A Dynamic Extent Control Operator for Partial Continuations *

POPL '91, pp 174–184.

Christian Queinnec[†] Bernard Serpette[‡]
École Polytechnique & INRIA–Rocquencourt

Abstract: A partial continuation is a prefix of the computation that remains to be done. We propose in this paper a new operator which precisely controls which prefix is to be abstracted into a partial continuation. This operator is strongly related to the notion of dynamic extent which we denotationally characterize. Some programming examples are commented and we also show how to express previously proposed control operators. A suggested implementation is eventually discussed.

Keywords: continuation, partial continuation, dynamic and indefinite extent, escape feature.

Continuations were introduced within denotational semantics to express the "rest of the computation" in these cases where some constructs of a language can alter it. Non local exits or jumps (`stop`, `goto`), exception handling, failure semantics in Prolog-like languages are usually described with continuations [Stoy 77, Schmidt 86]. The Scheme language [Rees & Clinger 86] offers procedural first-class continuations with indefinite extent. Like functions (lexical closures) which reinstall the environment of variables that was active when they were defined, continuations reinstall the rest of the computation which remains to be done when they were defined. Continuations have been proved to be valuable tools in Scheme where they are used to program non local exits [Haynes & Friedman 87b], multitasking [Wand 80], engines [Haynes & Friedman 87a] etc.

The problem of continuations is that they are too powerful [Haynes & Friedman 87b] since they reify [Friedman & Wand 84] the *whole* rest of the computation. To call one of them means losing control since they never return. A partial continuation [Johnson 87] is only a prefix of the computation that remains to be done. A partial continuation is thus a function that returns to its caller therefore partial continuations are composable like regular functions. Yet full continuations are still useful since they allow imperative (abortive) transfer control.

Programming languages widely differ with their offer of continuations. Continuations can be first-class or not, procedural or not, be accessed from a different namespace or from the regular lexical variable environment. Continuations can have a dynamic or indefinite extent, they can behave lexically or dynamically. Moreover two main uses of continuations can be recognized. First, continuations are a non-local exit facility and can be met in COMMON LISP as `block/return-from` or `catch/throw`, in Modula-3 as `try-except/raise` and in C as `setjmp/longjmp`. These constructs allow to escape from a computation and to return to a previous control point. They are often used to handle exceptional situations. Since these continuations can only be used while in their dynamic extent, their implementation is therefore very efficient.

The second use of continuations is particular to Scheme and is closely related to their indefinite extent. A computation may be reified (frozen) into a continuation, passed along as a normal value and finally unfrozen later on. This allows to easily implement a non preemptive multitasking facility [Wand 80]. But the most intriguing feature that makes them differ from threads (or lightweight processes) is that continuations are immutable objects that may be resumed more than once. From a naïve implementation side, reifying a continuation roughly corresponds to save the current evaluation stack[1]; resuming a continuation reinstalls the

*This work has been partially funded by GDR Programmation.

†LIX, École Polytechnique, 91128 Palaiseau Cedex, France, Email address: `queinnec@poly.polytechnique.fr`

‡INRIA–Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France, Email address: `serpette@inria.fr`

0

[1] This is straightforward for heap-based implementation of the

saved stack overwriting the current stack. The evaluation stack represents the above mentioned "rest of the computation" therefore, to copy back a saved stack implies that these continuations are not composable since there is no means to return to an overwritten stack.

Partial continuations are usually based on a couple of features: one which marks the beginning of continuations that might be abstracted later on and another feature which reifies them. Several models of partial continuations exist: mainly [Felleisen & Wand & Friedman & Duba 88] with `run/control` and [Danvy & Filinski 90] with `reset/shift`. These constructs do not permit to easily pair related control operators i.e. to precisely specify up to which context partial continuations must be reified.

The paper therefore presents, in the framework of Scheme, a new control operator named `splitter` (cf. section 2) which creates pairs of associated control operators: one is a non-local exit facility whilst the other reifies partial continuations upto the point where `splitter` was called. Various dressings for `splitter` are compared in section 4 as well as how they are typed.

The `splitter` construct reconciles the two previously recognized uses of continuations: dynamic extent full continuations and indefinite extent partial continuations. The basic idea is that a classical evaluation stack (or a list of linked activation records) may be marked in order to be later on split into two parts: the lower part, under the mark, represents the rest of the computation that can be escaped into. The upper part, above the mark, may be abstracted into a partial continuation and therefore multiply applied, stored etc. A partial continuation abstracts the part of the control between a mark and the point where it is reified. Therefore partial continuations can only be taken upto a mark which must exist i.e. be in the dynamic extent of the progenitor `splitter`, cf. section 1.

Eventually the paper suggests an implementation technique inspired from, at least, [Danvy 87] and [Hieb & Dybvig & Bruggeman 90] that avoids copying activation records from and back to stack (cf. section 5). A partial continuation freezes its associated activation records. Popping frames from the stack is non-destructive since it is only a pointer translation. Conversely pushing new frames may overwrite frozen frames. An extra pointer (`free-stack`) exists above which it is always possible to push new frames. To push frames in a regular zone is done as usual i.e. relatively to the regular stack pointer; conversely to push frames in a frozen zone is done relatively to `free-stack` with the necessary (and then explicit) control links. This technique allows allocating mutable locations in the stack since these locations are not duplicated.

evaluation stack i.e. a linked list of activation records; it usually involves some copy for stack-based implementation [Clinger 88, Hieb & Dybvig & Bruggeman 90].

1 Semantical Framework

This section presents much of the semantics of a Scheme-like language and formalizes our conception of dynamic extent, see also [Steele 90, chapter 3].

The term *extent* refers to a period of time: the lifetime of an entity[2]. Any entity of Scheme has an indefinite extent i.e. entities exist forever. In most languages (Scheme excepted) applications have a dynamic extent. The extent of the application of a function on its arguments is the time during which is computed the body of the function, this includes the time taken by the computation of all subforms that appear in this body. In that sense, dynamic extents are always nested or disjoint. When the language offers non-local exits, the dynamic extent of an application might be interrupted before its natural end which is when the function returns a value to its continuation. To finish a dynamic extent forces the end of all inner-nested dynamic extents.

Dynamic extent is strongly related to the height of the evaluation stack and governs the possibility of stack-allocating various entities such as lexical environments. It is a common compiling optimization to determine (or conservatively approximate) the exact extent that entities have and to allocate them accordingly.

$$
\begin{array}{lll}
\pi & \mathbf{Prog} & = \textit{The set of forms} \\
\nu & \mathbf{Id} & = \textit{The set of identifiers} \\
\rho & \mathbf{Env} & = \mathbf{Id} \rightarrow \mathbf{Loc} \\
\alpha & \mathbf{Loc} & = \textit{The set of locations} \\
\sigma & \mathbf{Store} & = \mathbf{Loc} \rightarrow \mathbf{Val} \\
\varepsilon & \mathbf{Val} & = \mathbf{Fun} + \mathbf{Pair} + \mathbf{Num} + \ldots \\
\varphi & \mathbf{Fun} & = \mathbf{Val}^* \times \mathbf{Store} \times \mathbf{Ext} \times \mathbf{Cont} \rightarrow \mathbf{PAns} \\
\kappa & \mathbf{Cont} & = \mathbf{Store} \times \mathbf{Val} \times \mathbf{Ext} \rightarrow \mathbf{PAns} \\
\zeta & \mathbf{Ext} & = \mathbf{Loc} \rightarrow \mathbf{Bool}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E} = \mathbf{Prog} - \\
\mathcal{E}^* = \mathbf{Prog}^*
\end{array}
$$

Figure 1: Domains of Scheme with explicit dynamic extent

When continuations have an indefinite extent as in Scheme, it is possible that an expression multiply return results. In that case the concept of applications having a dynamic extent must be precisely defined. We therefore propose a denotational semantics for Scheme formally expressing our notion of dynamic extent. The main question concerning an entity with such an extent is whether it is alive or not ? We thus add the domain **Ext** to the standard denotation of Scheme [Rees & Clinger 86] mapping active locations to the boolean true, see figure 1. Initially no location is active. Like the store, this map is passed to functions and continuations. Locations were chosen since they can be compared and thus distinguished. The **PAns** domain

[2] First- or second-class objects form entities.

is the domain of partial answers yielded by regular (`splitter`-free) computations. This domain will be explicited when describing the semantics of `splitter`. Except for **PAns** which stands as if it is the domain of the final answers and **Ext** which is not yet used, the denotations are fairly standard and should not pose problems, see figure 2.

2 The `splitter` operator

The Scheme philosophy tries to reduce the number of special forms (even if introducing very special functions like `call/cc`), sticks to a single namespace and makes nearly all concepts first-class citizens. *Le fin du fin* is to represent new concepts procedurally. We thus express our solution with respect to these tenets: a single function, named `splitter`, is necessary:

```
(splitter (lambda (abort call/pc) expression)))
    with   (abort (lambda () ...))
    and    (call/pc (lambda (c) ...))
                  and    (c expression)
```

When `splitter` is invoked, the evaluation stack is marked. The argument of `splitter` is a binary function which is then applied on two new synthesized functions tied to this mark. These two functions can only be safely invoked during the dynamic extent of `splitter` otherwise they provoke a run-time error. If neither *abort* nor *call/pc* is used, `splitter` returns what returns its argument.

```
(splitter (lambda (abort call/pc)
              'foo )) → foo
```

The first function *abort* takes a single argument, a thunk, and invokes it with the continuation of `splitter` as continuation. This allows to abandon a computation and to return to the level of the mark set by the progenitor `splitter`. For instance, to multiply the elements of a list of numbers and exit if a zero is found, may be written as:

```
(define (multlist l)
  (splitter
   (lambda (abort call/pc)
     (define (mult l)
       (if (pair? l)
           (if (= (car l) 0)
               (abort (lambda () 0))
               (* (car l) (mult (cdr l))) )
           1 ) )
     (mult l) )) )
```

When `abort` is invoked, all waiting computations upto `splitter` are discarded and the thunk `(lambda () 0)` is invoked as if it replaces the original `splitter` form.

The second function *call/pc*[3] takes an unary function as single argument. It then reifies the partial continuation upto its parent `splitter` and invokes its argument on it. When reified, the partial continuation is left in place i.e. is still the current continuation. For instance,

```
(splitter
  (lambda (abort call/pc)
    (cons (call/pc (lambda (c)
                     ;c = (λ(x) (cons x 'a))
                     (cons 'b (c (c 'd))) ))
          'a ) ))
```

This form returns `((b . ((d . a) . a)) . a)`.

As regular objects, *abort* and *call/pc* have an indefinite extent but since they remove or reify the partial continuation upto the associated parent `splitter`, they must be invoked in the dynamic extent of `splitter`: their safe behavior has a dynamic extent. Contrarily the reified partial continuation has an indefinite extent behavior; once created, it can be used forever:

```
((cdr
  (splitter ;returns (a . partial-cont)
   (lambda (abort1 call/pc1)
     (cons
      'a
      (splitter
       (lambda (abort2 call/pc2)
         (cons
          'b
          (call/pc1
           (lambda (c)
             ;c = (λ(x) (cons 'a (cons 'b x)))
             (abort2 (lambda () c))
            ) ) ) ) ) ) ) ))
 'c )   → (a b . c)
```

Since partial continuations have an indefinite extent behavior, `splitter` can be used to simulate the `call/cc` operator of Scheme. A toplevel *expression* of regular Scheme is equivalent[4] to the following:

```
(splitter
 (lambda (abort0 call/pc)
   (let ([call/cc
          (lambda (f)
            (call/pc
             (lambda (c)
               (f (lambda (v)
                    (abort0
                     (lambda ()
                       (c v) )) )) )) )])
     expression ) ))
```

There is no restriction on *expression* which can arbitrarily invokes `call/cc` or `splitter` without interference.

[3] `call/pc` stands for "call with partial continuation".

[4] A slight difference might exist if one uses a toplevel loop since a new `call/cc` is synthesized every cycle.

$\mathcal{E}[\![\nu]\!] = \lambda\rho\sigma\zeta\kappa.\kappa(\sigma, \sigma(\rho(\nu)), \zeta)$

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi'\ \pi''\texttt{)}]\!] = \lambda\rho\sigma\zeta\kappa.\ \textbf{let}\ \kappa' = \lambda\sigma'\varepsilon\zeta'.\ \textbf{if}\ \varepsilon\ \textbf{then}\ \mathcal{E}[\![\pi']\!](\rho, \sigma', \zeta', \kappa)\ \textbf{else}\ \mathcal{E}[\![\pi'']\!](\rho, \sigma', \zeta', \kappa)\ \textbf{endif}\\ \textbf{in}\ \mathcal{E}[\![\pi]\!](\rho, \sigma, \zeta, \kappa')$$

$$\mathcal{E}[\![\texttt{(set! } \nu\ \pi\texttt{)}]\!] = \lambda\rho\sigma\zeta\kappa.\ \textbf{let}\ \kappa' = \lambda\sigma'\varepsilon\zeta'.\kappa(\sigma'[\rho(\nu) \to \varepsilon], \varepsilon, \zeta')\ \textbf{in}\ \mathcal{E}[\![\pi]\!](\rho, \sigma, \zeta, \kappa')$$

$allocate : \textbf{Store} \times\ \textbf{Num} \times\ (\textbf{Store} \times\ \textbf{Loc}^* \to\ \tau) \to\ \tau$ /* *allocates* num *locations in* store. */

$$\mathcal{E}[\![\texttt{(lambda } \nu^*\ \pi\texttt{)}]\!] =\\ \lambda\rho\sigma\zeta\kappa.\ \textbf{let}\ \varphi = \lambda\varepsilon^*\sigma'\zeta'\kappa'.allocate(\sigma', \#\nu^*, \lambda\sigma''\alpha^*.\mathcal{E}[\![\pi]\!](\rho[\nu^* \xrightarrow{*} \alpha^*], \sigma''[\alpha^* \xrightarrow{*} \varepsilon^*], \zeta', \kappa'))\\ \textbf{in}\ \kappa(\sigma, \varphi, \zeta)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!] = \lambda\rho\sigma\zeta\kappa.\ \textbf{let}\ \kappa' = \lambda\sigma'\varphi\zeta'.\ \textbf{let}\ \kappa' = \lambda\sigma''\varepsilon^*\zeta''.\varphi(\varepsilon^*, \sigma'', \zeta'', \kappa)\\ \textbf{in}\ \mathcal{E}^*[\![\pi^*]\!](\rho, \sigma', \zeta', \kappa')\\ \textbf{in}\ \mathcal{E}[\![\pi]\!](\rho, \sigma, \zeta, \kappa')$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!] = \lambda\rho\sigma\zeta\kappa.\ \textbf{let}\ \kappa' = \lambda\sigma'\varepsilon\zeta'.\ \textbf{let}\ \kappa' = \lambda\sigma''\varepsilon^*\zeta''.\kappa(\sigma'', <\varepsilon> \S\varepsilon^*, \zeta'')\\ \textbf{in}\ \mathcal{E}^*[\![\pi^*]\!](\rho, \sigma', \zeta', \kappa')\\ \textbf{in}\ \mathcal{E}[\![\pi]\!](\rho, \sigma, \zeta, \kappa')$$

$\mathcal{E}^*[\![]\!] = \lambda\rho\sigma\zeta\kappa.\kappa(\sigma, <>, \zeta)$

$\rho_{init}(\nu) = wrong($“`Undefined variable`”$, \nu)$

$\sigma_{init}(\alpha) = wrong($“`Unknown location`”$, \alpha)$

Figure 2: Semantics of main Scheme special forms

The `splitter` function separates two effects i.e. reifying the partial continuation and removing it from the current continuation. The *abort* function is a kind of “tail-recursivor” which takes a thunk and invokes it as if it was in tail position with respect to the associated `splitter`. *abort* clearly involves a side-effect on control. It closely corresponds to how exceptions are sometimes handled: the erroneous computations is escaped and the appropriate handler is invoked at the level where this handler was bound to that exception. On the other hand *call/pc* does not affect control. To express the formal semantics[5] of `splitter`, see figure 3, we define the **PAns** domain of partial answers to be:

$$\textbf{PAns} = \textbf{Cont}^* \to \textbf{Ans}$$

where **Ans** is the domain of final answers and $\textbf{Cont}^*$ is the domain of sequences of continuations. In a computation $(\mathcal{E}[\![\pi]\!]\rho\sigma\zeta\kappa)\kappa^*$, κ and κ^* form together the regular full continuation.

The `splitter` operator first extends the set of active objects with a fresh location yielding ζ'. It then creates *abort* and *call/pc* (φ and φ') and invokes its argument on them. The new continuation is κ_{return} while the former κ is “pushed” onto κ^*. When a value is sent to κ_{return}, κ_{return} will consider the current sequence of continuations and send the value to the first of them.

The *abort* function just invokes its argument at the level of `splitter`: it resets the set of active objects and the continuation to these of `splitter`. Contrarily *call/pc* does not affect the set of active objects. Observe that *last* and *butlast* access κ^* from the tail whereas other models count contexts from the head.

[5] A simulation of `splitter` written in regular Scheme appears in appendix.

3 Generators

Our `splitter` operator can be put to work on the well-known `same-fringe` problem. Two or more binary trees are compared: `same-fringe` returns true if they all have a similar fringe i.e. the same sequence of leaves. Classical solutions involve only two trees and explicitly interleaves two coroutines enumerating sequentially the leaves of the two trees. Non-classical solutions can also be found for example in [Gabriel 89]. Our solution satisfies two goals (i) it handles an arbitrary number of trees and (ii) it is independent of how the tree is walked through.

Visiting the leafs of a tree, in prefix order, can be done thanks to:

$\zeta_{init}(\alpha) = \mathit{false}$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{splitter}]\!] = \lambda\varepsilon^*\sigma\zeta\kappa.\lambda\kappa^*.\\
\quad \mathit{allocate}(\sigma, 1, \lambda\sigma'\alpha^*.\\
\qquad \mathbf{let}\ \zeta' = \zeta[\alpha^*\downarrow_1 \rightarrow \mathit{true}]\\
\qquad\qquad \kappa' = \lambda\sigma''\varepsilon\zeta'.\lambda\kappa^{*\prime}.\ \mathbf{if}\ \#\kappa^{*\prime} = \#\kappa^*\ \mathbf{then}\ \kappa(\sigma'', \varepsilon, \zeta)(\kappa^{*\prime})\ \mathbf{else}\ \kappa(\sigma'', \varepsilon, \zeta')(\kappa^{*\prime})\ \mathbf{endif}\\
\qquad \mathbf{in}\ \ \mathbf{let}\ \varphi = \lambda\varepsilon^{*\prime}\sigma''\zeta''\kappa''.\ \mathbf{if}\ \zeta''(\alpha^*\downarrow_1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \lambda\kappa^{*\prime}.\varepsilon^{*\prime}\downarrow_1(<>, \sigma'', \zeta', \kappa_{return})(\mathit{last}(\kappa^{*\prime}, 1 + \#\kappa^*))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{wrong}(\text{“}\texttt{obsolete escape}\text{”})\ \mathbf{endif}\\
\qquad\qquad\quad \varphi' = \lambda\varepsilon^{*\prime}\sigma''\zeta''\kappa''.\ \mathbf{if}\ \zeta''(\alpha^*\downarrow_1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \lambda\kappa^{*\prime}.\ \mathbf{let}\ \kappa^{*\prime\prime} = \mathit{butlast}(<\kappa''>\S\kappa^{*\prime}, 1 + \#\kappa^*)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{let}\ \varphi = \lambda\varepsilon^{*\prime\prime}\sigma'''\zeta'''\kappa'''.\lambda\kappa^{*\prime\prime\prime}.\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa_{return}(\sigma''', \varepsilon^{*\prime\prime}\downarrow_1, \zeta''')(\kappa^{*\prime\prime}\S<\kappa'''>\S\kappa^{*\prime\prime\prime})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \varepsilon^{*\prime}\downarrow_1(<\varphi>, \sigma'', \zeta'', \kappa'')(\kappa^{*\prime})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{wrong}(\text{“}\texttt{out of extent}\text{”})\ \mathbf{endif}\\
\qquad\qquad \mathbf{in}\ \varepsilon^*\downarrow_1(<\varphi, \varphi'>, \sigma', \zeta', \kappa_{return})(<\kappa'>\S\kappa^*))
\end{array}
$$

$\kappa_{return}(\sigma, \varepsilon, \zeta) = \lambda\kappa^*.\kappa^*\downarrow_1(\sigma, \varepsilon, \zeta)(\kappa^*\dagger 1)$

$\mathit{last}(\kappa^*, \iota)$ /* *returns the last* ι *elements of* κ^* */

$\mathit{butlast}(\kappa^*, \iota)$ /* *returns all but the last* ι *elements of* κ^* */

Figure 3: Semantics of `splitter`

```
(define (visit tree fn)
  (if (pair? tree)
      (begin (visit (car tree) fn)
             (visit (cdr tree) fn) )
      (fn tree) ) )
```

We now abstract over this tree walk. For each tree the comparing process must receive the leaf and the partial continuation expressing the rest of the visit (and not the rest of the computation). If they are all equal the comparison goes on otherwise it is aborted. We thus wrap `visit` into a new function which allows to return at the same time the leaf and the rest of the visit.

```
(define (make-tree-walker visit)
  (lambda (tree)
    (splitter
     (lambda (exit grab)
       (visit
        tree
        (lambda (leaf)
          (grab
           (lambda (c)
             (exit (lambda ()
                     (cons
                      leaf
                      (lambda (v)
                        (splitter
                         (lambda (k j)
                           (set! grab j)
                           (set! exit k)
                           (c v) )) ) )
                   )) )) ) ) ) ) ) )
```

When a leaf is found, a pair made of this leaf and the partial continuation is returned. The `splitter` controlling this partial continuation is thus no longer valid and cannot be reused. The partial continuation which is returned is thus wrapped inside a new invocation of `splitter`, the new `grab` and `exit` functions are also updated with respect to this new `splitter`. It will therefore be possible to make a partial continuation of the restarted partial continuation.

We can now simply compare fringes i.e. sequences of pairs (leaf . continuation) as enumerated by a particular tree-walker:

```
(define (compare-fringes walk trees)
  (let ((end (list 'end)))  ;sentinel
    (define (end? leaf) (eq? leaf end))
    (define (loop leafs)
      (define (same-leaf? leaf)
        (eq? (car leaf) (caar leafs)) )
      (or ;all trees are finished ?
          (every? end? leafs)
          ;some trees are finished ?
          (if (some? end? leafs) #f
```

```
            (and (every? same-leaf?
                         (cdr leafs) )
                 ;all leaves are equal !
                 (loop (mapcar
                        (lambda (leaf)
                          ((cdr leaf) end) )
                        leafs )) ) ) ) )
    (loop (mapcar walk trees)) ) )
```

Finally same-fringe is just:

```
(define (same-fringe . trees)
  (compare-fringes (make-tree-walker visit)
                   trees ) )
```

Observe the modularity of this solution. The visit function is expressed in direct style and can be varied, for instance to visit every other leaf or numeric leaves only. This visit function does not have to know how it is used; in particular it does not bear the burden of interleaving the various visits of the different trees. Similarly compare-fringe only handles the multiplicity of trees, compares leaves and resumes the visiting processes; it is not concerned with the details of the tree-walking. The whole burden is borne by make-tree-walker which makes computations progress step by step.

This example shows that splitter has the ability to construct complex generators not restricted to simple and reduced internal state held in some variables. Complex generators can compute arbitrary things, be saved in their current state of computation and later on resumed. Note that computations within such generators are not restricted and can even use partial continuations.

4 Variants

Other dressings can be imagined for splitter, we investigate two of them that can be put to work in Scheme or other languages.

It might be tempting to create a new type to represent marks of the evaluation stack. The interface to splitter would become:

(splitter (lambda (*mark*) ...)

Such marks can be used thanks to two new functions which only know how to handle marks: abort and call/pc:

(abort *mark* (lambda () ...))
(call/pc *mark* (lambda (*c*) ...))

The partial continuation is still functional but the mark is now a first class non-procedural object which can be asked whether it is alive or not thanks to a third specialized function:

(alive-mark? *mark*)

This variant leads to simple type equations, where (Mark τ) is the type of marks created in a context where splitter must yield a τ value.

splitter : $((\text{Mark } \tau) \rightarrow \tau) \rightarrow \tau$
abort : $(\text{Mark } \tau) \times (\text{Unit} \rightarrow \tau) \rightarrow \tau'$
call/pc : $(\text{Mark } \tau) \times ((\tau' \rightarrow \tau) \rightarrow \tau') \rightarrow \tau'$

Obviously we gain a little power since it is now possible to explicitly test if the mark is alive or not but the neat interest of this dressing is that only one object is allocated (the mark) instead of the two synthesized functions *abort* and *call/pc*. On the other hand the interface is more complex since at least one type and three more specialized functions have been added to Scheme.

Another variant is to consider that marks are no more first-class objects but must be accessed *à la* COMMON LISP[6] with ad-hoc special forms.

(splitter *label body*)
(abort *label new-body*)
(call/pc *label* (lambda (*c*) *expression*))

This introduces a new namespace, the space of *labels* which associates names to marks. This space is lexically managed: a *label* can only be used in the *body* of the splitter special form binding it. This does not restrict the power of this variant since an abort or call/pc special form referencing this *label* can be closed in a lambda abstraction and exported outside.

Using this solution makes simpler for a compiler to recognize the places where some optimizations might be performed such as the validity of a mark, static computation of partial continuations or abortion. It also makes the compiler more complex since it introduces new special forms.

Depending on the natural inclination of the embedding language, the splitter facility can be provided under multiple forms. To have first-class marks and specialized functions might be a good choice.

5 Implementation

From the viewpoint of the implementor, splitter induces the same problems as call/cc in Scheme. Above all is the management of the evaluation stack. A partial continuation is represented by a slice of the execution stack. We propose an implementation scheme where partial continuations are not copied in the heap nor copied back onto the execution stack. The technique is closely related to [Hieb & Dybvig & Bruggeman 90].

Once a partial continuation is reified, the corresponding stack slice is considered as frozen and must not be overwritten before the Garbage Collector unfreezes it.

[6] It is a simple exercise to write the structured lexical non-local exit facilities of COMMON LISP, see [Steele 90]: block and return-from with this variant of splitter.

Let us only consider push and pop as the only operations on the evaluation stack. Push allocates new activation records and increments the stack pointer (SP) while pop just decrements it. Observe that pop is non destructive. We must then forbid pushing while SP is in a frozen zone. We thus suppose to have an extra pointer (`free-stack` or, FS, for short) above which there is no frozen zone: therefore it is always safe to allocate new activation records there, see figurefigstack1. In regular mode only SP is used. When a partial continuation is created or activated, FS is set to SP. Popping activation records is always done with respect to SP. Conversely pushing a frame depends on the mode: in regular mode, pushing a frame is done relatively to SP but if SP is in a frozen slice i.e. if SP<FS then pushing must be done relatively to FS. A special frame (or return address) *R* is then pushed at FS to save the current SP. Afterthat SP is reset to FS (plus some offset) and pushing is performed as usually since we are now in a non-frozen zone. This extra work is only done once since afterthat SP is higher than FS. When returning to *R*, SP is reset to its old value somewhere in the frozen zone.

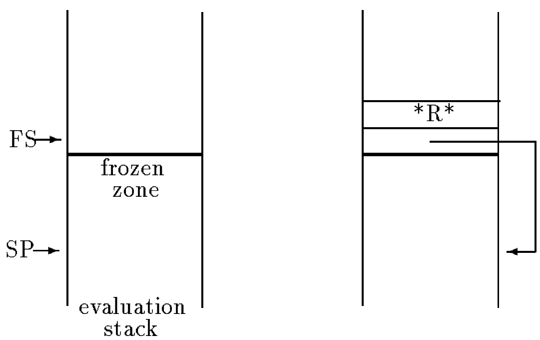

Figure 4: Calling a function from a frozen zone

An extra stack, the control stack, is necessary. Each times `splitter` is called, its continuation is pushed onto the control stack. When a partial continuation is created, the corresponding slice of the control stack is copied in the heap, see figure 5. When a partial continuation is called, the current SP is pushed onto the control stack as well as the saved slice. Then SP is set to the top of the partial continuation. Its last frame (*contPOP*) is a special activation record imposing to return where the control stack specifies it.

The benefits of this implementation are: (i) a stack-based discipline with implicit control linkage is used, (ii) copying stack slices is avoided, (iii) mutable locations can be directly put in the stack since stack slices are never duplicated. There are also inconveniences. The technique puts a fix cost on every push (two instructions) to check the current mode i.e. to compare FS and SP. When a stack slice is frozen when set-

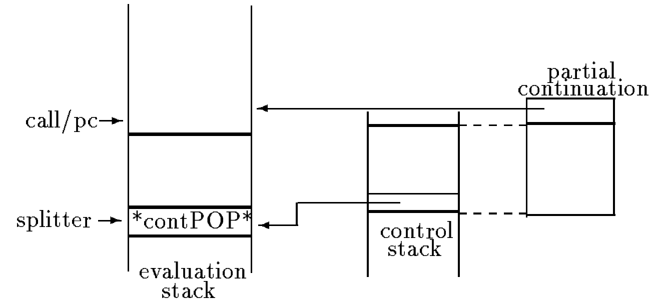

Figure 5: Partial continuation reification

ting FS, the whole bottom of the stack is frozen. The stack must now be scavenged (as in [Hieb & Dybvig & Bruggeman 90]) and this requires cooperation of the GC and a perfect knowledge of the stack. This probably precludes conservative GCs, but to compact the stack may allow some frame elimination as illustrated in [Saint-James 84, Hanson 90].

The above implementation of `splitter` is not tail-optimal: the synthesized functions *abort* and *call/pc* have to be invalidated when `splitter` returns. On the other hand, *call/pc* is tail-optimal. Yet another inconvenience is that strictly tail-recursive calls are not possible from a frozen zone since a new frame has to be pushed at FS. The tail-recursion property is immediately regained since afterthat calls are performed from a free zone.

6 Related Works

Felleisen and others have introduced prompts and partial continuations in a series of paper [Felleisen 88, Felleisen & Wand & Friedman & Duba 88, Sitaram & Felleisen 90]. Their control operators behave dynamically since `control` can only abstract control upto the most recent `run`. We can easily provide these operators:

```
(let ((run-stack '()))
  (set! run
        (lambda (thunk)
          (let ((old-rs run-stack))
            (begin0
             (splitter
              (lambda (abort call/pc)
                (define (cpc f)
                  (call/pc
                   (lambda (c)
                     (abort (lambda ()
                              (f c) )) )) )
                (set! run-stack
                      (cons cpc run-stack) )
```

```
              (thunk) ) )
           (set! run-stack old-rs) ) ) ) )
  (set! control (lambda (f)
                 ((car run-stack) f) )) )
```

The above programmation records all `call/pc`s in a stack in order for `control` to abstract upto the last `run`[7]. Our construct is not limited to the latest context; it is possible to take partial continuations upto different `splitters`. Sitaram and Felleisen [Sitaram & Felleisen 90] introduced hierarchies to solve the problem of correctly pairing `runs` and `controls` but require some protocol to be respected. Our solution solves naturally this problem.

Another major point is that our construct reintroduces dynamic extent continuations which are both useful and efficient.

The "Abstracting Control" paper [Danvy & Filinski 90] introduced two special forms acting as control operators: **`shift`** and **`reset`**. A denotational semantics accompanies them. Intuitively, a program using these special forms is translated into a regular program with explicit continuations i.e. written in an Extended Continuation Passing Style. These programs can also be directly typed thanks to complex judgements involving the natural type of the expression and of its continuation yielding another natural type for the final result, see [Danvy & Filinski 89]. The main difference with our work is that they only consider the first embedding contexts (i.e. the head of $< \kappa > \S\kappa^*$ in our terminology) whereas we count them from the tail. To change `(lambda (`ν^*`)` π`)` into `(lambda (`ν^*`) (reset` π`))` might deeply affect the meaning of a whole program. But to replace it with `(lambda (`ν^*`) (splitter (lambda (`k j`)` π`)))` does not alter it. This property eases modularity.

Hieb and Dybvig introduced a new construct `spawn` in [Hieb & Dybvig 90b]. Although close to our work there are some fundamental differences. `spawn` is defined in a concurrent context and allows to suspend or resume bunches of processes. `(spawn (lambda (`*control*`) ...))` creates a new process controlled by *control*. When *control* is invoked, it applies its argument, a monadic function, on the partial continuation up to `spawn`. Moreover it freezes all the concurrent subcomputations initiated between `spawn` and the invocation of *control*. These subcomputations will be resumed when the partial continuation is called. *control* can only be used while the process is not suspended. `spawn` can be accurately written with `splitter`. But since the suspend/resume capabilities are needless but to handle concurrency and forbid to write sequential programs calling *control* inside *control*, we therefore simplify `spawn` to the following sequential version:

```
(define (spawn f)
  (define (mk-cpc curr-cpc curr-abort)
    (lambda (c)
      (curr-cpc
       ;takes the current partial continuation
       (lambda (cc)
         (curr-abort  ;and aborts it
          (lambda ()
            (c (lambda (v)
                 (splitter
                  (lambda (na ncpc)
                    ;resets the root of the process
                    (set! curr-cpc ncpc)
                    (set! curr-abort na)
                    (cc v) )) )) ) ) ) ) ) )
  (splitter
   (lambda (abort call/pc)
     (f (mk-cpc call/pc abort)))) )
```

The first difference concerns the lifetime of *control* which is not reduced to the dynamic extent of `spawn`. Every times the partial continuation is invoked, it is legal to call *control* again. We simulated this behavior by wrapping the partial continuation itself within a new `splitter`. This eases to multiply suspend a computation upto its root (see, for instance, the `same-fringe` example) but makes more difficult to know what is the partial continuation since it is upto the last point where it has been invoked. Assume that the evaluation order of arguments is from right to left, and consider the following example where the partial continuation itself contains another call to *control*:

```
(spawn (lambda (f)
         (cons (f (lambda (c1)
                    (cons 1 (c1 4)) ))
               (f (lambda (c2)
                    (cons 2 (c2 3))
                   )) ) ))
```

The value is `(2 1 4 . 3)`. The reason is when calling `foo`, the partial continuation `c2` is `(lambda (u) (cons (f (lambda (c1) (foo c1 4))) u))`, The new partial continuation reified by the second `f` in `c1` is: `(lambda (y) (cons y 3))` and not the whole partial continuation upto `spawn` i.e.`(lambda (v) (cons 2 (cons v 3)))`. In effect in this case there are two simultaneous roots and Hieb and Dybvig restrict the partial continuation to take upto the most recent root i.e. the partial continuation is restricted upto the calling site of `f`.

The analog form using `splitter` is:

```
(splitter
 (lambda (abort call/pc)
   (cons
    (call/pc
```

[7]A dynamic binding facility would simplify the management of `run-stack`.

```
    (lambda (c1)
      (abort (lambda () (foo c1 4))) ) )
   (call/pc
    (lambda (c2)
      (abort (lambda ()
               (cons 2 (c2 3)))))) ) ))
```

The value, assuming the same order of evaluation[8], is `(1 2 4 . 3)` since the second call to `call/pc` reifies upto `splitter`.

To sum up, our construct controls more precisely extent and still offers `spawn`.

7 Conclusions

Partial continuations are new and very powerful tools. Their use is nevertheless complex. Not being able to know up to where, escapes (provoked by `abort`) or partial continuations are created (thanks to `call/pc`), makes partial continuations nearly useless except for toy programs. Our `splitter` construct allows to appropriately pair two moments: — marking the evaluation stack, — referring to the lower part of this evaluation stack (under the mark) or referring to the intermediate slice (above the mark) between this mark and the current control point. It is thus possible to write programs using simultaneously multiple marks without interferences. We showed how to use `splitter` to program generators enumerating leaves of trees as well as we showed how to rebuild `call/cc` from `splitter`.

Our view of computation recognizes that computations involves a structure of nested control points. A partial continuation is therefore taken upto an active control point. It is interesting to observe that this view supports the concept of dynamic extent but still allows to define indefinite extent partial continuations.

Acknowledgements

Thanks to Matthias Felleisen, Dorai Sitaram, Olivier Danvy, Pierre Weis and Damien Doligez for their help while drafting, revising and discussing this paper.

Bibliography

[Clinger 88] William Clinger, Anne Hartheimer, *Implementation Strategies for Continuations*, Proceedings of the 1988 ACM Lisp and Functional Programming, Snowbird, Utah, July 1988, pp 124–131.

[Danvy 87] Olivier Danvy, *Memory Allocation and Higher-Order Functions*, SIFPLAN'87 Conference, pp 241–252.

[8] Remark that the final value strongly depends on the order of evaluation. Partial continuations are at their best when used in unary contexts.

[Danvy & Filinski 89] Olivier Danvy, Andrzej Filinski, *A Functional Abstraction of Typed Contexts*, DIKU report 89/12, DIKU, University of Copenhagen, Denmark, August 1989.

[Danvy & Filinski 90] Olivier Danvy, Andrzej Filinski, *Abstracting Control*, ACM conference on Lisp and Functional Programming, Nice, France, Juin 1990.

[Felleisen & Wand & Friedman & Duba 88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, Bruce Duba, *Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps*, ACM conference on Lisp and Functional Programming, Snowbird, Utah, July 1988.

[Felleisen 88] Matthias Felleisen, *The Theory and Practice of First-Class Prompts*, ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, CA, january 1988.

[Friedman & Wand 84] Daniel P. Friedman, Mitchell Wand, *Reification: Reflection without Metaphysics*, Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, pp 348–355.

[Gabriel 89] Richard P. Gabriel, *Using CLOS-like Concepts in a Prototyping System*, Common Lisp Object System Workshop, OOPSLA '89, October 1989.

[Hanson 90] Chris Hanson, *Efficient Stack Allocation for Tail-Recursive Languages*, ACM conference on Lisp and Functional Programming, Nice, France, Juin 1990.

[Haynes & Friedman 87a] Christopher T. Haynes, Daniel P. Friedman, *Abstracting Timed Preemption with Engines*, Computer Languages, Volume 12, # 2, 1987, pp 109–121.

[Haynes & Friedman 87b] Christopher T. Haynes, Daniel P. Friedman, *Embedding Continuations in Procedural Objects*, ACM TOPLAS, Volume 9, # 4, October 1987, pp 582–598.

[Hieb & Dybvig & Bruggeman 90] Robert Hieb, R. Kent Dybvig, Carl Bruggeman, *Representing Control in the Presence of First-Class Continuations*, Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June 1990.

[Hieb & Dybvig 90b] Robert Hieb, R. Kent Dybvig, *Continuations and Concurrency*, Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1990, pp 128–136.

[Johnson 87] Gregory F. Johnson, *GL – A Denotational Testbed with Continuations and Partial Continuations*, proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, Saint-Paul, MA, June 1987.

[Johnson & Duggan 88] Gregory F. Johnson, Dominic Duggan, *Stores and Partial Continuations as First-Class Objects in a Language and its Environment*, ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, CA, january 1988.

[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.

[Saint-James 84] Emmanuel Saint-James, *Recursion is More Efficient than Iteration*, 1984 ACM symposium on Lisp and Functional Programming, Austin, Texas.

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

[Sitaram & Felleisen 90] Dorai Sitaram, Matthias Felleisen, *Control Delimiters and Their Hierarchies*, Lisp and Symbolic Computation: An International Journal, Volume 3, # 1, January 1990, pp 67–99.

[Steele 90] Guy L. Steele Jr., *Common Lisp, the Language*, Second Edition, Digital Press, Burlington MA, 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.

Appendix: Embedding `splitter` in standard Scheme

In [Sitaram & Felleisen 90] was explained how to program `control/run` in Scheme. We like this idea since it helps to popularize the concept of partial continuations. The embedding that follows has been designed to mimic a native implementation.

A call to `call/cc` must be thought of just as a copy of the stack-pointer. Push and pop operations are simulated with `cons` and `cdr`. Other allocations in the heap are named with a "`make-`" prefix. We make explicit the data that are necessary and do not use implicit closures to keep them.

```
(define stack-of-marker '())

(define (splitter f)
  (let ((marker '())
        (v '())
        (topmarker '()) )
     (set! v
       (call/cc
        (lambda (kk)
          (set! marker (cons kk '()))
          (set! stack-of-marker
              (cons marker stack-of-marker))
          (let ((v (f
                    (make-abort marker)
                    (make-call/pc marker))))
            (set-car! (car stack-of-marker)
                      '() )
            v ) ) ) ) )
     (if (not (null? (caar stack-of-marker)))
        ;;Someone did (kk thunk)
         (begin
          ;;markers downto marker are obsolete.
          (obsolete-stack! marker)
          ;;Compute thunk with continuation kk.
          (set! v (v))
          (set-car! (car stack-of-marker)
                    '() ) ))
     (set! topmarker (car stack-of-marker))
     (set! stack-of-marker
           (cdr stack-of-marker) )
     (cond
      ;;The continuation is 'return'
      ((null? (cdr topmarker)) v)
      ;;end of a partial continuation.
      (else ((cdr topmarker) v)) ) ) )

(define (make-abort marker)
  (lambda (thunk)
    (if (null? (car marker))
      (wrong "obsolete splitter")
      ;;Return to the marker.
      ((car marker) thunk) ) ) )

(define (make-call/pc marker)
  (lambda (g)
    (if (null? (car marker))
      (wrong "out of extent")
      (call/cc
       (lambda (kj)
         (g (make-pc kj marker)) )))))

(define (make-pc kj marker)
  (let ((slice (marker-prefix
                stack-of-marker
                marker )))
    (lambda (v)
```

```
      (call/cc
       (lambda (kc)
         (set! stack-of-marker
           (append slice
                   (cons (cons #t kc)
                         stack-of-marker ) ) )
         (kj v) ) ) ) ) )

(define (marker-prefix l m)
  (if (eq? (car l) m)
      '()
      (cons (cons #t (cdar l))
            (marker-prefix (cdr l) m) )))

(define (obsolete-stack! marker)
  (if (eq? (car stack-of-marker) marker)
    marker
   (begin (set-car! (car stack-of-marker)
                    '() )
          (set! stack-of-marker
                (cdr stack-of-marker) )
          (obsolete-stack! marker) ) ) )
```