposium on Principles and Practice of Parallel Programming, Sigplan Notices vol 25, # 3, March 1990.

[Hoare 74] C. A. R. Hoare, *Monitors: An operating system structuring concept*, Comm ACM, volume 17, number 10, October 1974, pp 549-557.

[Perrott 87] R. H. Perrott, *Parallel Programming*, Addison-Wesley 1987.

[Queinnec 90] Christian Queinnec, *Struggle: The First Denotational Game*, EuroPAL'90, Cambridge, UK, March 1990.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[Schmidt 86] David A. Schmidt, *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Inc., Newton, Mass., 1986.

[Steele 90] Guy L. Steele, Jr., COMMON LISP*: The Language*, Digital Press (Bedford, Massachusetts), 1990.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[Wand 80] Mitchell Wand, *Continuation-based Multiprocessing*, Conference Record of the 1980 LISP Conference.

[Wand & Friedman 86] Mitchell Wand, Daniel Friedman, *The Mystery of the Tower revealed: A non reflective Description of the Reflective Tower*, Lisp and Symbolic Computation, Volume 1, Number 1, June 1988.